MAPLE
User's Manual
Third Edition

Keith O. Geddes
Gaston H. Gonnet
Bruce W. Char

Research Report CS-83-41

December 1983

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

MAPLE

User's Manual

Third Edition

Keith O. Geddes

Gaston H. Gonnet

Bruce W. Char

December 1983

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

PREFACE

The design and implementation of the Maple system are currently in progress. This manual is a working document for the project. The version of the Maple system which has been completed at the time of publication of this third edition of the manual (December 1983) is version 3.1.

The Maple project was first conceived in the fall of 1980 as the logical outcome of discussions on the state of symbolic computation at the University of Waterloo. The authors wish to acknowledge many fruitful discussions with colleagues at the University of Waterloo, particularly Morven Gentleman, Michael Malcolm and Frank Tompa. It was recognized in these discussions that none of the locally-available systems for symbolic computation provided the environment nor the facilities that should be expected for symbolic computation in the 1980's. We concluded that since the basic design decisions for current symbolic systems such as ALTRAN, CAMAL, Reduce, and MACSYMA[Hal71a, Bou71a, Hea71a, Mar71a] were made more than ten years ago, it would make sense to design a new system from scratch taking advantage of the software engineering technology that has become available since then, as well as drawing from the lessons of experience.

Like other algebraic manipulation systems, Maple's basic features (e.g. elementary data structures, input/output, rational number arithmetic, and elementary simplification) are coded in a systems programming language for efficiency. For users, there is a high-level language with an Algol68-like syntax more suitable for describing algebraic algorithms. An important property of Maple is that most of the algebraic facilities are defined using the high-level user language. The basic system is sufficiently compact and efficient to be practical to use in a present-day time-sharing environment while providing a useful array of facilities. To this basic system can be added successive levels of 'function packages', each of which adds more facilities to the system as may be required, such as polynomial factorization or the Risch integration algorithm. The modularity of this design should allow users latitude in selecting which algebraic facilities they wish to have.

The basic system is written in a language belonging to the BCPL/B/C family. The Margay macro processor[Joh83a, Mera] is used to generate versions of the source code in B (for Honeywell TSS) and C (for the Vax UNIX† system). It is anticipated that very high level use of Maple (e.g., the Risch integration algorithm) will be impractical in a heavily-used time-sharing environment. Such use will be more practical on a dedicated microprocessor with one or more megabytes of main memory. Currently, the Maple system has been installed on several Motorola 6800 based systems. These include the Pixel system, the WICAT system, the Spectrix system, and the Sun workstation.

† UNIX is a Trademark of Bell Laboratories.

CONTENTS

1. INTRODUCTION

Maple is a mathematical manipulation language. (The name can be said to be derived from some combination of the letters in the preceding phrase, but in fact it was simply chosen as a name with a Canadian identity). The type of computation provided by Maple is known by various other names such as 'algebraic manipulation' or 'symbolic computation'. A basic feature of such a language is the ability to, explicitly or implicitly, leave the elements of a computation unevaluated. A corresponding feature is the ability to perform 'simplification' of expressions involving unevaluated elements.

In Maple, statements are normally evaluated as far as possible in the current 'environment'. For example the statement

```
a := 1;
```

assigns the value 1 to the name a. If this statement is later followed by the statement

```
x := a + b;
```

then the value 1+b is assigned to the name x. Next if the assignments

```
b := -1;  f := sin(x);
```

are performed then x evaluates to 0 and the value 0 is assigned to the name f. (Note that sin(0) is automatically 'simplified' to 0). Finally if we now perform the assignments

```
b := 0;  g := sin(x);
```

then x evaluates to 1 and the value sin(1) is assigned to the name g. (Note that sin(1) cannot be further evaluated or simplified in a symbolic context, but there is a facility to evaluate an expression into floating point form which will yield the decimal expansion of sin(1) to a number of digits controlled by the user).

As each complete statement is entered by a user, it is evaluated and the results are printed on the output device, usually the user's terminal. The printing of expression may be formatted on several lines to give as nice a display of the expression as is possible. There are several ways in which the user can alter the display of There are several ways in which the user can alter the display of If the global variable 'prettyprint' is assigned a value of 0, then expressions are printed on a single line. Setting the value of 'prettyprint' to 1 resets the prettyprinting mode. The global variable 'printlevel' determines how much output may be printed. Setting this variable to -1 prevents the printing of any results. This may be helpful in silently reading a file of Maple statements. Finally, the silent statement terminator ':', when used to terminate a statement also prevents the results of the statement from being displayed. (The normal statement terminator used to separate statements is ';'.) A sample Maple session is shown in section 3.3.

2. LANGUAGE ELEMENTS

2.1. Character Set

The Maple character set consists of letters, digits, and special characters. The letters are the 26 *lower case letters*

a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

and the 26 *upper case letters*

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.

The 10 *digits* are

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

and the 25 *special characters* are

	blank	[	left bracket
;	semicolon	]	right bracket
:	colon	{	left brace
=	equal	}	right brace
+	plus	`	grave accent
−	minus	'	single quote
*	asterisk	"	double quote
/	slash	<	less-than
!	exclamation	>	greater-than
.	period	_	underscore
,	comma	@	at-sign
(	left parenthesis	#	sharp
)	right parenthesis	^	caret

2.2. Tokens

The tokens consist of keywords, operators, strings, natural integers, and punctuation marks.

The *keywords* are the following reserved words:

break	by	do	done
elif	else	end	fi
for	from	if	local
od	option	options	proc
quit	read	save	stop
then	to	while	

The *operators* consist of the *binary* operators

+	addition; set union
−	subtraction; set difference
*	multiplication; set intersection
/	division
**	exponentiation
^	exponentiation
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
=	equal
<>	not equal
and	logical and
or	logical or
:=	assignment
.	concatenation; decimal point
..	ellipsis (more generally, ...*)

the *unary* operators

+	unary plus (prefix)
−	unary minus (prefix)
!	factorial (postfix)
not	logical not (prefix)
.	decimal point (prefix or postfix)

and the *nullary* operators

"	last expression
""	penultimate expression
"""	before penultimate expression

(Note that three of the operators are reserved words: and, or, not).

The simplest instance of a *string* is a letter followed by zero or more letters, digits, and underscores. Another instance of a string is the at-sign (@) followed by

* two *or more* consecutive periods are parsed as an ellipsis.

zero or more letters, digits, and underscores. More generally, a string can be formed by enclosing in grave accents any sequence of characters (except a grave accent). In all cases, the maximum length of a string in Maple is 499 characters. A string is a valid name (e.g. a variable name or a function name) but the user should not indiscriminately use names involving the at-sign (@) since such names are used globally by the Maple system. A Maple string is also used in the sense of a 'character string', usually by enclosing a sequence of characters in grave accents.

A *natural integer* is any sequence of one or more digits. The basic constants in Maple (integers, rational numbers, and floating point numbers) are formed from the natural integers using operators. The length of a natural integer (and hence the length of integers, rational numbers, and floating point numbers) is arbitrary (i.e. the length limit is system dependent but generally much larger than users will encounter).

The *punctuation marks* are

;	semicolon	,	comma
:	colon	[	left bracket
'	single quote	]	right bracket
`	grave accent	(	left parenthesis
{	left brace	)	right parenthesis
}	right brace		

The semicolon and the colon are used to separate statements. The only difference is that during an interactive session if a statement is followed by a colon, the result of the statement is not printed. The comma is used to separate expressions in an expression sequence (as in a function call or in specifying a list or a set). Enclosing an expression in a pair of single quotes specifies that the expression is to be unevaluated. The grave accent is used in forming strings. The left and right parentheses have their familiar uses in grouping terms in an expression and in grouping parameters in a function call. The left and right brackets are used to form subscripted names. The left and right brackets are also used to form lists and the left and right braces are used to form sets.

2.3. Blanks, Lines, and Comments

The *blank* is a character which separates tokens, but is not itself a token. Blanks cannot occur within a token but otherwise blanks may be used freely. The one exception to the free use of blanks is when forming a string by enclosing a sequence of characters within grave accents, in which case the blank is significant like any other character.

Input to the Maple system consists of a sequence of statements separated by the statement separators, which are the semicolon and the colon characters. The system operates in an interactive mode, executing statements as they are entered. A *line* consists of a sequence of characters followed by <return>. A single line may contain several statements or it may contain an incomplete statement (i.e. a statement to be completed on succeeding lines), or it may contain several statements followed by an incomplete statement. A statement will normally be recognized as complete only when a statement separator is encountered, except in the cases of the quit statement, the break statement, the if ... fi construct and the do ... od construct. When a line is entered, the system evaluates (executes) the statements (if any) which have been completed on that line.

On a line, all characters which follow a sharp character (#) are considered to be part of a *comment*. The comment is echoed by the system. However the sharp character is not treated as the beginning of a comment if it is enclosed within a pair of quote marks.

2.4. Files

The file system is an important part of the Maple system. The user interacts with the file system either explicitly by way of the **read** and **save** statements, or implicitly by specifying a function name corresponding to a file which the system will read in automatically.

A *file* consists of a sequence of statements either in 'Maple internal format' or in 'user format'. If the file is in user format then the effect of reading the file is identical to the effect of the user entering the same sequence of statements. The system will display the result of executing each statement which is read in from the file. On the other hand, if the file is in Maple internal format then reading the file causes no information to be displayed to the user but updates the current Maple environment with the contents of the file. Maple assumes that a file will be in Maple internal format when its file name ends with the characters '.m' . For example, some typical names for files in user format are:

```
temp
`/lib/src/gcd`
```

while some typical names for files in internal format are:

```
`temp.m`
`/lib/gcd.m`
```

(Note that file names involving characters such as '/' or '.' must be enclosed in grave accents in order to be interpreted properly as <name>s).

The contents of a file in user format are written into the file either from a text editor external to Maple or else from Maple by using the **save** statement (no '.m' suffix in the file name). The contents of a file in Maple internal format are written into the file from Maple by using the **save** statement ('.m' suffix in the file name). Either type of file may be read into a Maple session by using the Maple **read** statement. Some Maple functions are not part of the basic Maple system which is loaded in initially, but rather reside in files in the Maple library. When one of these functions is invoked in Maple, the corresponding file is automatically read into the Maple session in Maple internal format. (See section 8 -- Library Functions).

3. STATEMENTS AND EXPRESSIONS

3.1. Types of Statements

There are nine types of statements in Maple. They will be described informally here. The formal syntax is given in section 4.2.

3.1.1. Assignment Statement

The form of this statement is

```
<name> := <expression>
```

and it associates a name with the value of an expression.

3.1.2. Expression

An <expression> is itself a valid statement. The effect of this statement is that the expression is evaluated.

3.1.3. Read Statement

The statement

```
read <expression>
```

causes a file to be read into the Maple session. The expression must evaluate to a name which is a valid file name in the host system. The file name may be one of two types as discussed in section 2.4. A typical example of a read statement is

```
read `/u/dmackenz/lib/f.m`
```

where the grave accents are necessary so that the expression evaluates to a name.

3.1.4. Save Statement

The statement

```
save <expression>
```

causes the current Maple environment to be written into a file. The expression must evaluate to a name which is a valid file name in the host system. If the file name ends with the characters '.m' then the environment is saved in Maple internal format, otherwise the environment is saved in user format.

3.1.5. Selection Statement

The selection statement takes one of the following general forms. Here <expr> is an abbreviation for <expression> and <statseq> stands for a sequence of statements.

```
if <expr> then <statseq> fi
if <expr> then <statseq> else <statseq> fi
if <expr> then <statseq> elif <expr> then <statseq> fi
if <expr> then <statseq> elif <expr> then <statseq> else <statseq> fi
```

Wherever the construct 'elif <expr> then <statseq>' appears in the above forms, this construct may be repeated any number of times to yield a valid selection statement. The sequence of statements in the branch selected (if any) is executed.

3.1.6. Repetition Statement

The syntax of the repetition statement is as follows, where <expr> and <statseq> are as above.

```
for <name> from <expr> by <expr> to <expr> while <expr>
     do <statseq> od
```

where any of 'for <name>', 'from <expr>', 'by <expr>', 'to <expr>', or 'while <expr>' may be omitted. The sequence of statements in <statseq> is executed zero or more times. The 'for <name>' part may be omitted if the index of iteration is not required in the loop, in which case a 'dummy index' is used by the system. If the 'from <expr>' part and/or the 'by <expr>' part are omitted then the default values 'from 1' and/or 'by 1' are used. If the 'to <expr>' part and/or the 'while <expr>' part are present then the corresponding tests for termination are checked at the beginning of each iteration, and if neither is present then the loop will be an infinite loop unless it is terminated by the execution of the **break** statement (see section 3.1.7), or the **quit** statement (see section 3.1.8), or by the execution of a return from a procedure (see section 6.5).

3.1.7. Break Statement

The syntax of the break statement is

```
break
```

and it causes an immediate exit from the innermost repetition statement within which it occurs. It is an error if the break statement occurs at a place which is not within a repetition statement.

3.1.8. Quit Statement

The syntax of the quit statement is any one of the following three forms:

```
quit
done
stop
```

The result of this statement is to terminate the Maple session and return the user to the system level from which Maple was entered. (In the Vax UNIX and Honeywell TSS versions of Maple, hitting the break/interrupt key twice in rapid succession will also exit from Maple).

3.1.9. Empty Statement

The empty statement is syntactically valid in Maple. For example

```
a := 1; ; quit
```

is a valid statement sequence in Maple consisting of an assignment statement, the empty statement, and the quit statement. Of course since blanks may be freely used, any number (including zero) of blanks could appear between the semicolons here yielding a syntactically identical statement sequence.

3.2. Expressions

Expressions are the fundamental entities in the Maple language. The various types of expressions are described informally here. The formal syntax is given in section 4.2.

3.2.1. Basic Constants

The basic constants in Maple are integers, rational numbers, and floating point numbers. A <natural integer> is any sequence of one or more digits of arbitrary length (i.e. the length limit is system dependent but generally much larger than users will encounter). An *integer* is a <natural integer> or a signed integer (i.e. +<natural integer> or −<natural integer>). A *rational number* is of the form

```
<integer> / <natural integer>
```

(Note that a rational number is always simplified so that the denominator is unsigned, and it will also be reduced to lowest terms).

An <unsigned float> is one of the following three forms:

```
<natural integer> . <natural integer>
<natural integer> .
. <natural integer>
```

A *floating point number* is an <unsigned float> or a signed float (i.e. +<unsigned float> or −<unsigned float>). The **evalf** function is used to force an expression to be evaluated to a floating point number (if possible). The number of digits carried in the 'mantissa' when evaluating floating point numbers is determined by the value of the global name 'Digits' which has 10 as its initial value. Note that the current version of Maple displays floating point numbers with very small or very large magnitudes using the notation Float(mantissa,characteristic) which corresponds to the internal data structure. For example, evalf(exp(−10)) yields Float(4539992971,−14) which represents the number

```
4539992971 * 10^(−14)
```

while evalf(exp(−2)) yields .1353352832 .

3.2.2. Names

A name in Maple has a value which may be any expression or, if no value has been assigned to it, then it stands for itself. A name is usually a <string>, which in its simplest form is a letter followed by zero or more letters, digits, and underscores (with a maximum length of 499 characters). Keywords may not be used as names formed in this manner. Note that lower case letters and upper case letters are distinct, so that the names

```
g    G    new_term    New_Term    x13a    x13A
```

are all distinct. Another type of <string> is formed by the at-sign (@) followed by zero or more letters, digits, and underscores. Names beginning with the at-sign are used as global variable names by the Maple system and therefore should not be used indiscriminately by users.

A <string> can also be formed by enclosing in grave accents any sequence of characters (except the grave accent). Keywords enclosed in grave accents may be used as names, but we recommend against using such dubious names. The following are valid strings (and hence names) in Maple:

```
`This is a strange name:`    `2D`    `n-1`
```

The grave accents do not themselves form part of the string so they disappear when the string has been input to Maple. For example, if n has the value 5 then the statement

```
`n-1` := n-1;
```

will yield the following response from Maple:

```
n-1 := 4
```

The user should beware of misusing this facility for string (name) formation to the point of writing unreadable programs!

More generally, a <name> may be formed using the *concatenation operator* in one of the following three forms:

```
<name> . <natural integer>
<name> . <string>
<name> . ( <expression> )
```

Some examples of the use of the concatenation operator for <name> formation are:

```
v.5    p.n    a.(2*i)    V.(N.(i-1))    r.i.j
```

The concatenation operator is a binary operator which requires a <name> as its left operand. Its right operand is evaluated and then concatenated to the left operand. For example if n has the value 4 then p.n evaluates to the name p4, while if n has no value then p.n evaluates to the name pn. Similarly if i has the value 5 then a.(2*i) evaluates to the name a10. As a final example if N4 has the value 17 and i has the value 5 then V.(N.(i-1)) evaluates to the name V17, while V.N.(i-1) evaluates to the name VN4 (assuming that N has no value).

3.2.3. Expression Sequence

An *expression sequence* is an expression of the form

$<\text{expression}_1>$, $<\text{expression}_2>$, ... , $<\text{expression}_n>$

The comma operator is used to concatenate expressions into expression sequences. It has the lowest precedence of all operators. When expression sequences are concatenated, the result is simplified to a single, un-nested expression sequence.

A zero-length expression sequence is syntactically valid only in certain constructs, namely: an empty list, an empty set, a function call with no parameters, or an indexed name with no subscripts. The special name 'NULL' is initially assigned a zero-length expression sequence which can be used in any expression.

Examples:

a := A, B, C, D;	assigns to 'a' a 4 element expression sequence
b := NULL;	assigns to 'b' a 0 element expression sequence
c := a, b, a;	assigns to 'c' an 8 element expression sequence
f := proc() args[6] end;	
f(c);	yields B

3.2.4. Sets and Lists

A *set* is an expression of the form

{ <expression sequence> }

and a *list* is an expression of the form

[<expression sequence>] .

Note that an <expression sequence> may be empty so that the empty set is represented by { } and the empty list is represented by [ ]. A set is an *unordered* sequence of expressions and the user should not assume that the expressions will be maintained in any particular order. (A Maple system will use a particular ordering that may be convenient for its implementation.) A list is an *ordered* sequence of expressions with the order of the expressions specified by the user. For example, if the user inputs the set x, y, y the system might respond with the representation y, x while if the user inputs the list [x, y, y] then the representation used by the system will be precisely this list.

3.2.5. Indexed Expressions

One form of expression in Maple is the indexed expression. The input syntax is

<name> [<expression sequence>]

For an indexed expression, the zeroth operand is the <name> and the i^{th} operand is the i^{th} element of the <expression sequence>.

An indexed expression is syntactically legal anywhere a name is. It follows that an indexed expression may also be input using the syntax

<indexed expression> [<expression sequence>]

The use of the indexed name b[1] does not imply that 'b' is an array, as is true in many other languages. The statement

a := b[1] + b[2] + b[1000];

simply forms the sum of three indexed names. It is not necessary that 'b' have an array value. (However, if 'b' does evaluate to an array or a table, then 'b[1]' is the element of the array or table respectively selected by '1'.) The assignment of a value to an indexed expression will implicitly create a table.

b[1] := 10;

If 'b' had no value previously, then this assignment statements creates a table for 'b' and initializes it with the value of 10 for the index value 1. (c.f. Arrays and Tables).

3.2.6. Algebraic Operators

There are ten *algebraic operators:*

", "", """, !, +, −, *, /, **, ^

The nullary operator " has as its value the latest expression, the nullary operator "" has as its value the penultimate expression, and the nullary operator """ has as its value the expression preceding the penultimate expression. The unary operator ! is used as a postfix operator and it denotes the factorial function of its operand. + and − may be used as prefix operators representing unary plus and unary minus. The latter six operators all may be used as binary operators, representing addition, subtraction, multiplication, division, exponentiation, and exponentiation respectively. The two operators ** and ^ are synonymous and may be used interchangably.

The operators +, −, and * have a different semantics when their operands are sets, in which case they denote set union, set difference, and set intersection, respectively. For example, if the following statements are executed:

```
set1 := {x+y, x, y};  set2 := {y, y−x};
a := set1 + set2;  b := set1 − set2;  c := set1 * set2;
```

then the value of a is {y, x, y−x, x+y}, the value of b is {x, x+y}, and the value of c is {y}.

The order of precedence of all operators is described in section 3.2.12 below. However, any expression may be enclosed in parentheses yielding a new valid expression and this mechanism can be used to force a particular order of evaluation.

3.2.7. Relations and Logical Operators

A new type of expression can be formed from ordinary algebraic expressions by using the *relational operators* <, <=, >, >=, =, <>. The semantics of these operators is dependent on whether they occur in an *algebraic* context or in a *boolean* context.

In an algebraic context, the relational operators are simply 'place holders' for forming equations or inequalities. Addition of equations and multiplication of an equation by a constant are fully supported in Maple. In the case of adding or subtracting two equations, the addition or subtraction is applied to each side of the equations yielding a new equation. In the case of multiplying an equation by a constant, the multiplication is distributed to each side of the equation. Other operations on equations can be performed, using the 'expand' function as required. No operations on inequalities are currently supported in Maple.

In a boolean context a relation is evaluated to the value 'true' or the value 'false'. In the case of the operators <, <=, >, >= the difference of the operands must evaluate to a constant and this constant is compared with zero. In the case of either of the relations

```
op1 = op2
op1 <> op2
```

the operands can be arbitrary algebraic expressions.

More generally, an expression can be formed using the *logical operators*

```
and
or
not
```

where the first two are binary operators and the third is a unary (prefix) operator.

In Maple, the names 'true' and 'false' have special meanings when they occur in boolean contexts but they are ordinary <name>s which may be freely manipulated. Any arbitrary expression may be used in a boolean context and if the expression does not evaluate to either the value of 'true' or the value of 'false' then a semantic error will be reported. Note that since 'true' and 'false' are ordinary names, it is possible to assign values to them. For example, a user could assign

```
true := 1;  false := 0;
```

and thereafter expressions which previously evaluated to 'true' or 'false' will evaluate to '1' or '0'. Normally users will leave the names 'true' and 'false' unassigned so that their values are their own names.

3.2.8. Ranges

Yet another type of expression is a *range* which is formed using the ellipsis operator:

```
<expression> .. <expression>
```

(the operator here can be specified as two *or more* consecutive periods). The ellipsis operator simply acts as a 'place holder' in the same manner as when the relational operators are used in an algebraic context.

Two common uses of ranges are for Maple's built-in functions sum and int. For example, in the function call

```
sum( i^2, i = 1..n )
```

the sum function interprets this to mean that the lower and upper limits of summation are 1 and n, respectively. Similarly, in the function call

```
int( exp(2*x), x = 0..1 )
```

the integration function interprets this as a definite integration with lower and upper limits of integration 0 and 1, respectively. The range construct is also used by Maple's built-in function op, which extracts operands from an expression. For example, if

```
a := [ x, y, z, w ];
```

then op(2,a) yields y, op(3,a) yields z, and op(2..4,a) yields the expression sequence y,z,w (which might be formed into a new list as [op(2..4,a)] since an <expression sequence> is not itself a valid <expression> in Maple). A final example of the use of a <range> is the construct

```
<name> . ( <range> )
```

which is a generalization of the name-formation construct using the concatenation operator. This construct produces an <expression sequence> which, as we have noted, is not itself a valid <expression> but it can be used wherever an <expression sequence> is valid. For example,

```
print( p.(1..5) )
```

is exactly equivalent to

```
print( p1, p2, p3, p4, p5 ) .
```

3.2.9. Selection Operator

The selection operator '[]' can be used to select components from an aggregate object. The form for a selection operation is

```
<name> [ <arg> ]
```

The <name> must evaluate to one of the following:

name, table, array, list, set, expression sequence

If <name> evaluates to an unassigned name, then <name>[<arg>] is an indexed expression as described earlier (see section 3.2.5). If <name> evaluates to a table or an array, then the selection operation is an indexing operation. The use of arrays, tables, and indexing operations is deferred to section 5. If <name> evaluates to a list, set, or expression sequence, then either one or more components of such an aggregate object may be returned by the selection operation. In these cases, <arg> must evaluate to an integer, a range, or null. If <arg> is an integer i, then the i^{th} operand of the aggregate object is returned. (See also op().) If <arg> is a range, then an expression sequence is returned containing the operands of the aggregate object as specified by the range. If <arg> is empty, then an expression sequence containing all the operands (except the zeroth one) of the aggregate object is returned.

3.2.10. Unevaluated Expressions

An expression enclosed in a pair of single quotes is called an *unevaluated expression*. For example, the statements

```
a := 1;  x := a + b;
```

cause the value 1+b to be assigned to the name x while the statements

```
a := 1;  x := 'a + b';
```

cause the value a+b to be assigned to the name x. The latter effect can also be achieved (if b has no value) by the statements

```
a := 1;  x := 'a' + b;
```

The effect of evaluating a quoted expression is to strip off (one level of) quotes, so in some cases it is useful to use nested levels of quotes. Note that there is a distinction between 'evaluation' and 'simplification' (see section 7) so that the statement

```
x := '2 + 3';
```

will cause the value 5 to be assigned to the name x even though the expression appearing here is quoted. This is because the 'evaluator' simply strips off the quotes but it is the 'simplifier' which transforms the expression 2 + 3 into the constant 5. Simplification can be avoided in a case like this by using two levels of single quotes:

```
x := ''2 + 3'';
```

in which case the result of evaluating the right hand side will be the unevaluated expression '2 + 3' which will be left unchanged by the simplifier.

A special case of 'unevaluation' arises when a name which may have been assigned a value needs to be unassigned, so that in the future the name simply stands for itself. This is accomplished by assigning the quoted name to itself. For example, if the statement

```
x := 'x';
```

is executed, then even if x had previously been assigned a value it will now stand for itself in the same manner as if it had never been assigned a value.

3.2.11. Procedures

Another valid expression in Maple is a *procedure definition* which takes the form

```
proc ( <nameseq> ) local <nameseq>; options <nameseq>; <statseq> end
```

where the 'local <nameseq>;' part and/or the 'options <nameseq>;' part may be omitted, and where <nameseq> stands for a (possibly empty) sequence of <name>s. This construct has some similarities with the concept of unevaluated expressions, but in this case it is more generally a <statseq> (i.e. a sequence of statements) which is unevaluated. Note that the keywords 'proc' and 'end' serve a purpose similar to the single quotes in unevaluated expressions (except that evaluation of this expression does not cause these keywords to be stripped off). An example of a procedure definition is

```
max := proc ( a, b ) if a>b then a else b fi end
```

which is syntactically an assignment statement where the <expression> on the right hand side is a procedure definition.

A procedure is invoked by using the syntax

```
<name> ( <expression sequence> )
```

which is another instance of an expression. For example if max is defined as above then the expression max(1, 2) causes a procedure invocation in which the 'actual parameters' 1 and 2 are substituted for the 'formal parameters' a and b, respectively, and then the 'procedure body' is executed yielding the value 2 in this case. The syntax of a procedure invocation may also be used in cases where the <name> has not been assigned, in which case the result is an *unevaluated function*, such as sin(x) or exp(x^2). (A more general discussion of procedures will be postponed until section 6).

3.2.12. Precedence of Operators

The order of precedence of all unary and binary operators is listed in the following table, from highest to lowest binding strengths. In parentheses it is stated whether the operators are left associative, right associative, or non-associative.

.	(left associative)
!	(left associative)
**	(non-associative)
^	(non-associative)
*, /	(left associative)
+, −	(left associative)
..	(non-associative)
<, <=, >, >=, =, <>	(non-associative)
not	(right associative)
and	(left associative)
or	(left associative)
,	(left associative)
:=	(non-associative)

Thus the concatenation or decimal point operator '.' has the highest binding strength and the assignment operator ':=' has the lowest binding strength. Note that the exponentiation operators '**' and '^' are defined to be non-associative and therefore a^b^c is syntactically invalid in Maple. (The user must use parentheses to state his intentions).

The evaluation of expressions involving the logical operators proceeds in an intelligent manner which exploits more than the simple associativity and precedence of these operators. Namely, the left operand of the operators 'and' and 'or' is always evaluated first and the evaluation of the right operand is avoided if the truth value of the expression can be deduced from the value of the left operand alone. For example, the construct

```
if d <> 0 and f(d)/d > 1 then . . . fi
```

will not cause a division by zero because if d=0 then the left operand of 'and' becomes false and the right operand of 'and' will not be evaluated.

3.3. Sample Maple Session

This section presents a sample interactive session using the Maple system. At the University of Waterloo, Maple is initiated on the Vax UNIX system by issuing the command '/u/maple/bin/maple' and is initiated on Honeywell TSS by issuing the command 'maple/maple'. In the following presentation of the Maple session, all lines containing italic characters are user input lines and all other lines are system responses. Each user input line must be terminated by <return>.

```
# Initiate the Maple system using the command
#     /u/maple/bin/maple    if on Vax UNIX,
#     maple/maple    if on Honeywell TSS.

     |\^/|
._|\|   |/|_.
 \  MAPLE  /  Version 3.1 --- October 1983
 <____ ____>
       |

# Integers, rational numbers, and floating point numbers.

254 + 5280*99999;
527994974
3!;
6
3!!;
720
1 + 1/4 + 1/16 + 1/64 + 1/256;
341/256
evalf(");
1.33203125000
a := (5^40 + 3^50) / 2^90;
a := 4547832457858487115869580437/618970019642690137449562112
evalf(a);
7.3474196060
Digits := 40;
Digits := 40
evalf(a);
7.3474196060154781089322604350878881161323
evalf(a, 60);
7.347419606015478108932260435087888116132378478690193035482991
b := 2^90;
b := 1237940039285380274899124224
a*b;
9095664915716974231739160874

# Names, including the concatenation operator.

g := 52;  G := 4;
g := 52
G := 4
g*G;
208
for i to 5 do p.i := i^2 od;
```

```
p1 := 1
p2 := 4
p3 := 9
p4 := 16
p5 := 25
print(p.(1..5));
1  4  9  16  25
x := 333;  x := 'x';
x := 333
x := x

# Sets and lists.

set1 := {x, 2*y+1/3};  set2 := {z-4, x};
set1 := {x,2*y+1/3}
set2 := {x,z-4}
set1 + set2;
{x,z-4,2*y+1/3}
set1 * set2;
{x}
set3 := set1 - set2;
set3 := {2*y+1/3}
set2 * set3;
{}
list1 := [x, 2*y+1/3];  list2 := [z-4, x];
list1 := [x,2*y+1/3]
list2 := [z-4,x]
new_list := [op(list1), op(list2)];
new_list := [x,2*y+1/3,z-4,x]

# Polynomials and rational functions.

p := x^2 - x - 2;
p := x^2-x+(-2)
q := (x+1)^2;
q := (x+1)^2
r := "" * ";
r := (x^2-x+(-2))*(x+1)^2
expand(r);
x^4+x^3-3*x^2-5*x+(-2)
s := p/q;
s := (x^2-x+(-2))*(x+1)^(-2)
normal(s);
(x+(-2))/(x+1)
r*s;
```

```
(x^2-x+(-2))^2

# Equations.

eqn1 := 3*p + 5*q = 13;
eqn1 := 3*p+5*q=13
eqn2 := 4*p - 7*q = 30;
eqn2 := 4*p-7*q=30
3*eqn2 - 4*eqn1;
-41*q=38
q := 38/(-41);
q := -38/41
eqn1;
3*p+(-190/41)=13
p := (13 + 190/41)/3;
p := 241/41
eqn1; eqn2;
13=13
30=30
solve( {5*x + 10*y = 97, x - y = 12}, {x,y} );
{y=37/15,x=217/15}

# Unevaluated expressions and procedures.

a; b;
45478324578584871158695804 37/61897001964269013744956 2112
1237940039285380274899124224
f := 'b * (a+5)';
f := b*(a+5)
f;
1528536511214387560623478 1994
max := proc (a,b) if a>b then a else b fi end;
max := proc (a,b)if args[2]<args[1] then args[1] else args[2] fi end
max(a,b);
1237940039285380274899124224
max(25/7, 525/149);
25/7
```

```
# Integers can be arbitrarily long.
# Here is one that almost fills one screen on a typical video terminal.
# Recall from above that 3!! (i.e. 6!) is 720
#   so the following statement yields the same result as 720! .

3!!!;

2601218943565795100204903227081043611119152187501694578572754183785083563115694
822406785779581304570826199205758922472595366415651620520158737919845877408325
105244690388811884123764341191951045505346658616243271940197113909845536727278
709934562985558671936977407000370043078375899742067678401696720784628062922903
071616698672605489884455142571939854994489395944960640451323621402659861930732
369770477606067680670176491669403034819961881455625195592566918830825514942947
653727484562462882423452659778973774089646655399243592878621251596748322097602
056966999272846705637471375330192483135870761254126834158601294475660114554207
589952563543068288634631084965650682771552996256790845235702552186222358130016
083452344323682193579318470195651072978180435417389056072742804858399591972902
266122912984205160675790362323376994539641914751755675576953922338030568253085
977441675784352815913461340394604901269542028838347101363733824484506660093348
444071193129253769465735433737572477223018153403264717753198453734147867432704
579837866187032574059389242157096959946305575210632032634932092207383209233563
923267504401701760572026010829288042335606643089888710297380797578013056049576
283868305719066220529117482251053669775660302957404338798347151855260280533386
571391010463364197690973974322859942198370469791099563033896046758898657957111
566670039156748153115943980043625399399731203066490601325311304719028898491856
376666916446879112524919375442584589500031156168297430464114253807489728172337
5538066171980140467793561479363526626568333950976000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000

quit
```

4. DATA TYPES AND FORMAL SYNTAX

4.1. Data Types

Every expression in Maple is represented internally by an expression tree where each node is a particular data type. While some data types are strictly for internal use, most of the data types corresponding to expressions are accessible to the user and can be tested for via the *type* function. The user can examine the components of such a data type by using the *op* function. In this section we discuss the data types that are accessible to the user. For a more detailed description of the internal data types, see section 7.

4.1.1. Integer

An expression is of type 'integer' if it is an (optionally signed) sequence of one or more digits of arbitrary length (i.e. the length limit is system dependent but generally much larger than users will encounter). The 'op' function considers this data type to have only one operand, so if n is an integer then the value of op(n), and also the value of op(1, n), is the integer n.

4.1.2. Rational Number

A rational number (called type 'rational') is represented by a pair of integers (numerator and denominator) with all common factors removed and with a positive denominator. Like integers, rational numbers are of arbitrary length. The 'op' function considers this data type to have two operands, where the first operand is the numerator and the second operand is the denominator.

4.1.3. Floating Point Number

An expression of type 'float' is a number represented externally as a sequence of digits with a decimal point (e.g. 1.5, 15000., .15). Floating point numbers are represented internally by a pair of integers (the mantissa and the characteristic), which represent the number $mantissa \times 10^{characteristic}$. Thus op(150.1) yields the expression sequence 1501, -1. Arithmetic with floating point numbers is performed via the *evalf* function. The number of digits carried in the mantissa when evaluating floating point numbers is determined by the value of the global name Digits which has 10 as its initial value. Note that the current version of Maple displays floating point numbers with very small or very large magnitudes using the notation Float(mantissa,characteristic) which corresponds to the internal data structure. For example, evalf(exp($-$10)) yields Float(4539992971,$-$14) while evalf(exp($-$2)) yields .1353352832 .

4.1.4. Name

An expression is of type 'name' if it is a <string> as defined by the Maple grammar. For example, if x has not been assigned a value then x is of type 'name' and op(x) has the value x.

4.1.5. Expression Sequence

There is an internal data type in Maple for an <expression sequence>, which is a sequence of <expression>s separated by commas. An <expression sequence> is not, by itself, a valid <expression> but it occurs in many places as a component of an <expression>. There is no type name known to the type function for this data type.

When the op function is used to extract parts of an expression, the result is often an expression sequence. For example,

```
a := [x,y,z,w];  op(a);
```

yields the expression sequence x,y,z,w. An important special case of an expression sequence is the null expression sequence and there is a global name in Maple, NULL, whose value is the null expression sequence. The value of the global name NULL is equivalent to the value of the operation op([ ]).

4.1.6. Set and List

Two more data types are the 'set' and the 'list'. Each of these types consists of a sequence of expressions and if expr is an object of either of these two types then op(expr) yields the expression sequence. The external representation of a set uses braces '{', '}' to surround the expression sequence and the external representation of a list uses brackets '[', ']' to surround the expression sequence.

4.1.7. Addition, Multiplication, and Power

An expression can be composed using the algebraic operators +, −, *, /, ^, **. Such an expression is of type \`+\`, type \`*\`, or type \`**\`. Thus the expression a − b is of type \`+\` and op(a − b) yields the expression sequence a, −b. Similarly the expression a/b is of type \`*\` and op(a/b) yields the expression sequence a, b**(−1). Of course, b**(−1) is an example of an expression of type \`**\`. The representation used for these algebraic expressions is often referred to as sum-of-products form.

4.1.8. Series

The 'series' data type in Maple is a special data type which represents an expression as a (truncated) power series in one specified indeterminate. This data type is created by a call to the *taylor* function. For this data type, the zeroth operand is defined to be the name of the indeterminate, the first, third, . . . operands are the coefficients (generally expressions), and the second, fourth, . . . operands are the corresponding exponents. The exponents are ordered from least to greatest. Usually, the final pair of operands in this data type are the special 'order' symbol O(1) and the integer n which indicates the order of truncation. (Note: The print routine displays the final pair of operands using the notation O(x**n) rather than more directly as O(1)*x**n, where x represents the zeroth operand). However, if the series is known to be exact then there will be no 'order' term in the series. An example of this occurs when the 'taylor' function is applied to a polynomial whose degree is less than the truncation degree for the series.

4.1.9. Equation and Inequality

An expression of type 'equation' (also called type `=`) has two operands, the left-hand-side expression and the right-hand-side expression. An equation is represented externally using the binary operator '='.

There are three internal data types for inequalities, corresponding to the operators '<>', '<', and '<='. Inequalities involving the operators '>' and '>=' are converted to the latter two cases for purposes of representation. Correspondingly, only three names are known to the type function for inequalities: `<>`, `<`, `<=` . Like an equation an inequality has two operands, the left-hand-side expression and the right-hand-side expression.

4.1.10. Boolean Expression

The simplest cases of Boolean expressions are the names true and false*. Equations and inequalities (formed using the relational operators =, <>, <, <=, >, >=) are also treated as Boolean expressions if they appear in a 'Boolean context'. More complicated Boolean expressions can be built out of these simple expressions with the logical operators **and**, **or**, and **not**. The built-in function *evalb* can be called with a Boolean expression as argument in order to cause the expression to be evaluated as a Boolean. For example, the equation a = b is an algebraic equation if it appears alone but evalb(a = b) will evaluate this equation as a Boolean. However, an equation or inequality will be recognized as being in a Boolean context if it appears in the 'while part' of a repetition statement or in the 'if part' of a selection statement. In addition to the type names for equations and inequalities, the following type names are also known to the type function: `and`, `or`, `not` .

4.1.11. Range

An expression of type 'range' (also called type `..`) has two operands, the left-hand-side expression and the right-hand-side expression. A range is represented externally using the binary operator '..' which simply acts as a place-holder.

4.1.12. Procedure Definition

A procedure definition in Maple is a valid expression and its type is called 'procedure'. The external representation of a procedure definition is

proc (<nameseq>) local <nameseq>; options <nameseq>; <statseq> end

The internal data structure represents each <nameseq> in the order shown above followed by the statement sequence <statseq>. Since <statseq> is not a valid expression in Maple, this part of the data structure is not retrievable by the *op* function. There are three operands defined for the *op* function applied to this data structure: the first operand is the <nameseq> of formal parameters, the second operand

*true and false are just Maple names that the system returns as the result of Boolean evaluation. Users can use true and false just like any other name, but to be safe it is best to avoid assigning values to these names.

is the <nameseq> of local variables, and the third operand is the <nameseq> of option names. Therefore, if

```
f := proc (a,b)
  local c;
  options remember;
  c := a/b;
  if type(c, integer) then c else FAIL fi
end;
```

then

```
op( 1, f );          yields   a,b
op( 2, f );          yields   c
op( 3, f );          yields   remember
op( f );             yields   a,b,c,remember
```

4.1.13. Unevaluated Function Invocation

A function invocation takes the form

```
<name> ( <expression sequence> )
```

and if <name> is undefined then the result is an unevaluated function invocation, called type 'function'. Typical examples of the type 'function' are sin(x), exp(x^2), g(a,b) where none of sin, exp, and g has been defined. For the *op* function applied to this data type, operand 0 is defined to be the name of the function and the remaining operands are the elements of the <expression sequence>. For example,

```
op( 0, g(a,b) );     yields   g
op( 1, g(a,b) );     yields   a
op( 2, g(a,b) );     yields   b
op( g(a,b) );        yields   a,b
```

4.1.14. Unevaluated Factorial

The factorial function is invoked through the use of the postfix operator '!'. If the operand to the factorial function does not evaluate to an integer then the result is an unevaluated function invocation of the type described above. For example,

```
type( n!, function );   yields   true
op( 0, n! );            yields   factorial
factorial( m );         yields   m!
type( j!, `!` )         yields   true
```

4.1.15. Unevaluated Concatenation

An expression which consists of an unevaluated concatenation is said to be of type `.` . Normally, the concatenation operator is evaluated to form a name but an example of an expression of type `.` would be the unevaluated expression 'a.i' . In the current version of Maple, if the name 'i' does not evaluate to an integer then the expression a[i] is another example of type `.` (i.e. an unevaluated concatenation).

4.2. Formal Syntax

This section presents the BNF grammar which describes the syntax accepted by Maple. In the following grammar, where a sequence of symbols is enclosed in a pair of "§" symbols it indicates that this portion of the statement is optional. Where *empty* occurs in the grammar, no symbol is required. A Maple *session* consists of a <statseq>, which is a sequence of statements separated by semicolons.

```
<statseq> ::=            <statseq> ; <stat>          |        <stat>

<stat> ::=                <expr>
                     |    <name> := <expr>
                     |    read <expr>
                     |    save <expr>
                     |    § for <name> §   § from <expr> §   § by <expr>§    § to <e
                                 § while <expr> §
                                           do
                                           <statseq>
                                           od
                     |    break
                     |    if <expr> then <statseq> <elsepart>
                     |    quit †
                     |    empty

<expr> ::=                <expr> or <expr>             /* Boolean expressions */
                     |    <expr> and <expr>        |          not <expr>

                     |    <expr> < <expr>          |      <expr> <= <expr>
                     |    <expr> > <expr>          |      <expr> >= <expr>
                     |    <expr> <> <expr>         |      <expr> = <expr>

                     |    <expr> ..‡ <expr>            /* range sequence */

                     |    <expr> + <expr>              /* algebraic expressions */
                     |    <expr> - <expr>
                     |    + <expr>                 |      - <expr>
                     |    <expr> * <expr>          |      <expr> / <expr>
                     |    <expr> ** <expr>
                     |    <expr> ^ <expr>
                     |    <expr> , <expr>
```

† **done** or **stop** can be used as synonyms for **quit**. In the Vax UNIX and Honeywell TSS versions of Maple, hitting the break/interrupt key twice in rapid succession will also exit from Maple.

```
                |     proc ( § <nameseq> § )                  /* procedure definition */
                            § local <nameseq> ; §
                            § options <nameseq> ; §
                            <statseq>
                            end

                |     <natural> . <natural>                   /* floating point numbers */
                |     <natural> .
                |     . <natural>

                |     <natural>                               /* unsigned integer */

                |     { <exprseq> }                           /* set */

                |     [ <exprseq> ]                           /* list */

                |     <name>                                  /* variable name */

                |     <expr> !                                /* factorial */

                |     ( <expr> )                              /* parenthesized expression */

                |     ' <expr> '                              /* unevaluated expression */

                |     <name> ( <exprseq> )                    /* function call */

                |     "                               /* previously computed expressions */
                |     ""        |       """

xprseq>         ::=      <exprlist>  |       empty

xprlist>        ::=      <exprlist> , <expr> |       <expr>

ame>  ::=       <string> |     <name> . ( <expr> )
                         |     <name> . <string>
                         |     <name> . <natural>
                         |     <name> [ <exprseq> ]

nameseq>        ::=     <nameseq> , <string>     |      <string>

elsepart>       ::=     fi       |      else <statseq> fi
                        |      elif <expr> then <statseq> <elsepart>
```

ctually, two *or more* consecutive periods are permitted.

```
<natural>        ::=      § <natural> §   <digit>

<digit>  ::=     0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<string> ::=     <letter>   § <alphanumeric> §
                  |         @   § <alphanumeric> §
                  |         ` <charstring> `

<alphanumeric>   ::=      § <alphanumeric> §   <letter>
                  |         § <alphanumeric> §   <digit>
                  |         § <alphanumeric> §   _

<letter>  ::=    /* Any lower-case or upper-case letter a-z or A-Z. */

<charstring>     ::=      <anychar>   § <charstring> §

<anychar>        ::=      /* Any character in the supported character set. */
```

7tr

5. ARRAYS AND TABLES

5.1. Overview

Two of the data types in Maple are *array* and *table*. Arrays are used similarly to those in other programming languages, while tables correspond roughly to the ones provided in Snobol or Icon.

In Maple, the type 'array' is a specialization of the type 'table'. An array is a table for which indices must be integer expression sequences lying within user-specified bounds.

As with other data types in Maple, tables are self-describing data objects, which may be created dynamically, passed as parameters, and so on. No declarations are needed; to make a name refer to a table, an assignment statement is used in which the right-hand side evaluates to a table object.

A table object consists of three parts:

- an indexing function
- an index bound (for arrays only)
- a collection of components

The indexing function allows a table to have a user-defined interface. A detailed discussion of indexing functions is given in section 5.5. If no special interface is to be used, the indexing function should have the value NULL. (This is the default.)

The only tables which have index bounds are arrays. The index bound is an expression sequence of integer ranges. The number of ranges is called the *dimension* of the array. Whenever a component of an array is referenced, the index is checked against the index bound. The i*th* range gives the bounds on the i*th* integer in the expression sequence used as the index.

The *op* function may be used to extract the operands of a table. The indexing function is available as the first operand of a table object. For arrays, the index bound is available as the second operand. The components are not available using the *op* function. The reason for this is that the collection of components is stored in an internal hash table and is not a user-level expression. (This is analogous to the statement sequence in a procedure not being available to the *op* function.)

Components of tables are referred to using brackets ('[' and ']') for indexing. If T evaluates to a table, then components of T are referenced using the syntax `T[<expression sequence>]`. For example, executing the following statements

```
T[1,2] := a;  T[2,0] := b;  V := T[1,2]+T[2,0]
```

causes V to be assigned the value a+b.

Tables may be created either (i) explicitly, by calling one of the builtin functions *array* or *table*, or (ii) implicitly, by making an assignment to an indexed expression of the form `A[<expression sequence>]` when A does not evaluate to a table. The creation of tables is described in section 5.2.

Expressions of type 'array' and of type 'table' are represented internally using the same data structure. The external representation is as a call to one of the functions *array* or *table* which would re-create the object. Specifically, the external representations are:

```
array(<indexing function>, <range sequence>, [<equation sequence>])
```

and

```
table(<indexing function>, [<equation sequence>]) ,
```

where each equation in the <equation sequence> is of the form

```
(index) = component_value .
```

The equation sequence enclosed in brackets is a representation of an internal hash table. The equations will appear in an apparently arbitrary order. The order in which they appear can not easily be controlled by the user.

5.2. Creating Tables

5.2.1. Explicit Table Creation

The function *array* is used to create an array explicitly. To explicitly create a table which is not an array, the function *table* is used. These functions take a number of optional parameters which specify information about the table to be created.

Probably the most common uses of these functions are illustrated by the following examples:

```
t := table();
a := array(1..n);
b := array(1..n, 1..m);
```

Here, 't' has been assigned a new table object, 'a' has been assigned a one-dimensional array with n components, and 'b' has been assigned an n by m, two-dimensional array.

The *table* function, in general, takes two parameters: an indexing function and an initialization list. The function *array* takes an indexing function, an initialization list, and an index bound. The index bound is passed as a number of integer ranges appearing adjacently in the parameter sequence. The indexing function, initialization list and, for arrays, index bound are all optional and may appear in any order in the parameter sequence.

When a table is created using one of these functions, the following sequence of events takes place:

(1) The parameters are sorted out.

(2) If no indexing function is supplied, NULL is taken as the default.

(3) If no initialization list is supplied, an empty list is taken as the default.

(4) If it is an array that is being created and no index bound has been supplied, the index bound to be used is deduced from the initialization list.

(5) If the initialization list is not empty, the initial values are inserted into the table.

(6) The table is returned.

The indexing function must be given as either a procedure or as a name. Not specifying an indexing function is the only way to obtain NULL.

The deduction of index bounds for arrays and the initialization of table values are done by two procedures from the Maple library. It is possible to change the actions performed by redefining these procedures within the Maple session. This possibility is discussed further in section 5.2.3. The remainder of this section describes the actions performed by the standard functions.

The initializations must be given either as a list of equations or as a list of values. To avoid ambiguity, with a list of values, none of the values may be an equation. With a list of equations, the left-hand sides are the indices (of the components to be initialized) and the right-hand sides are the values. A list of values may be given only for the creation of a table or a one-dimensional array. If a list of values is given, the indices used are consecutive integers starting at 1 or the lower bound on the indices, if one is given, for an array.

If no index bound is given for an array, then one is deduced from the list of initializations. This is done as follows. If the initialization list is empty, then the array is assumed to be zero-dimensional and the index bound is NULL (i.e. a sequence of zero integer ranges). If the initialization is given as a list of n values, then the array is taken to be one-dimensional and the index bound is the range 1..n. Finally, if the initializations are given as a list of equations, then each range in the index bound is made as restrictive as possible while still encompassing all the indices used in the equations.

Examples:

```
table( );                         yields  table([ ])
(The indexing function is NULL and no components have been initialized.)

table([a,b,c]);                   yields  table([(1)=a,(3)=c,(2)=b])
table([1=a0, cos(x)=a1]);         yields  table([(1)=a0,(cos(x))=a1])

array(0..3);                      yields  array(0..3,[ ])
array([x,y,z]);                   yields  array(1..3,[(1)=x,(2)=y,(3)=z])
array([b,c,d], 0..3);             yields  array(0..3,[(1)=c,(0)=b,(2)=d])
array([NULL=val1]);               yields  array([()=val1])
array([(2,2)=22, (1,7)=17]);       yields  array(1..2,2..7,[(2,2)=22,(1,7)=17])
array(sparse,[5=x,100=y]);         yields  array(sparse,5..100,[(5)=x,(100)=y])
```

5.2.2. Implicit Table Creation

A table is implicitly created if an assignment is made to an indexed expression of the form T[<expression sequence>] where T does not evaluate to a table. Implicit table creation is provided primarily as a convenience for interactive use.

If T does not evaluate to a table, then the assignment

```
T[eseq] := expr
```

is exactly equivalent to the following statement sequence which uses explicit table creation:

```
T := table(); T[eseq] := expr
```

This rule is applied recursively if necessary.

Examples:

If A is a table but A[1] has not been assigned, then

```
A[1][2,x] := y
```

is equivalent to

```
A[1] := table(); A[1][2,x] := y
```

If B does not evaluate to a table, then

```
B[i,k][j] := f(i,j)
```

is equivalent to

```
B := table(); B[i,k] := table(); B[i,k][j] := f(i,j)
```

5.2.3. User Interface for Table Creation

As stated earlier, when a table is being created, the deduction of an index bound (if it is an array) and the initialization of components are done by two procedures. These procedures are called `table/initbds` and `table/initvals`, respectively. By default, the procedures from the Maple library are used.

It is possible to change the actions performed by redefining these procedures within a Maple session. (It is *not* necessary to know how to do this to use tables effectively.) As an example, a default lower value of 0 may be desired for index bounds, rather than 1. Another example would be if the user wanted an initialization of the form

```
table([1..3 = 0, 4 = 1, 5..8 = 0]);
```

to yield

```
table([(1)=0,(2)=0,(3)=0,(4)=1,(5)=0,(6)=0,(7)=0,(8)=0]) .
```

If a table being created is an array and no index bound has been specified, then the procedure `table/initbds` is called. It is passed the initialization list and the value it returns is used as the array's index bound.

If the initialization list is not empty, the procedure `table/initvals` is used to install the initial values in the table being created. It is passed the new, empty table object and the initialization list from the call to *table* or *array*. The library version of `table/initvals` simply assigns the components of the table in a loop.

To provide a model, the library functions are given in Appendix B.

5.3. Table Components

5.3.1. Evaluating Components

The semanitcs of referencing a table's components are defined by its indexing function. With the default indexing function, NULL, the usual notion of subscripting is used. With other indexing functions, a procedure determines how indexing is done. This more complicated case is discussed in section 5.5. In this section, the default indexing semantics are described.

Suppose that T evaluates to a table with a NULL indexing function. When T[<expression sequence>] is evaluated, the value of the entry in the table is returned, if there is one. If there is not an entry with the <expression sequence> as its key, then the table reference "fails".

This is analogous to a FAIL return from a procedure. The value returned is an indexed object where the index is the <expression sequence>, evaluated, and the zero*th* operand is the name which directly evaluated to the table. If T is a table rather than a name which evaluates to a table, then the zero*th* operand is the table itself.

Examples:

```
t := table();                     yields  t := table([ ])
t[k] := ZZ;                       updates the table to  table([(k)=ZZ])

s := 't';
s[1] := XX;                       updates the table to  table([(1)=XX,(k)=ZZ])
t[k];                             yields  ZZ
s[1];                             yields  XX
t[2];                             yields  t[2]
j := 2;
t[j];                             yields  t[2]
s[2];                             yields  t[2]

p := proc(a) a[2] end;
p('s');                           yields  t[2]
p(s);                             yields  table([(1)=XX,(k)=ZZ])[2]
```

In the above examples, the name 's' evaluates to the name 't' which then evaluates to the table. That is why s[2] yields t[2] when it fails. With the first procedure call, p('s'), this is what happens when the table reference fails. In the second call, p(s), the name 's' gets fully evaluated and it is the table object that is passed. Then, when the table reference fails, that object is used as the zero*th* operand of the result.

Even if the last name in the evaluation chain evaluates to some other object before evaluating to the table, it is still used if a table reference fails.

Example:

```
a := b;                           yields  a := b
b := f(table( ));                 yields  b := f(table([ ]))
f := proc(t) print(hello); t end;
a[x];                             yields  b[x] after printing "hello"
```

An array is zero-dimensional if its index bound is the null expression sequence. A zero-dimensional array has only one component and the index for this component is NULL.

Example:

```
t := array( );                    yields  t := array([ ])
t[ ] := tval;                     updates t to  array([( )=tval]);
t[ ];                             yields  tval
```

5.3.2. Assigning and Unassigning Components

If T is a table and an expression of the form T[<expression sequence>] appears on the left-hand side of an assignment statement, then an entry is assigned in the table. If there was previously no entry in the table with the <expression sequence>, evaluated, as its key, then a new entry is made. If there already is an entry, then it is updated to reflect the new value.

In many cases it is desired to assign a value to a parameter of a procedure. Table components may be assigned this way, in the same way as names. To assign a table component, an indexed expression is passed. Consider the procedure

```
assignsqr := proc(a,b) a := b**2 end
```

So long as the first parameter is a valid left-hand side, the assignment will be made.

Examples:

```
t := table();
assignsqr(t[2], 4);          assigns the value 16 to t[2]
s := 's';                    unassigns s
assignsqr(s[3], 3);          assigns 9 to s[3], implicitly creating a table
```

If the component to be assigned already has a value, then it is necessary to use quotes or the *evaln* function to pass the indexed name.

Examples:

After executing the statements

```
t := table();
for i to 5 do assignsqr(t[i], i) od;
```

assigning new values to the components of 't' may be achieved by

```
for i to 5 do assignsqr(evaln(t[i]), 1/i) od;
```

When the subscript need not be evaluated, quotes may be used:

```
assignsqr('t[1]', x);
```

To make a name stand for itself in maple, a statement is executed to "unassign" it. Exactly the same thing is done with the components of a table — to remove an entry from a table, it is "unassigned". This may be done either by quoting the right-hand side or by using the *evaln* function.

Examples:

```
    a := array([x, y, z]);        yields  a := array(1..3,[(1)=x,(2)=y,(3)=z])
    a[1];                         yields  x
    a[1] := 'a[1]';
    a[1];                         yields  a[1];
    a;                            yields  array(1..3,[(2)=y,(3)=z])
    i := 3;
    a[i] := evaln(a[i]);
    a;                            yields  array(1..3,[(2)=y])
```

5.4. Tables as Objects

5.4.1. Copying Tables

In Maple, only objects of type 'table' may be altered after having been created. This is because it is only with tables that it is valid to make an assignment to a *part* of the object.

The fact that a table object may be altered after creation means that if two names evaluate to the same table, then an assignment to a component of one affects the other as well. To illustrate, if the following statements are executed:

```
    a := array([t,x,y,z]);
    b := a;
    a[1] := 9;
```

then b[1] will evaluate to 9, not 't'.

For this reason the *copy* function is provided (see section 5.6.3). It may be used to create a copy of a table upon which operations may be performed without altering the original.

For example, if a procedure makes assignments to components of a table passed as a parameter, then it may be necessary to pass a copy. Suppose that 'decomp' has been assigned a procedure that does an in-place LU matrix decomposition, and takes an array as its only parameter. If it is desired to find the LU decomposition of the matrix given by 'a' while retaining 'a' for further use, then the following statements may be used:

```
    b := copy(a);
    decomp(b);
```

5.4.2. Tables Local to a Procedure

A variable local to a procedure may be assigned a table, just as it may be assigned an object of any other type.

A table object which is created and assigned to a local variable may be returned as the value of the procedure or passed out through one of the parameters, in exactly the same way as any other expression.

Example:

```
# put the coefficients of a polynomial in a table
getcoeffs :=
proc(poly, var)
        local Cs, c, i;
        if not type(poly, polynom, var) then
                ERROR(`must have a polynomial`)
        fi;
        Cs := table();
        for i from ldegree(poly, var) to degree(poly, var) do
                c := coeff(poly, var, i);
                if c <> 0 then Cs[i] := c fi
        od;
        Cs
end;

Cs := table([this, that]);          yields  table([(1)=this,(2)=that])
getcoeffs(3*x**67 + y, x);          yields  table([(0)=y,(67)=3])
Cs;                                 yields  table([(1)=this,(2)=that])
```

5.4.3. Tables as Parameters

A table may be passed as a parameter into or out of a procedure. Components added to the table or removed from the table while the procedure is executing affect the globally visible table, since it is the same object.

If a table is passed as a parameter in the following way:

```
a := table();  p(a);
```

then it should be noted that the name 'a' is evaluated to the table object before the procedure is invoked. Therefore, if a reference to the table "fails" within in the procedure, then the resulting indexed expression will have the table object as its zero*th* operand.

This can be avoided by passing the name of the table (i.e. p('a')), thereby making it available to any component references which may fail. Note that this situation does not arise if all the components which are used have been assigned prior to the procedure call.

Passing an un-named table object as a parameter may lead to awkward results if components which do not have values are used. If the procedure makes a component reference that fails and assigns it to another component of the same table, then doing the assignment creates a self-referential data structure. (Just as doing `x := y; y:= 'x**2'` does.) This would lead to an infinite evaluation recursion the next time the component was evaluated.

5.4.4. Automatic Loading of Tables

It is possible to define large tables that get loaded only when a component is referenced. This is done in the same way that procedures can be made to be read in only when used.

To cause a table to be loaded automatically, it is assigned an unevaluated call to *readlib*. If a user wants T to be loaded when it is used, then he makes the assignment

```
T := 'readlib('T', filename)';
```

where 'filename' is the name of the file in which the table has been saved.

Suppose a user enters Maple and executes the following statements:

```
Linverse := table();
Linverse[1/s**n] := t**(n-1)/(n-1)!;
Linverse[1/(s**2 + a**2)] := sin(a*t)/a;
save `/u/jqpublic/laplace.m`;
quit
```

If in a subsequent Maple session the assignment

```
Linverse := 'readlib('Linverse', `/u/jqpublic/laplace.m`)';
```

has been made, then evaluating

```
Linverse[1/s**n]
```

causes *readlib* to be executed and the table is read in. The indexed expression then evaluates to

```
t**(n-1)/(n-1)!
```

5.5. Indexing functions

5.5.1. The Purpose of Indexing Functions

The semantics of indexing into a table are described by its indexing function. Using an indexing function, it is possible to do such things as efficiently store a symmetric matrix or count how often each element of a table is referenced. Because each table defines its own indexing method, generic programs can be written that do not need to know about special data representations. For example, the same function would be used to perform an operation on sparse matrices as for dense matrices.

The normal method of indexing, described in section 5.3, is used when the indexing function of a table is NULL. The semantics correspond roughly to those of common programming languages, with the added notion of "failing" if a component has not been assigned.

If the indexing function for a given table is not NULL, then all indexing into that table is done through a procedure. This procedure is invoked whenever an expression of the form `T[<expression sequence>]` is encountered and T evaluates to the table.

Three parameters are passed to the procedure:

(i) the object which is being indexed, T, (unevaluated)

(ii) a list containing the index, `<expression sequence>`, (evaluated)

(iii) a Boolean value which is *true* (*false*) if the expression is being evaluated as on a left-hand (right-hand) side of an assignment.

T is passed unevaluated so that a name will usually be available if a table reference "fails". The value returned by the procedure is used in the place of the indexed expression.

The indexing function may be the procedure itself, or a name. Certain names are known to the basic system as built-in indexing functions. If a name is given which is not one of these, a function call is made using `` `index/`.<name> ``. First the current session environment is searched for this name. If it is not found, the Maple system library is searched for the file `` ``.libname.`index/`.<name>.`.m` ``. If no such file exists, then `` `index/`.<name> `` is applied as an undefined function.

5.5.2. Indexing Functions Known to the Basic System

At present, three names are known to the basic Maple system as indexing functions. These are *symmetric*, *antisymmetric*, and *sparse*.

The indexing function *symmetric* is used for tables in which the value of a component is independent of the order of the expressions in the index. The most common application is for symmetric matrices. When a component of a table with this indexing function is referenced, the index expression sequence is re-ordered to give a unique key. (The sort is done using the same internal ordering as for sets.)

Examples:

```
A := array(1..10,1..10,symmetric);      yields  A := array(symmetric,1..10,1..10,[ ])
A[1,2];                                 yields  A[1,2]
A[2,1];                                 yields  A[1,2]
A[i,j] - A[j,i];                        yields  0
A[3,4] := x;                            yields  A[3,4] := x
A[4,3] := y;                            yields  A[3,4] := y
A;                                      yields  array(symmetric,1..10,1..10,[(3,4)=y])
T := table(symmetric);                  yields  T := table(symmetric,[ ])
T[function,continuous,odd] := f;        yields  T[odd,continuous,function] := f
```

The *antisymmetric* indexing function yields the result of *symmetric*, multiplied by −1 if all components of the index are different and an odd number of transpositions were necessary to re-order the index. If two or more components of the index are the same, *antisymmetric* returns 0.

Examples:

```
B := table(antisymmetric);       yields  B := table(antisymmetric,[ ])
B[i,j];                          yields  B[i,j]
B[j,i];                          yields  -B[i,j]
B[i,j,k] + B[i,k,j];             yields  0
B[i,k,k];                        yields  0
B[i,j] := v;                     yields  B[i,j] := v
B[j,i] := u;                     yields  ERROR: invalid name forming operation
```

The indexing function *sparse* is used with tables for which a component is assumed to have value 0 if it has not been assigned. Suppose T is a table with this indexing function. Evaluating T[<expression sequence>] on a right-hand side yields the component's value, if it has been assigned, or 0, if it hasn't. When T[<expression sequence>] is evaluated on a left-hand side, the indexing function always returns the indexed expression T[<expression sequence>] . (Returning 0 would make assigning components impossible.)

Examples:

```
U := array(1..100,sparse,[90=u1]);      yields  U := array(sparse,1..100,[(90)=u1])
V := array(1..100,sparse,[34=v1]);      yields  V := array(sparse,1..100,[(34)=v1])
s := 0;
for i to 100 do
        s := s + U[i] + V[i]
od;
s;                                      yields  u1 + v1
```

5.5.3. User-Defined Indexing Functions

A user may create his own indexing function by writing a procedure which returns the expression to be used, given the object being indexed, the index, and an indication of whether a left- or right-hand side is desired.

Suppose we wish to define a large tridiagonal matrix. To avoid storing the off-diagonal elements, the following procedure may be used as the indexing function:

```
t3 :=
proc(A, index) local dummy;
        op(1,index) - op(2,index);
        if not type(", integer) or abs(") <= 1 then
                subs(dummy = op(index), 'A[dummy]')
        else
                0
        fi
end
```

The array may be created by the statement

```
Tri := array(1..10000, 1..10000, t3)
```

or by the statements

```
`index/tridiagonal` := t3;
Tri := array(1..10000, 1..10000, tridiagonal)
```

In the first case, 'Tri' would have the procedure as its first operand. In the second, it would have the name 'tridiagonal'. Assume for the following discussion, that 'Tri' has been assigned by the second method.

To explain how this procedure works, suppose the statement

```
Tri[3,20] := rhs;
```

is executed. After the right-hand side has been evaluated, 'Tri' is evaluated and found to be an array. Next, the index is evaluated and found to be within bounds. The indexing function is then found to be 'tridiagonal' so the following procedure call is made:

```
`index/tridiagonal`('Tri', [3,20], true)
```

The third parameter indicates that the evaluation is being done for the left-hand side of an assignment. (In this case the procedure `index/tridiagonal` does not use the third parameter, but it is passed anyway.) The element referred to is found not to be on the tri-diagonal band so the *else* part is executed and the value 0 is returned.

Since it is impossible to make an assignment to 0, the assignment statement generates an error message. This is reasonable; it should not be possible to make assignments to the off-band entries of a tridiagonal matrix.

If the statement

```
Tri[99,100];
```

is executed, then the following events occur. As before, 'Tri' is evaluated to a table and the index is found to be within bounds. Then the procedure call

```
`index/tridiagonal`('Tri', op([99,100]), false)
```

is made. Since this component is found to be on the upper diagonal, the statement

```
subs(dummy = [99,100], 'Tri[dummy]')
```

is executed. This returns Tri[99,100], unevaluated, as the procedure value. Then, if Tri[99,100] has been assigned, its value is retrieved. Otherwise the table reference fails as usual.

It is important to avoid evaluating the expression T[99,100] accidentally inside the procedure, as this would cause an infinite recursion. This is the reason that *subs* was used to create the expression returned by `index/tridiagonal`.

As a second example, suppose we want to count the number of assignments made to components of various arrays and other tables. The counts will be kept in a table, 'Count_table', initialized by

```
Count_table := table(sparse);
```

If 'A' is one of the tables to be monitored and an assignment is made to A[1,2], then Count_table[A,[1,2]] will be incremented by one.

The procedure below may be used as the indexing function for the tables to be monitored:

```
`index/count` :=
proc(T, index, is_lhs)
        local dummy;
        if is_lhs then
                T;
                Count_table[ ", index] := Count_table[ ", index] + 1
                # " is used to evaluate the name and get the table.
        fi;
        subs(dummy = op(index), 'T[dummy]')
end
```

Then, the tables under investigation are created as follows

```
t1 := table(count);
aa := array(1..100, count);
```

and used normally.

For a third example, we consider the "Riemann tensor" from general relativity. For our purposes it may be considered to be an array with (0..3,0..3,0..3,0..3) as its index bound. This object would have 256 components if all were independent. However, the Riemann tensor has (among others) the following symmetry properties

```
R[i,j,k,l] = -R[j,i,k,l]
R[i,j,k,l] = -R[i,j,l,k]
R[i,j,k,l] =  R[k,l,i,j] .
```

These imply that at most 21 components are algebraically independent. The array could be created with the following procedure as its indexing function:

```
`index/riemann` :=
proc(A,ix)
        local i,j,k,l,dummy;
        option remember;
        i := op(1,ix); j := op(2,ix); k := op(3,ix); l := op(4,ix);

        if i = j or k = l then        0
        elif not order(i,j) then      -A[j,i,k,l]
        elif not order(k,l) then      -A[i,j,l,k]
        elif not order([i,j],[k,l]) then A[k,l,i,j]
        else subs(dummy = op(ix), 'A[dummy]')
        fi
end;
```

where 'order' is a boolean function defining an ordering on expressions, such as

```
        order := proc(a,b) evalb( a = op(1,{a,b}) ) end
```

(which uses the ordering defined by Maple's ordering of elements in a set). The procedure `index/riemann` is recursive, since evaluating the expressions −A[j,i,k,l], −A[i,j,l,k], and A[k,l,i,j] causes the indexing function to be called again.

6. PROCEDURES

6.1. Procedure Definitions

One instance of an expression in Maple is a procedure definition, which has the general form

```
proc ( <nameseq> ) local <nameseq>; options <nameseq>; <statseq> end
```

Such a procedure definition may be assigned to a <name> and it may then be invoked using the syntax

```
<name> ( <expseq> ) .
```

When a procedure is invoked the statements in <statseq> are executed sequentially (and some of the names have special semantics as described below). The *value* of a procedure invocation is the value of the last statement in <statseq> that is executed.

It is possible in Maple to define and invoke a procedure without ever assigning the procedure definition to an explicit <name>, as in the following example:

```
proc (x) x^2 end;
"(2);
```

where the value of the procedure invocation "(2) is 4. Another example of using a procedure definition without a name is when a simple function is to be *mapped* onto an expression, as in:

```
a := [ 1, 2, 3, 4, 5 ];
map( proc (x) x^2 end, a );
```

which causes each element of the list 'a' to be squared, yielding the new list [1,4,9,16,25].

The keywords 'proc' and 'end' may be viewed as brackets which specify that the <statseq> is to remain unevaluated when the procedure definition is evaluated as an expression. The simplest instance of a procedure definition involving no formal parameters, no local variables, and no options can be seen in the following definition of a procedure called max:

```
max := proc () if a>b then a else b fi end
```

Executing the statements

```
a := 25/7;  b := 525/149;  max();
```

yields 25/7 as the value of the procedure invocation max(). This procedure is making use of the names 'a' and 'b' as *global names*. In Maple, all names are global names unless otherwise specified. One instance of non-global names is the case of *formal parameters* which are specified within the parentheses immediately following the keyword 'proc'. A more useful definition of the above procedure max can be obtained by making the names 'a' and 'b' formal parameters:

```
max := proc (a,b) if a>b then a else b fi end
```

This procedure may now be invoked in the form max(expr1, expr2) where expr1 and expr2 are expressions. For example, max(25/7, 525/149) evaluates to 25/7. The names 'a' and 'b' are now local to the procedure, so that if these names have values external to the procedure the external values neither effect, nor are affected by, the invocation of the procedure.

6.2. Parameter Passing

The semantics of parameter passing are as follows. Suppose the procedure invocation is of the form

$$name\ (\ expr_1,\ expr_2,\ \ldots,\ expr_n\ )\ .$$

Firstly, *name* is evaluated and let us suppose for now that it evaluates to a procedure definition with formal parameters

$$parm_1,\ parm_2,\ \ldots,\ parm_n\ .$$

Next, the *actual parameters* $expr_1, \cdots, expr_n$ are evaluated in order from left to right. Then every occurrence of $parm_i$ in the <statseq> which makes up the body of the procedure is substituted by the value of the corresponding actual parameter $expr_i$. It is important to note that these parameters will not be evaluated again during execution of the procedure body. (The consequences of this fact are explained in section 6.4 below). In terms of traditional parameter passing mechanisms used by various computer languages, Maple's parameter passing could be termed 'call by evaluated name'. In other words, all actual parameters are first evaluated (as in 'call by value') but then a strict application of the substitution rule is applied to replace each formal parameter by its corresponding actual parameter (as in 'call by name').

It is possible for the number of actual parameters to be either greater than, or less than, the number of formal parameters specified. If there are too few actual parameters then a semantic error will occur if (and only if) the corresponding formal parameter is referenced during execution of the procedure body. The case where the number of actual parameters is greater than the number of specified formal parameters is, on the other hand, fully legitimate. Maple allows an alternate mechanism for referencing parameters within a procedure body; namely, the special array 'args'. The name 'args[i]' references the ith actual parameter. For example, the above procedure max could be defined without any specified formal parameters as follows:

```
max := proc () if args[1] > args[2] then args[1] else args[2] fi end
```

This procedure may now be invoked exactly as before with two actual parameters and the semantics are identical to the previous definition. The user will notice that, when displaying procedure definitions, the current version of Maple uses the 'args' names for specifying formal parameters even if the user specified formal parameter names. For example, if the input to Maple is

```
max := proc (a,b) if a>b then a else b fi end;
```

then the response from Maple is

```
max := proc (a,b)if args[2]<args[1] then args[1] else args[2] fi end
```

(where, as a minor point, note also that Maple chooses to represent inequalities using the '<' relation rather than the '>' relation). There is no restriction against having extra actual parameters appear in a procedure invocation; if they are never referenced they are simply ignored (but they will be evaluated).

In addition to the special array 'args' there is one other special name that Maple understands within a procedure body, namely 'nargs'. The value of the name 'nargs' is the number of actual parameters (i.e. the number of arguments) with which the procedure was invoked. As an example of the use of the name 'nargs', let us generalize our procedure max so that it will be defined to calculate the maximum of an arbitrary number of actual parameters. Consider the following procedure definition:

```
max := proc ()
  result := args[1];
  for i from 2 to nargs do
    if args[i] > result then result := args[i] fi
  od;
  result
end;
```

With this definition of max we can find the maximum of any number of arguments. Some examples are:

```
max( 25/7, 525/149 );          yields     25/7
max( 25/7, 525/149, 9/2 );     yields     9/2
max( 25/7 );                   yields     25/7
max();                         causes an error
```

where the latter case is an example of a procedure being called with too few actual parameters. If we wish to change our definition of max so that the procedure invocation max() with an empty parameter list will return the null value then we may check for a positive value of nargs in a selection statement as in the following definition of max.

```
max := proc ()
  if nargs > 0 then
    result := args[1];
    for i from 2 to nargs do
      if args[i] > result then result := args[i] fi
    od;
    result
  fi
end;
```

6.3. Local Variables and Options

The mechanism for introducing *local variables* into a Maple procedure is to use the 'local part' of a procedure definition. The 'local part' must appear immediately following the parentheses enclosing the formal parameters, and its syntax is

```
local <nameseq>;
```

The semantics are that the names appearing in <nameseq> are to be local to the procedure. In other words, this can be viewed as causing a syntactic renaming of every occurrence of the specified names within the procedure body. As an example, let us reconsider the latest definition of max appearing above. There are two global variables appearing in the procedure definition which we would almost certainly want to make local: result and i. This is effected by the following version of the procedure definition.

```
max := proc ()
  local result, i;
  if nargs > 0 then
    result := args[1];
    for i from 2 to nargs do
      if args[i] > result then result := args[i] fi
    od;
    result
  fi
end;
```

The user will notice that, when displaying procedure definitions, the current version of Maple uses a function syntax of the form loc(i) for the various local variables that have been specified. The function call loc(i) returns the value of the ith local variable. The use of the loc(i) function calls is a reflection of the internal implementation, **but the user is not able to refer to local variables in this way**.

There is a facility to specify *options* for a procedure by using the 'options part' of a procedure definition. The 'options part' must appear immediately after the 'local part' and its syntax is either of the following two forms:

```
option <nameseq>;
options <nameseq>;
```

The only name that is currently recognized as an option is the name 'remember'. The semantics of specifying 'option remember' or 'options remember' as the options part of a procedure definition are as follows. After executing the procedure and obtaining the value of a particular procedure invocation, the Maple system makes an entry in a table called the *partial computation table* which associates the result with that particular procedure invocation. If there is ever another invocation of this procedure with actual parameters that have the same values then the Maple system will immediately retrieve the result from the partial computation table. In this way, it is possible to avoid redundant executions of procedures that may be very costly. (See also the remember function in section 8).

6.4. Assigning Values to Parameters

Let us now consider an example of a procedure where we may wish to return a value into one of the actual parameters. Recall that the integer quotient q and the integer remainder r of two integers a and b must satisfy the 'Euclidean division property'

```
a = b*q + r
```

where either r = 0 or abs(r) < abs(b). This property does not uniquely define the integers q and r, but let us impose uniqueness by choosing

```
q  =  trunc(a/b)
```

using the built-in Maple function trunc. The remainder r is then uniquely specified by the above Euclidean division property. (Note: This choice of q and r can be characterized by the condition that r will always have the same sign as a). The following definition of the procedure 'rem' returns as its value the remainder after division of the first parameter by the second parameter, and it also returns the quotient as the value of the third parameter (if present).

```
rem := proc (a,b,q)
  local quot;
  quot := trunc(a/b);
  if nargs > 2 then q := quot fi;
  a  -  quot * b
end;
```

The procedure rem as defined here may be invoked with either two or three parameters. In either case the value of the procedure invocation will be the remainder of the first two parameters. The quotient will be returned as the value of the third parameter if it appears. At this point recall that the semantics of parameter passing specify that the actual parameters are evaluated and then substituted for the formal parameters. Therefore, an error will result if an actual parameter which is to receive a value does not evaluate to a valid name. It follows that when a name is being passed into a procedure for such a purpose it should usually be explicitly quoted (to avoid having it evaluated to some value that it may have had previously). The following statements will serve to illustrate.

```
rem( 5, 2 );              yields    1
rem( 5, 2, 'q' );         yields    1
q;                        yields    2

rem( -8, 3, 'q' );        yields    -2
q;                        yields    -2
rem( 8, -3 );             yields    2
rem( 8, 3, q );           yields    System error (in evalname)
```

The latter error message arises because the actual parameter q has the value −2 from a previous statement, and therefore the value −2 is substituted for the formal parameter q in the procedure definition yielding an invalid assignment statement. The solution to this problem is to change the actual parameter from q to 'q'.

When values are assigned to parameters within a procedure, a restriction which must be understood is that parameters are evaluated only once. Basically this means that formal parameter names cannot be freely used like local variables within a procedure body, in the sense that once an assignment to a parameter has been made that parameter should not be referred to again. The only legitimate purpose for assigning to a parameter is so that on return from the procedure the corresponding actual parameter has been assigned a value. As an illustration of this restriction, consider a procedure get_factors which takes an expression expr and, viewing it as a product of factors, determines the number n of factors and assigns the various factors to the names f.i for i = 1 ,..., n. Here is one attempt at writing a procedure for this purpose.

```
get_factors := proc (expr,f,n)
  local i;
  if type(expr, `*`) then
    n := nops(expr);
    for i to n do
      f.i := op(i,expr)
    od
  else
    n := 1;
    f.1 := expr
  fi
end;
```

If this procedure is invoked in the form

```
get_factors(x*y, 'f', 'number');
```

the result is 'ERROR: unable to execute for statement'. What has happened is that the third actual parameter is a name (as it must be because it is to be assigned a value within the procedure) and when execution reaches the point of executing the for-statement, the limit n in the for-statement is the name 'number' that was passed in. The point is that the formal parameter n is evaluated only once upon invocation of the procedure and it will not be re-evaluated. A general solution to this type of problem is to use local variables where necessary, and to view the assignment to a parameter as an operation that takes place just before returning from the procedure. For our example, the following procedure definition follows this point of view.

```
get_factors := proc (expr,f,n)
  local i, nfactors;
  if type(expr, `*`) then
    nfactors := nops(expr);
    for i to nfactors do
      f.i := op(i,expr)
    od
  else
    nfactors := 1;
    f.1 := expr
  fi;
  n := nfactors
end;
```

Another solution to the problem in this example is to change the limit in the for-statement to the operator ", which will yield the desired value. This leads to the following procedure definition.

```
get_factors := proc (expr,f,n)
  local i;
  if type(expr, `*`) then
    n := nops(expr);
    for i to " do
      f.i := op(i,expr)
    od
  else
    n := 1;
    f.1 := expr
  fi
end;
```

6.5. Error Returns and Special Returns

The most common return from a procedure invocation occurs when execution 'falls through' the end of the <statseq> which makes up the procedure body, and the value of the procedure invocation is the value of the last statement executed. There are three other types of returns from procedures.

An *error return* occurs when the special function call

```
ERROR( <expr seq> )
```

is evaluated. This function call causes an immediate exit to the top level of the Maple system and the following error header

```
ERROR:
```

followed by the evaluated sequence of expressions which are parameters to the function call.

An *explicit return* occurs when the special function call

```
RETURN( <expseq> )
```

is evaluated. This function call causes an immediate return from the procedure and the value of the procedure invocation is the value of the <expseq> given as the argument in the call to RETURN. In the most common usage <expseq> will be a single <expression> but a more general <expseq> (including the null expression sequence) is valid. It is an error if a call to the function RETURN occurs at a point which is not within a procedure definition.

A *fail return* occurs when the special name

```
FAIL
```

is evaluated. The effect of evaluating this name is to cause an immediate return from the procedure. The value of the procedure invocation is the procedure invocation itself, as an unevaluated expression. It is an error if the name FAIL occurs at a point which is not within a procedure definition. The effect of FAIL can also be achieved by the construct

RETURN('*name*(args)')

where *name* is the name by which the procedure was invoked.

As an example of a procedure which includes an error return and a fail return, consider the function 'max' which is supplied in the Maple library. The latest definition that we developed in section 6.3 for the function 'max' has a property which makes it unacceptable as a library function. Namely, if a user calls this function with an argument that does not evaluate to a constant, such as in max(x, y) where x and y have not been assigned any values, then the result is an error message from the Maple system: 'ERROR: cannot evaluate boolean'. This error results from attempting to execute an if statement of the form

```
if y>x then . . .
```

where x and y are indeterminates. Since this call to the function 'max' may have occurred from a procedure nested several levels, the resulting error message will not be very informative to the user. In order to improve this situation, type-checking should be done within the 'max' function and appropriate action should be taken if an invalid argument is encountered. The following code from the Maple library shows how this can be done using the FAIL return facility, so that the result of the function call max(x, y) will be the unevaluated function max(x, y). This code also shows the use of the ERROR return facility to return an error message if the function is called with no arguments.

```
max := proc ()
  local i, M, p;
  if nargs = 0 then
    ERROR(`function max called with no parameters`)
  else
    M := args[1];
    for i to nargs do
      p := args[i];
      if not type( p, rational ) and not type( p, float ) then FAIL fi;
      if M < p then M := p fi
    od;
    M
  fi
end;
```

6.6. Boolean Procedures

It was noted in section 3 that the names 'true' and 'false' may be freely manipulated as names even though these names have a special significance when they arise in a Boolean context. It follows that Boolean procedures may be written like any other procedures. As an example of a Boolean procedure, consider the following definition of a function called 'member' which tests for list membership.

```
member := proc (element, l)
  local i;
  false;  for i to nops(l) while not " do
                      evalb( element = op(i, l) )  od;
  "
end;
```

Some examples invoking this procedure follow.

```
member( x*y, [1/2, x*y, x, y] );          yields    true
member( x, [1/2, x*y] );                  yields    false
member( x, [ ] );                         yields    false
```

A few points about this procedure should be noted. Firstly, note that the equation

```
element = op(i, l)
```

is to be evaluated as a Boolean expression and therefore it is necessary to apply the function 'evalb' to it. Otherwise, this expression would be treated as an algebraic equation. Secondly, note the use of the nullary operator " in the while-part of the loop to refer to the 'latest expression'. Alternatively, this could be coded with the use of another local variable but in this case it seems preferable to use the " operator. Finally, it should be noted that the final " appearing in this procedure would be redundant in some contexts but is necessary here. If it were left out then the value of the procedure invocation would be the value of the last statement executed, which would be the value of the for-loop. This value will be null in the case where 'l' is the empty list, but the correct value to return in such a case is 'false' rather than the null value. However, the value of " is never updated by a null value and this fact is exploited in the above procedure definition.

6.7. Reading and Saving Procedures

It is usually convenient to use a text editor to develop a procedure definition and to write it into a file. The file can then be read into a Maple session. For example, the max procedure might be written into a file named /u/gahill/max . In a Maple session the statement

```
read `/u/gahill/max`;
```

will read in the procedure definition. Since this procedure is in 'user format' Maple will echo the statements as they are read in. Once the procedure is debugged it is desirable to save it in 'Maple internal format' so that whenever it is read into a Maple session the reading is very fast (and no time is spent displaying the statements to the user). To accomplish this one must use a file with a name ending in the characters '.m' . Within Maple the 'user format' file is read in and then Maple's save statement is used to save the file in 'Maple internal format'. For example, suppose that we have saved our procedure definition in a file named /u/gahill/max. If we then enter the Maple system and execute the statements

```
read `/u/gahill/max`;
save `/u/gahill/max.m`;
```

we will have saved the internal representation of the procedure in the second file. This file may be read into a Maple session at any time in the future by executing the statement

```
read `/u/gahill/max.m`;
```

which will update the current Maple environment with the contents of the specified file. (It is often convenient to place the 'save' statement at the end of the 'user format' file so that simply reading in the file will cause it to be saved in 'Maple internal format'). The user will quickly discover the time-saving advantages of saving procedure definitions in 'Maple internal format'.

A special case of reading procedure definitions in 'Maple internal format' can be accomplished using the built-in function *readlib*. Specifically, the function invocation readlib(pname) will cause the the following **read** statement to be executed:

```
read `` . libname . pname . `.m`
```

where 'libname' is a global name in Maple which is initialized to the pathname of the standard Maple system library on the host system. For example, on the UNIX system the value of 'libname' is `/u/maple/lib/`. (The value of 'libname' on any host system can be determined by entering Maple and simply displaying its value). The complete pathname being specified in the above **read** statement is a concatenation of the values of 'libname', 'pname', and the suffix `.m`, which could alternatively be specified by

```
cat( libname, pname, `.m` ) .
```

In order to specify this concatenation using only Maple's concatenation operator '.' it is necessary to concatenate these values to the null string ``, because the left operand of Maple's concatenation operator is not fully evaluated but is simply evaluated as a name. (See section 3.2.2.)

The *readlib* function is more general than this. If it is called with two arguments then the second argument is the complete pathname of the file to be read, and the first argument 'pname' is the procedure name which is to be defined by this action. Thus the following two function calls are equivalent:

```
readlib( 'f' )
readlib( 'f', `` . libname . `f.m` )
```

but if the procedure definition for 'f' is not in the standard Maple system library then the second argument is required to specify the correct file. (Even more generally, the *readlib* function can be called with several arguments in which case all arguments after the first are taken to be complete pathnames of files to be read, and the first argument is a procedure name which is to be defined by this action). The definition of the *readlib* function involves more than just the execution of one or more **read** statements. This function will also check to ensure that after the files have been read, 'pname' has been assigned a value and this value is returned as the value of the *readlib* function. In other words, the *readlib* function is to be used when the purpose of the **read** is to define a procedure named 'pname' (and some other names may or may not be defined at the same time).

The most common application of the *readlib* function is to cause automatic loading of files. For this purpose, the value of 'pname' is initially defined to be an unevaluated *readlib* function, as in one of the following assignments:

```
pname := 'readlib( 'pname' )';
pname := 'readlib( 'pname', filename )';
```

where the single quotes around the argument 'pname' are required to avoid a recursive evaluation. Then if there is subsequently a procedure invocation pname(...), the evaluation of 'pname' will cause the *readlib* function to be executed, thus reading in the file which defines 'pname' as a procedure, and the procedure invocation will then proceed just as if 'pname' had been a built-in function in Maple. Indeed, this method is precisely how the names of Maple's system-defined library functions are initially defined so that the appropriate files will be automatically loaded when needed. (For example, enter Maple and display op('gcd') to see what the name 'gcd' is defined to be and readlib('gcd') will be the response).

For completeness, the following is a definition in Maple code of Maple's built-in function *readlib*.

```
readlib := proc ( pname )
  local i, errmsg;
  errmsg := `wrong number (or type) of parameters`;
  if nargs=0 or not type(pname, name) then
    print( `In function readlib;` );  ERROR( errmsg )
  fi;
  pname := 0;
  if nargs=1 then
    read cat( libname, pname, `.m` )
  else
    for i from 2 to nargs do
      if not type( args[i], name ) then
        print( `In function readlib;` );  ERROR( errmsg )
      else
        read args[i]
      fi
    od
  fi;
  if op( pname ) = 0 then
     print( cat( pname, `:` ));
     ERROR(`ineffective readlib`)
  fi;
  pname;
  "
end;
```

7. INTERNAL REPRESENTATION AND MANIPULATION

7.1. Internal Organization

Maple appears to the user as an interactive "calculator". This mode is achieved by immediately executing any statement which is typed at the user level. It is in this context where we can define Maple as a *parser-driven* program. The parser is effectively the main program; its task is to read input, parse statements and call the statement evaluator each time a statement is input.

The parser accepts the Maple language which has been kept simple enough to have the LALR(1) property. The parser, being the main program, retains control throughout the session. For each production which is successfully reduced, it creates the appropriate data structure. Additionally the reduction of the nonterminal <stat> produces a call to the statement evaluator, the main Maple evaluator. Maple will read an infinite number of statements; its normal conclusion is achieved by the evaluation (not the parsing) of the <quit> statement. Thus it is possible to write a statement like:

```
if <condition> then quit fi;
```

which will terminate execution conditionally.

The initialization phase is normally called before the parser. In some sense we may say that both initialization and parser are at the topmost level of control. This is particularly true for some parsers, like yacc, which provide a "canned" main program whose only task is to call sequentially the initialization, parser, and possibly a finalization routine.

The internal functions in Maple can be divided into four distinct groups.

(1) Evaluators. The evaluators are the main functions responsible for evaluation. There are five types of evaluations: statements (done by evalstat); algebraic expressions (eval); boolean expressions (evalbool); name forming (evalname), and floating point arithmetic (evalf). Although the parser calls only evalstat, thereafter there are many interactions between the evaluators. For example, the statement

```
if a>0 then b.i := 3.14/a fi;
```

is first analyzed by evalstat which calls evalbool to resolve the if-condition. Once this is done, say with a true result, evalstat is invoked again to do the assignment, for which evalname has to be invoked with the left-hand-side and eval with the right-hand expression. Finally evalf will be called to evaluate the result. Most of the time the user will not directly invoke any of the evaluators; these are invoked automatically as needed. In some circumstances, when a different type of evaluation is needed, the user can directly call evalf, evalbool (evalb for the user), and evalname (evaln).

(2) Algebraic functions. These are functions which are directly identified with a function available at the user level, and are commonly called "basic". Some examples clarify this immediately: taking derivatives (diff), picking parts of an

expression (op), dividing polynomials (divide), finding coefficients of polynomials (coeff), series computation (taylor), mapping a function (map), substitution of expressions (subs, subsop), expansion of expressions (expand), finding indeterminates (indets), etc. Some functions in this group may migrate to the Maple level (Maple library) and vice versa due to tradeoffs between size and efficiency.

(3) Algebraic service functions. These functions are algebraic in nature, but serve as subordinates of the functions in the above group. Most frequently these functions cannot be explicitly called by the user. Examples of functions in this group are: the arithmetic (integer, rational, and float) packages (const, consti) the basic simplifier (simpl), printing (print), the series package (polyn), the set-operations package (sets), retrieval of library functions (retrieve), etc.

(4) General service functions. Functions in this group are at the lowest hierarchical level; i.e., they may be called by any other function in the system. Their purpose is general, and not necessarily tied to symbolic computation. Some examples are: storage allocation and garbage collection (storman), table manipulation (hash,pc), internal input/output (put), non-local returns, and various error handlers.

The flow of control within the basic system is not bound to remain at this level. In many cases, where appropriate, a decision is made to call functions written in Maple and residing in the library. For example, most uses of the function expand(...) will be handled by the basic system; however, if an expansion of a sum to a power greater than 4 is required, the internal expand will call the external (Maple library) function `expa/large` to resolve it. Functions such as diff, evalf, taylor, and type make extensive use of this feature. (For example, the basic function diff does not know how to differentiate any function; all its knowledge resides in the Maple library at pathnames `diff/<function name>`). This is a fundamental feature in the design of Maple as it permits flexibility (changing the library), personal tailoring (defining your own handling functions), readability (the source is in Maple code and available to all users), and allows the system to remain small by unloading unnecessary functions from the basic system.

7.2. Internal Representation of Data Types

The parser and some basic internal functions are responsible for building all of the data structures used internally by Maple. All of the internal data structures have the same general format:

Header	data 1	data 2	...	data n

The header field encodes the length (n+1) of the structure, the type, one bit to indicate simplification status, and two bits to indicate garbage collection status. The data items are normally pointers to similar data structures; the few exceptions to this rule are the terminal symbols.

Every data structure is created with its own length, and this length will not change during its entire existence. Furthermore, data structures should not be changed during execution since it is not predictable how many other data structures are pointing to a given structure. The normal procedure to modify structures is to create a copy and modify the copy, hence returning a new data structure. The only safe modifications are those done by the basic simplifier which produces the same value, albeit simpler. It is the task of the garbage collector to identify unused structures.

In the following figures we will describe the individual structures and the constraints on their data items. We will use the symbolic names of the structures since the actual numerical values used internally are of little interest. The symbol ↑<xxx> will indicate a pointer to a structure of type xxx. In particular we will use, whenever possible, the same notation as in the formal syntax (section 4.2).

Logical and

AND	↑<expr>	↑<expr>

Array

ARRAY	↑<indexing fcn>	↑<expr seq>	↑<components>

The <expr seq> is a sequence of integer ranges which are the bounds for the array. The <components> operand is not accessible through the *op* function.

Assignment statement

ASSIGN	↑<name>	↑<expr>

The <name> entry should evaluate to a valid name, which is one of the following data structures: NAME, CATENATE, LOCAL, or PARAM.

Break statement

BREAK

Concatenation of a name

CATENATE	↑<name>	↑<expr>

The <name> entry is treated as in ASSIGN. The <expr> entry must evaluate to a nonnegative integer or to a name to be successful. There are two exceptions: if <expr> is an EXPRSEQ the entry is taken to be an array reference (the content of the EXPRSEQ being the indices), and if <expr> is a RANGE the entry is a generator of an EXPRSEQ (e.g. a.(1..2) generates a1,a2).

Equation or test for equality

EQUATION	↑<expr>	↑<expr>

This structure, together with all of the relational operators, has a double interpretation: as an equation and as a comparison.

Expression sequence

EXPRSEQ	↑<expr>	↑<expr>	...

An EXPRSEQ may be of length 1 (no entries); this empty structure is called NULL.

Floating point number

FLOAT	↑<integer>	↑<integer>

The floating point number is interpreted as the first integer times 10 powered to the second.

For-while loop statement

FOR	↑<name>	↑<from>	↑<by>	↑<to>	↑<while-cond>	↑<statseq>

The entries for <from>, <by>, <to>, and <while> are general expressions which are filled with their default values, if necessary, by the parser. The <name> entry follows the same rules as in ASSIGN except that a NULL value indicates its absence. A NULL value in the <to> expression indicates that there is no upper limit on the loop.

Fortran function

FORTRAN	?

Not implemented for production yet.

Function call

FUNCTION	↑<name>	↑<exprseq>

This structure represents a function invocation (as distinct from a procedure definition which uses the PROC data structure). The <name> entry follows the same rules as in ASSIGN, or it may be a PROC definition. (The parser will not generate this structure with a PROC definition for the <name> entry, but this may happen internally). The <exprseq> contains the list of parameters.

If statement

IF	↑<if-condition>	↑<statseq>	↑<statseq>

The parser generates a NULL third entry for the if-then-fi statement, and generates an IF entry for the if-then-elif… statement.

Not equal or test for inequality

INEQUAT	↑<expr>	↑<expr>

Same comments as for EQUATION.

Negative integer

INTNEG	integer	integer	…

Integers are represented in base BASE (BASE=10000 for 32-bit machines and BASE=100000 for 36-bit machines). Each entry contains one "digit". A normalized integer contains no additional zeros. The integers are represented in reverse order; i.e., the first entry is the lowest order "digit", the last is the highest order "digit". BASE is the largest power of 10 such that BASE^2 can be represented in the host-machine integer arithmetic.

Positive integer

INTPOS	integer	integer	…

Similar to INTNEG.

Less or equal relation

LESSEQ	↑<expr>	↑<expr>

Similar to EQUATION. The parser also translates a "greater or equal" into a structure of this type, interchanging the order of its arguments.

Less than relation

LESSTHAN	↑<expr>	↑<expr>

Similar to EQUATION. The parser also translates "greater than" into a structure of this type, interchanging the order of its arguments.

List

LIST	↑<exprseq>

Occurrence of a local variable

LOCAL	integer

This entry indicates the usage of the <integer>th local variable. This structure is only generated by the simplifier when it processes a function definition. LOCAL entries cannot exist outside functions.

Identifier

NAME	↑<assigned-expr>	string	string	...

The first entry contains a pointer to the assigned value (if this identifier has been assigned a value) or 0. The string entries contain the name of the variable.

Logical not

NOT	↑<expr>

Logical or

OR	↑<expr>	↑<expr>

Occurrence of a parameter variable

PARAM	integer

Similar to LOCAL, but using the parameters of the function.

Rational number

RATIONAL	↑<INTPOS or INTNEG>	↑<INTPOS>

The second integer is always positive and different from 0 or 1. The two integers are relatively prime.

Series

SERIES	↑<expr>	↑<expr$_1$>	integer$_1$	...	...

The first expression is the "taylor" variable of the series, the variable used to do the series expansion. The remaining entries have to be interpreted as pairs of coefficient and exponent. The exponents are integers (not pointers to integers) and appear in increasing order. A coefficient O(1) (function call to the function "O" with parameter 1) is interpreted specially by Maple as an "order" term.

Power

POWER	↑<expr>	↑<expr>

If the second entry is a rational constant, this structure is changed to a PROD structure by the simplifier.

Procedure definition

PROC	↑<nameseq$_1$>	↑<nameseq$_2$>	↑<nameseq$_3$>	↑<statseq>

The first <nameseq> is an EXPRSEQ of the names specified for the formal parameters. The second corresponds to an EXPRSEQ of the names specified for the local variables and the third to the options specified. The <statseq> points to the body of the function.

Product/quotient/power

PROD	↑<expr$_1$>	↑<expon$_1$>	...	...

This structure should be interpreted as pairs of expressions and their (rational) exponents. Rational or integer expressions to an integer power are expanded. If there is a rational constant in the product, this constant will be moved to the first entry by the simplifier.

Range

RANGE	↑<expr$_1$>	↑<expr$_2$>

Read statement

READ	↑<expr>

The expression should evaluate to a name (string).

Save statement

SAVE	↑<expr>

The expression should evaluate to a name (string).

Set

SET	↑<exprseq>

The entries in the <exprseq> are sorted in increasing address order. This is an arbitrary order, but is necessary for sets. (Any other arbitrary, but consistent, order could serve.)

Statement sequence

STATSEQ	↑<stat$_1$>	↑<stat$_2$>	...

End execution

STOP

Sum of several terms

SUM	↑<expr$_1$>	↑<factor$_1$>	...	...

This structure should be interpreted as pairs of expressions and their (rational) factors. The simplifier lifts as many constant factors as possible from each expression and places them in the <factor> entries. A rational constant is multiplied by its factor and represented with factor 1.

Table

TABLE	↑<indexing fcn>		↑<components>

The <components> are not accessible through the use of the *op* function.

Unevaluated expression

UNEVAL	↑<expr>

7.3. Portability of the Maple System

One of the design goals of Maple is to be portable. The level of portability that we envision is one for which the scope of machines includes personal computers as well as present-day time-sharing systems. It was a very early decision that the language to be chosen should belong to the BCPL family. The reasons behind this decision are: efficiency, suitability, and availability. On the other hand, no single language in the BCPL family is sufficiently widely available to satisfy our needs. In view of this, we decided to write our system in a language which closely resembles B and C. This language is processed by the Margay macro-processor into either B or C, and in the near future into Port and WSL (which are two systems implementation languages developed at the University of Waterloo). Margay is a straightforward macro-processor which resembles closely, although is more powerful than, C's macro-processing. The most important difference is that Margay is written in its own macros and hence is portable across several systems.

The level of portability for the Maple user should be total. That is to say, a user should not be able to recognize in which hardware he is running. This is an easy consequence of the fact that there is a single source for Maple; the macro processing is done only before compilation and the intermediate code is never kept. It is important to realize that the entire basic system is only about 4500 lines of code. With such a small system, we can afford minor changes in the code to improve portability across systems. In many instances we add redundant information to be used by Margay, which may be ignored by some systems and used by others.

The Margay macros which aid in portability can be classified in various groups:

- name changes; e.g. concat(..) in B is equivalent to strcat(..) in C.
- declaration information; Margay recognizes EXF (external function definitions), FUN (function definition), PAR (definition of parameters), LOC (definition of local variables), EXT (definition of external variables), and three types: ALGEB (algebraic), LONGINT (multiple precision integers), and INT.
- system constants; EOF (end-of-file), TRUE, FALSE, MAXINT, MAXADDR, QUOTE, CHNL (new line character), etc.
- casting: I(...), forcing an expression to be of type integer.
- input/output: The input and output is one of the most delicate aspects of portability. The Maple system requires a very simple type of sequential input/output. Maple knows of only one sequential input stream (possibly stacked) and one sequential output stream (unique). The input and output are done either in words or in characters.

The macros used for input are:

- Ropen("filename") Opens a file for input, stacks the present input file, and returns false if it failed to open the file.
- Readch() Reads one character from the input stream.
- Readws(vect,nws) Reads nws binary words into vect. Returns the number of words read; 0 if EOF.
- Rclose() Closes file and unstacks previous file for input.

The output macros are:

- Wopen("filename") Opens a file for output (there is only one at a given time), and returns false if it failed.
- printf("format",...) Outputs characters.
- Writews(vect,nws) Outputs nws binary words from the vector vect.
- Wclose() Closes the output file.

7.4. Searching Tables in Maple

Maple handles all table searching in a uniform way. All of the searching is done by an algorithm which is a slight modification of direct-chaining hashing. Although it is not obvious, the internal tables play a crucial role; they are used for: locating variable names (nametab); keeping track of simplified expressions (simpltab); keeping track of partial computations (pctable); mapping expression trees into sequential files for internal input/output (puttab); and for storing arrays and tables. It is immediately obvious that the searching in these tables has to be fast enough to guarantee overall efficiency.

The algorithm used for these tables can be understood as an implementation of direct-chaining where instead of storing a linked list for each table entry, we store a variable-length array. This requires a versatile and efficient storage manager, but without one symbolic computation would not be feasible.

The two data structures used to implement tables are:

Table entry

HASHTAB	↑<HASH>	↑<HASH>	...	↑<HASH>

Each entry points to a HASH entry or it is 0 if no entry was created. The size of HASHTAB is constant for the implementation. For best efficiency, the number of entries should be prime.

Hash-chain entry

HASH	key	value	...

Each entry in the table consists of a consecutive pair, the first one being the hashing key and the second the stored value. A key cannot have the value 0 as this is the indicator for the end of a chain. For efficiency reasons, the HASH entries are incremented by 5 entries at a time and consequently some entries may not be filled. Keys may be any integer or pointer which is representable in one word. In many cases the key is itself a hashing value (two step hashing).

7.4.1. The Simplification Table

All simplified expressions and subexpressions are stored in the simplification table. The main purpose of this table is to ensure that expressions appear internally only once. Every expression which is entered to Maple or which is internally generated is checked against this table, and if found, the new expression is discarded and the old one is used. This task is done by the simplifier which recursively simplifies (applies all the basic simplification rules) and checks against the table.

The task of checking for equivalent expressions within thousands of subexpressions would not be possible if it was not done with the aid of a "hashing" concept. Every expression is entered in the simplification table using its ***signature*** as a key. The signature of an expression is a hashing function itself, with one very important attribute: it is order independent. For example, the signatures of the expressions a+b+c and c+a+b are identical; the signatures of a^b and b^a are also identical. Searching for an expression in the simplification table is done by:

- Simplifying recursively all of its components;
- Applying the basic simplification rules.
- Computing its signature and searching this signature in the table. If the signature is found then we perform a full comparison (taking into account that additions and products are commutative, etc.) to verify that it is the same expression. If the expression is found, the one in the table is used and the searched one is discarded.

The number of times that we have to do a full comparison on expressions is minimal; it is only when we have a "collision" of signatures. Some experiments have indicated that signatures coincide once every 50000 comparisons for 32-bit signatures. (Notice that the signatures are still far from uniform random numbers). The resulting expected time spent doing full comparisons is absolutely negligible. Of course, if the signatures disagree then the expressions cannot be equal at the basic level of simplification.

7.4.2. The Partial-Computation Table

The partial-computation table is responsible for handling the option remember in function definitions in its explicit and implicit forms. Basically, the table stores function calls as keys and their results as values. Since both these objects are data structures already created, the only cost (in terms of storage) to place them in the table is a pair of entries (pointers). Searching these hashing tables is extremely efficient and even for simple functions it is orders of magnitude faster than the actual computation of the function.

The change in efficiency due to the use of the remembering facility may be dramatic. For example, the Fibonacci numbers computed with

```
f := proc(n)
        if n<2 then n else f(n-1)+f(n-2) fi end;
```

take exponential time to compute, while

```
f := proc(n) option remember;
        if n<2 then n else f(n-1)+f(n-2) fi end;
```

requires linear time.

Besides the facility provided to users, the internal system uses the partial-computation table for diff, taylor, expand, and evalf. The internal handling of expand is straightforward. There are some exceptions with the others, namely:

- diff will store not only its result but also its inverse; in other words, if you integrate the result of a differentiation the result will be "table-looked up" rather than computed. In this sense, integration "learns" from differentiation.
- taylor and evalf need to store some additional, environment, information (Degree for taylor and Digits for evalf). Consequently the entries in these cases are extended with the precision information. If a result is requested with less precision than what it is stored in the table, it is retrieved anyway and "rounded". If a result is produced with more precision than what it is stored, it is replaced in the table.
- evalf only remembers function calls (this includes constants); it does not remember the results of arithmetic operations.

Both the simplification table and the partial-computation table are cleared of all unreferenced entries at garbage collection time.

7.4.3. Arrays

Arrays and tables are implemented with internal tables. In this case the address of the simplified EXPRSEQ of indices is used as a key for the searching. (Note that since simplified expressions appear only once, we can use their addresses as keys.) Arrays and tables are treated very similarly at the internal level. This implementation permits efficient use of sparse arrays of any kind without overhead.

7.5. Style Recommendations for Library Contributions

In this section we include several recommendations (or a checklist) which should be useful in preparing Maple software intended to be part of the library. The main motivation for this document is to provide uniformity and ease in porting and maintaining the library. We expect that contributors will find the recommendations sound and that these will be followed as closely as possible.

General Recommendations

7.5.1. Nothing, absolutely nothing, replaces good algorithms and good programming techniques. No matter how closely it follows the recommendations or how much it is embellished, a bad algorithm will always be a disgrace to the library.

7.5.2. Each function should have a precise objective. In this respect we think that functions that can be trivially implemented with other commands or functions, are a disservice to the user community. They take space in the libraries, manual, and minds without giving a substantial service.

7.5.3. Each function should have comments in its heading which, without much verbosity, explain the usage, purpose, author, level, algorithm and possibly some other useful information. Pages of comments where it is difficult to find the above information may be worse than no comments at all.

7.5.4. Each function should be accompanied by a test file which tests its correctness. Test files should not be tedious repetitions of the same situation, but instead the shortest and quickest program that explores all of the code in the function. Test files will normally grow with the examples that detected errors previously undetected. Such "errors" are pieces of code which run through a sensitive path and are, in general, excellent tests. Long and slow tests tend not to be run, and are self-defeating.

7.5.5. The code in the library is likely to be taken as an example for users and future implementors of Maple. Consequently we are doubly motivated to produce high quality code.

7.5.6. If a function resides in <any directory>/xxx.m then its source, that is the Maple source code that generates it, will be placed in <any directory>/src/xxx . The only exceptions to this rule are the functions that, for being thematically related and very short, are included in a single file. System library functions should be saved with a statement like

```
save cat( libname, `xxx.m` );
```

so that their ".m" files can be created in a portable way.

7.5.7. Files names (without the ".m"), and consequently function names, should be 9 characters long or shorter. This is not counting directories. This is caused by system limitations and the need to load with readlib(...). Internal functions defined entirely within another function body are not restricted by this limitation. Upper-lower case distinctions are not respected by some systems; consequently, different function names should not rely on case differences alone.

7.5.8. Local variables should be reasonably economized. Also, excessive use of local variables tends to reduce readability of the programs. E.g.,

```
for i to nops(expr) do ... od;
```

is more efficient and readable than

```
limit := nops(expr);
for i to limit do ... od;
```

The use of the RETURN(...) function typically eases the understanding of the flow of control and saves local variables. Simple operations on parameters may not be worth the assignment of local variables; for example, if op(1,param1) is used only twice, then assigning temp := op(1,param1) is not a real saving.

7.5.9. Global variables should be avoided. If unavoidable, global variables should be named starting with the at-sign (@). The Maple library convention for returning a "fail" condition (in cases where a direct FAIL return is inappropriate) is to return the global name @FAIL.

7.5.10. Data types should be used properly where needed. For example: a pair of two elements where order is important should be accommodated in a list; an indicator should only take the values true or false; etc.

7.5.11. Packages (collections of functions for a given purpose) should be structured according to the following example. Suppose users want to call X(...) directly. If X is nontrivial, it may optionally call sub-functions A, B, or C. Furthermore let us assume that any of X, A, B or C can call the lowest level functions E and F. Then:

(a) X should be the only name known to Maple, or the only name subject to be read with readlib(X).

(b) All of the functions that are likely to be loaded within the execution of X should be included in the module of X. That is to say that the number of loading operations should be minimized.

(c) The remaining functions in the package, which may or may not be loaded, should be defined in the module X as B:='readlib('B')', etc. This will cause the loading of B to be delayed until B(...) is used.

(d) When there are two or more possible entry points which share most of the package, then all of the definitions should be included in a single module. If X and Y are two entry points for the same package, all of the code for X and Y will be stored together (say in X.m). The initial definitions of X and Y will now be:

```
X := 'readlib('X')';
Y := 'readlib( 'Y', `` . libname . `X.m` )';
```

(e) Finally, a sub-function may be used directly by the user independently of the package. For example, if C could be used independently of X and Y then we need an entry for C. Within the package X we will define C as before, namely:

```
C := 'readlib('C')';
```

For the direct use of C we need to load its accompanying E and F; consequently, the definition for direct usage of C (not through X) will be:

```
C := 'readlib( 'C', ``.libname.`C.m`, ``.libname.`E.m`, ``.libname.`F.m` )';
```

7.5.12. The option 'remember' may be crucial for efficiency. It should be used when it is reasonably effective: whenever recomputation is likely. It should not be unnecessarily nested. Functions which produce side effects (printing of values, returning values through parameters, etc.) cannot use the option remember since this option reproduces the function result, not its side effects.

7.5.13. Atomized programming (splitting all steps of a computation) is not very efficient and, frequently, is unreadable (the "Assembler syndrome"). At the other extreme, "one-liners" are also unreadable (the "APL syndrome"). Both extremes should be avoided.

7.5.14. In complicated packages, it may be desirable to inform the user about the progress of a computation. Such printing should be regulated by printlevel. The values 2 and 3 are reserved for this purpose. For example:

```
if printlevel>2 then print(`Risch method applied`) fi;
```

7.5.15. It has proved to be valuable to have a "benchmark" for each function. A benchmark is a test file that not only tests for correctness but also for timing. When changes are done, it can be precisely measured if more/less time/space is used. Sometimes naive-looking modifications produce significant changes in performance.

7.5.16. Remember: Don't forget to use the *load* option "−1" when loading Maple library functions. (See section 8.3).

8. LIBRARY FUNCTIONS

Maple's library functions fall into three categories: functions internal to the Maple system, automatically-loaded library functions, and miscellaneous library functions that are not automatically loaded. Functions in the first category are coded internally in the basic Maple system. Functions in the second category are specified by Maple code in the Maple system library, and their names are initially assigned as unevaluated *readlib* functions (see section 6.7). The functions in the first two categories will be grouped together in this section since the user will not generally make any distinction between these two categories. In fact, the grouping of functions into these two categories may be different on different host systems. For a specific function 'f', the user can easily determine which of the first two categories it belongs to by entering Maple and displaying the value op('f'); the result will be the name 'f' for functions in the first category and the result will be 'readlib('f')' for functions in the second category. Functions in the third category will be listed separately at the end of this section because they cannot be used without being explicitly loaded by the user.

The general rule for function invocations in Maple is that all arguments are fully evaluated. Two exceptions are the functions *assigned* and *evaln* where the argument is evaluated to a name, a third exception is the function *evalb* where the argument is evaluated by the Boolean evaluator rather than by the general expression evaluator, and a fourth exception is the function *remember* where the argument involves a procedure invocation which will not be invoked. The names of the library functions are not reserved words in Maple. A user may define his own function using the same name as one of the system-supplied functions.

8.1. Standard Library Functions

8.1.1. abs (expr)

If expr is of type integer, rational, or floating point, then the absolute value of expr is returned, otherwise the function invocation remains unevaluated.

8.1.2. analyze (expr)

The purpose of this function is to analyze an expression in the following sense. The expression expr is viewed as a sum of products, each of the form:

$$\mathrm{const} * f_1^{e_1} * f_2^{e_2} * \ldots * f_n^{e_n}$$

where the f_i's and e_i's can be general expressions. If expr is a product (including the case of a single factor) then the value returned is the list

$$[\ \mathrm{const},\ f_1,\ e_1,\ \ldots,\ f_n,\ e_n\ ]$$

(where const will be 1 if there is no explicit constant in the product). If expr is a sum or an equation or a range then the function *analyze* is mapped onto expr. (See the function *map*).

8.1.3. anames ( )

This function takes no arguments. It returns an expression sequence consisting of all of the active names in the current Maple session which are *assigned names,* meaning names which have been assigned values other than their own names. (See also the function *unames*).

8.1.4. array (indexing_function, init_list, $lo_1..hi_1$, $lo_2..hi_2$, ...)

To create an expression of type array, a call is made to this function. The parameters *array* takes are an indexing function, initializations, and an array bound. Each of these is optional and they may appear in any order in the parameter sequence.

The indexing function is given either as a procedure or as a name. If one is not given, then a default of NULL is used. (Actually, that is the *only* way to obtain a NULL indexing function.)

The initializations are given either as a list of equations or as a list of values. (To avoid ambiguity, if a list of values is used, none of the values may itself be an equation.) If a list of equations is given, then for each equation, the left-hand side is used as the index of a component and the right-hand side is used as its value. With a list of values, consecutive integer indices are used starting at the low index specified in the index bound if an index bound is given, or at 1 if one is not given. The default for initializations is the empty list.

The index bound is passed as a number of integer ranges which appear adjacently in the parameter sequence. If no index bound is given, then one is deduced from the list of initializations. If the initializations are given as a list of equations, then the index bound is taken to be a sequence of ranges of the same length as the indices. Each range is made as restrictive as possible while still encompassing all the indices from the equations. If the initializations are given as a list of values then the array is taken to be one-dimensional and the index bound is a range from 1 to the number of values given. If as well as no index bound, no initializations are given (or if an empty list is given), then the array is taken to be zero-dimensional. (In this case, the only valid index is NULL.)

Examples:

```
array( );                      yields array([ ])
array([ ]);                    yields array([ ])
array(0..3);                   yields array(0..3,[ ])
array(1..4,0..3);              yields array(1..4,0..3,[ ])
array([x,y,z]);                yields array(1..3,[(1)=x,(2)=y,(3)=z])
array(0..3, [x,y,z]);          yields array(0..3,[(0)=x,(1)=y,(2)=z])
array([x,y,z], 0..3);          yields array(0..3,[(0)=x,(1)=y,(2)=z])
array([3=X,10=Y]);             yields array(3..10,[(3)=X,(10)=Y])
array(9..11, [10=Y]);            yields  array(9..11,[(10)=Y])
array([(1,2)=12, (2,7)=27]);     yields  array(1..2,2..7,[(1,2)=12,(2,7)=27])
array(sparse,[1=x,100=y]);       yields  array(sparse,1..100,[(1)=x,(100)=y])
```

8.1.5. assigned (name)

This function returns the value *true* if name is active in the current session and it has a value other than its own name, and returns the value *false* otherwise. The argument to this function must be a valid name. The argument is not fully evaluated but is evaluated to a name.

8.1.6. asympt (expr, x) or asympt (expr, x, n)

The purpose of this function is to compute the asymptotic expansion of expr with respect to the variable x (as x approaches infinity). If there is a third argument 'n' then it must evaluate to an integer which specifies the 'truncation degree' to be used. If there is no third argument then the 'truncation degree' is specified by the current value of the global variable *Degree* (which initially has the value 5 in the Maple system). Specifically, this function is defined in terms of the *taylor* function as follows:

```
subs( x=1/x, taylor( subs(x=1/x, expr), x=0, n ) )
```

(where the third argument 'n' to the *taylor* function will be omitted if it was omitted in the call to *asympt*).

8.1.7. cat (a, b, c, . . .)

This function takes an arbitrary number of arguments, which are evaluated and then concatenated to form either a name or an object of type '.'. The result of this function can be specified in terms of Maple's concatenation operator '.' as follows:

```
`` . a . b . c
```

(for the case of only three arguments, for example).

8.1.8. coeff (expr, x, n)

For this function the expression expr must be in expanded form (see the function *expand*). The value of this function is the coefficient in expr of the term involving x^n .

Examples: If

```
p := 70*y*x^4 - 70*x^4 - 177*x^2 + 19*y^5*x - 35*y^2 + 105
```

then

```
coeff( p, x, 0 );            yields   105-35*y^2
coeff( p, x, 1 );            yields   19*y^5
coeff( p, x, 2 );            yields   -177
coeff( p, x, 3 );            yields   0
coeff( p, x, 4 );            yields   70*y-70
```

8.1.9. convert (expr, class, arg_3, arg_4, ...)

The purpose of this function is to explicitly convert an expression from one type to another. Some of these conversions are coercions from one datatype to another, e.g., from a rational expression to a floating point number. Others of these conversions convert integer values into representations in bases other than decimal, e.g., conversion into binary format. Finally, there are conversions from one symbolic unit into another, The second parameter *class* specifies what type of conversion is required. Usually only two arguments are given when convert is called but some classes of conversion may require additional information. In that case these would be passed as the third and succeeding arguments.

If *class* is 'array', then an attempt is made to construct an array from the expression given. Only expressions of type 'list' or type 'table' may be converted. (One important use of this is to extend the bounds of an existing array, e.g., to add a column to a matrix.) The value returned by convert is a new array created by the array function. If extra arguments are given after 'array' in the call to convert, then they are used as the index bounds. When expr is a list, then the array is created using expr as the initialization list. When expr is a table, the initialization list used to create the array has an equation of the form index = expr[index] for each component of expr.

If *class* is 'binary' or 'octal', then convert returns an integer which *represents* the value of expr in binary or octal format.

If *class* is 'list', then convert returns a list containing all the operands of expr.

If *class* is 'metric', then expr may contain non-metric unit names, e.g., 5*ft+10*in. This expression is then converted into metric form. An additional third argument which may be either 'imp' or 'US' may be given in cases where there is a difference between Imperial units and U.S. units. A list of all the units which are known to the convert routine is given in Appendix A.

If *class* is 'name', then convert returns a name which looks like the expression.

For the case where *class* is 'polynom', if expr is not of type 'polynom' then it must be of type 'series' and the result is the polynomial obtained by removing the order term (if any) from the series and converting from the series data structure to the ordinary sum-of-products data structure.

For the case where *class* is 'rational', if expr is not of type 'rational' then it must be of type 'float' and a rational number is generated which approximates the given floating point number. The accuracy of the approximation depends on the number of significant digits in the input floating point number.

If *class* is 'series', then expr is expected to be a polynomial and this polynomial is converted into a power series form.

If *set* is 'set', then convert returns a set containing all the operands of expr.

If *class* is `*`, the all the op's of the expression are mulitiplied together to give the result.

If *class* is `+`, then all the op's of the expression are added together to give the result.

Examples:

```
convert( [a,b,c], array )          yields     array( 1..3, [(1)=a,(2)=b,(3)=c] )
table( [ (1,1)=11, (1,2)=0 ] );
convert( ", array )  yields        array( 1..1, 1..2, [(1,1)=11, (1,2)=0] )
convert( ", array, 0..3, 0..3 ) yields       array( 0..3, 0..3, [(1,1)=11, (1,2)=0] )

convert( 9, binary )  yields       1001
type( ", integer )    yields       true
convert( 15, octal )  yields       17

convert( x+y, list )  yields       [ x, y ]
convert( -13, set )   yields       { -13 }

convert( 3*inch, metric )          yields     7.62*cm

convert( 8, name )    yields       8
type( ", name )       yields       true

s := taylor( sin(x), x=0 );        yields     s := 1*x+(-1/6)*x^3+1/120*x^5+O(x^6)
convert( s, polynom );             yields     x-1/6*x^3+1/120*x^5

convert( 3.14, rational )          yields     22/7
convert( 3.1415, rational )        yields     311/99
convert( 0.30, rational )          yields     1/3
convert( 0.300, rational )         yields     3/10

convert( 1+x, series )             yields     1 + 1*x

convert( [a,b,c], '*' )            yields     a*b*c
```

User Interface: New conversion procedures can be made known to the *convert* function by the following mechanism. If the user assigns a procedure to the name `conv/newtype` (where 'newtype' is any name chosen by the user) as in

```
`conv/newtype` := proc ( expr, <extra parameters> ) . . . end
```

then the function invocation

convert (expr, newtype, <extra parameters>)

will cause the function invocation

`conv/newtype` (expr, <extra parameters>) .

If `conv/newtype` is not assigned then Maple looks for it in the Maple system library at the pathname

cat(libname, `conv/newtype.m`)

and if it is not found then an error occurs.

8.1.10. copy (expr)

The *copy* function returns a copy of its parameter. The primary use of this function is to copy tables.

A table is the only type of data object which can be modified after creation. (A table is the only type of object for which it is possible to make an assignment to a part of the object.) Therefore it would only ever be necessary to use *copy* when 'expr' was a table or had a table as a subexpression. For other inputs, *copy* simply returns its parameter.

copy is applied recursively to the subexpressions of 'expr' so that all tables in it are copied. The expression returned is immune to side effects caused by assignment to components of tables that existed when *copy* was called.

Examples:

```
copy(2 + sin(x));                  yields 2+sin(x)
copy(proc() a end);                yields proc() a end

u := table([X]);                   yields u := table([(1)=X])
v := u;                            yields v := table([(1)=X])
w := copy(u);                      yields w := table([(1)=X])

u[1] := 8;
v[1];                              yields 8
w[1];                              yields X

L := [u,u]; N := copy(L); u[1] := 9;

L;                                 yields [table([(1)=9]),table([(1)=9])]
N;                                 yields [table([(1)=8]),table([(1)=8])]
```

8.1.11. degree (expr, x)

If expr is a polynomial in x (allowing both positive and negative exponents) then this function returns the degree of expr in x. It is not necessary that expr be in expanded form. This function may be applied as well to the series data structure. If expr is neither a series in the indeterminate x nor a polynomial in the indeterminate x then the value returned is @FAIL. (See also *ldegree()*).

8.1.12. denom (expr)

This function computes the common denominator of an expression. Specifically, it first applies the *analyze* function to expr. Then it extracts from each term the denominator of the constant factor and all factors whose exponents have negative sign, and forms the least common multiple of the denominators thus extracted from each term.

8.1.13. diff (expr, x_1, x_2, . . ., x_n)

This function computes the partial derivative of expr with respect to x_1, x_2, ..., x_n, respectively. The latter n expressions must evaluate to names. In the case where n is greater than one, the syntax is simply a shorthand notation for nested applications of the *diff* function.

Examples: Assuming that x and y are names which stand for themselves, if the following statements are executed:

```
p := -30*x^3*y + 90*x^2*y^2 + 5*x^2 - 6*x*y;
diff(p, x, y);
```

then the result of the function invocation of 'diff' is:

```
-90*x^2+360*x*y+(-6)
```

This is equivalent to executing the statement diff(diff(p,x), y) .

User Interface: New functions can be made known to Maple's *diff* function by the following mechanism. If the user assigns a procedure to the name `diff/newfcn` (where 'newfcn' is any name chosen by the user) as in

```
`diff/newfcn` := proc (expr,x) newfcn1(expr) * diff(expr,x) end
```

(where the name 'newfcn1' is being used as the name of the derivative function) then the function invocation

```
diff ( newfcn(expr), x )
```

will cause the function invocation

```
`diff/newfcn` ( expr, x ) .
```

If `diff/newfcn` is not assigned then Maple looks for it in the Maple system library at the pathname

```
cat( libname, `diff/newfcn.m` )
```

and if it is not found then a FAIL return occurs from the *diff* function.

Functions whose derivatives are currently defined in the Maple system library include the elementary functions (all of the circular, inverse circular, hyperbolic, and inverse hyperbolic functions, as well as the functions exp and ln), abs, GAMMA, Psi (which satisfies the relationship

```
Psi(x) = diff(GAMMA(x),x) / GAMMA(x) ),
```

and the first four derivatives of Psi (represented by the names Psi1, Psi2, Psi3, and Psi4). The derivative of an unevaluated 'int' function is also defined in the Maple system library.

8.1.14. divide (a, b, 'q')

The purpose of this function is to attempt to perform exact polynomial division of expression 'a' by expression 'b'. The division is considered successful only if the resulting quotient is a 'true polynomial' in its indeterminates -- i.e., negative exponents are not acceptable in the result of a polynomial division. The value of the *divide* function is 'true' if the division was successful, 'false' otherwise. Furthermore, if there is a third argument 'q' (which must evaluate to a name) and if the division was successful then the value of the quotient is assigned to q. In the case of an unsuccessful division the value of q will remain unaffected.

Examples:

```
a := 7*y^3*x^4 - 2*y*x^3 - (y^4 - 21*y^3)*x^2 - 6*y*x - 3*y^4;
b := y*x^2 + 3*y;
divide(a, b, 'q');                      yields    true
q;                                      yields    7*y^2*x^2-2*x-y^3

r := expand( (2*x-5)^3 * (x+1) );
while divide(r, 2*x-5, 'r') do od;
r;                                      yields    x+1

f := expand((c-1)/c);                   yields    1-c^(-1)
g := c-1;                               yields    c-1
divide(f, g);                           yields    false
```

In the latter example, note that it is possible to simplify the expression f/g to the value c^(-1) but this cannot be accomplished by the *divide* function because the result is not a 'true polynomial'. For this purpose, the *normal* function should be used, as in:

```
f/g;                                    yields    (1-c^(-1))/(c-1)
normal(");                              yields    c^(-1)
```

8.1.15. ERROR (<expr seq>)

This function is a special function whose purpose is to cause an immediate exit from a procedure. (See section 5.5). Upon execution of this function, control returns to the top level of the Maple system and the message "ERROR: " followed by the values of the expression sequence given in the parameter list.

Example:

```
ERROR( in, 'f', x, x**2 );     prints     ERROR: in, f, 3, 9
```

8.1.16. evalb (expr)

This function invokes the Boolean expression evaluator on expr. For example, the expression a = b will be considered an algebraic equation if it does not appear in an explicit Boolean context, but evalb(a = b) will evaluate the equation as a Boolean (i.e., it will evaluate the equation to the value 'true' or to the value 'false').

8.1.17. evalc (expr)

This function evaluates and simplifies the complex expression expr. It uses the global variable I for (−1)**(1/2).

Examples:

```
evalc( (3 + 5*I) * (7 + 4*I) ); yields     1 + 47*I
evalc( (5 − I) / (1 + 2*I) );    yields     3/5 − 11/5*I
evalc( (9 + 8*I) ^ 2 );          yields     17 + 144*I
```

8.1.18. evaln (name)

The purpose of this function is to apply to the argument 'name' Maple's name evaluator, which is the evaluator that is always applied to left-hand-sides of assignments, for example. The argument must be a syntactically valid name and, of course, it is not fully evaluated. One of the uses for this function is to *unassign* names formed with the concatenation operator. For example,

```
for i to 5 do a.i := evaln(a.i) od
```

will unassign the names a1, a2, a3, a4, and a5. Note that in this case the *evaln* function cannot be replaced by the use of the unevaluated expression construct 'a.i' because then the concatenation on the right-hand-side will remain unevaluated (and the names a1, . . . , a5 will remain *assigned*).

8.1.19. evalf (expr) or evalf (expr, n)

This is the 'evaluate to a floating point form' function, which evaluates the argument 'expr' to a floating point number (if possible). If there is no second argument then the number of significant digits appearing in the result is controlled by Maple's global variable *Digits*. (The initial value of the global variable *Digits* is 10, but the user may assign any integer value to this global variable). If there is a second argument 'n' to the *evalf* function then it must evaluate to an integer, and the number of significant digits appearing in the result is determined by the value of n.

Examples:

```
a := (5^40 + 3^50) / 2^90;      yields
          a := 4547832457858487115869580437/618970019642690137449562112
evalf(a);            yields     7.3474196060
evalf(a, 40);        yields     7.3474196060154781089322604350878881161323

Digits := 25;
evalf( 5/3 * exp(-2) * sin(Pi/4) );        yields     .159494160850684873267980
```

User Interface: New functions, and also new constants, can be made known to Maple's *evalf* function by the user.

For the case of new functions, if the user assigns a procedure to the name `float/newfcn` (where 'newfcn' is any name chosen by the user) as in

```
`float/newfcn` := proc (x)
                  local t;
                  t := evalf(x);
                  evalf( exp(t^2) * sin(Pi/2 * t) )
                  end
```

then the function invocation

```
evalf ( newfcn(x) )
```

will cause the function invocation

```
`float/newfcn` ( x ) .
```

If `float/newfcn` is not assigned then Maple looks for it in the Maple system library at the pathname

```
cat( libname, `float/newfcn.m` )
```

and if it is not found then an error occurs.

For the case of new constants, if the user assigns a procedure to the name `float/constant/newconst` (where 'newconst' is any name chosen by the user) as in

```
`float/constant/newconst` := proc () evalf( (5^(1/2) - 1) / 2 ) end;
```

then the function invocation

```
evalf ( newconst )
```

will cause the function invocation

```
`float/constant/newconst` ( ) .
```

If `float/constant/newconst` is not assigned then Maple looks for it in the Maple system library at the pathname

```
cat( libname, `float/constant/newconst.m` )
```

and if it is not found then an error occurs.

Functions for which 'evalf' procedures are currently defined in the Maple system library include the elementary functions (all of the circular, inverse circular, hyperbolic, and inverse hyperbolic functions, as well as the functions exp and ln), and the functions GAMMA, Psi, Psi1, and zeta. Constants for which 'evalf' procedures are currently defined in the Maple system library include:

Pi, e (exp(1)), gamma (Euler's constant), and Catalan (Catalan's constant) .

8.1.20. expand (expr) or expand (expr, e_1, e_2, . . ., e_n)

The purpose of this function is to expand expr by distributing products over sums. It also expands certain function calls. If the number of arguments is greater than one then the additional arguments e_1, e_2, ..., e_n are expressions which will be 'frozen' (i.e., the effect is to replace every occurrence of e_i by a name before performing the *expand* operation and then to restore the original expression e_i unchanged). If the expression is an equation then *expand* is applied to the operands of the equation. The functions that *expand* knows are:

bigprod, bigpow, C, cos, exp, factorial, large, ln, power, sin, prodequa

Examples:

```
p := (2*x - 5) * (35*x^2 - x + 7);
expand(p);                    yields     70*x^3-177*x^2+19*x+(-35)

q := 3*sin(x) * (x*sin(x) - y*z) * (2*x^2 - 3);
expand(q);                    yields
          6*sin(x)^2*x^3-9*sin(x)^2*x-6*sin(x)*y*z*x^2+9*sin(x)*y*z

r := 3*(x+1)^3 - 5*(x+1)^2;
expand(r, x+1);               yields     3*(x+1)^3-5*(x+1)^2
expand(r);                    yields     3*x^3+4*x^2-x+(-2)

expand( C(n,r) )              yields     n!/r!/(n-r)!
expand( 7*cos(2*x) ) yields   7*cos(x)^2 - 7*sin(x)^2
expand( (n+1)! )              yields     (n+1)*n!
expand( (x+1)^3 )             yieldsx^3 + 3*x^2 + 3^x + 1
expand( sin(x+y) )            yields     cos(x)*sin(y) + cos(y)*sin(x)
expand( sin(x+y), sin )       yields     sin( x+y )
```

User Interface:

New functions can be made known to Maple's *expand* function by assigning a procedure to the name 'expand/fcn' where 'fcn' can be any name chosen by the user. To illustrate, we will give a definition for 'expand/tan':

```
'expand/tan' := proc( x )
      subs( cos='@ONE', sin=tan, expand( sin(x)/cos(x) ) );
      ''
      end;
@ONE := proc() 1 end;
```

The function invocation

```
expand( tan( 2*x ) );
```

will in turn invoke

```
'expand/tan'( 2*x );
```

which returns

```
2*tan(x)/(1-tan(x)^2) .
```

If 'expand/tan' is not assigned then Maple looks for it in the Maple system library at the pathname

```
cat( libname, 'expand/tan.m' );
```

and if that file is not found, then the unexpanded function call

```
tan( 2*x )
```

is returned.

The functions *expandoff* and *expandon* (c.f. *expandoff* and *expandon* in section 8.2) can be use to selectively supress or apply Maple's knowledge of how function calls are to be expanded.

8.1.21. factor (expr)

This function computes a complete factorization over the integers of the multivariate polynomial expr. (Work on this function has not been completed at the time of writing).

8.1.22. Float (m, exp)

This is a special function used to specify a *floating point number*. The arguments to this function must evaluate to integers, and the value of Float(m, exp) is the floating point number

```
m * 10^exp .
```

This function is particularly useful for specifying a floating point number with a very large or a very small magnitude, as in Float(173, 21) or Float(1952135, −30).

8.1.23. frac (a)

This function computes the *fractional part* of a rational number. It is the complement of the *trunc* function; the value of frac(x) is specified by

x − trunc(x) .

8.1.24. gcd (a, b, 'result1', 'result2')

This function computes the *greatest common divisor* of the multivariate polynomials 'a' and 'b'. It is an error if 'a' and 'b' are not polynomials in their indeterminates. The gcd is computed in the domain of polynomials with integer coefficients, but the input polynomials may have rational coefficients in which case the common denominator is simply removed. If the third argument 'result1' is present then it must evaluate to a name and upon return its value will be a / gcd(a,b) . Similarly, if the fourth argument 'result2' is present then it must evaluate to a name and upon return its value will be b / gcd(a,b) .

8.1.25. has (expr1, expr2)

The value of this function is *true* if expr1 contains expr2 as an explicit subexpression, *false* otherwise. The concept of 'explicit subexpression' corresponds to the semantics of the *op* function: if op(expr1), or recursive application of op to each operand of expr1, yields expr2 as an operand then expr1 contains expr2 as an 'explicit subexpression'; otherwise it does not.

Examples:

```
has ( (a+b)^(4/3), a+b );      yields    true
has ( (a+b)^(4/3), a );        yields    true
has ( a+b+c, a+b );            yields    false
```

8.1.26. icontent (expr)

This function computes the integer content of expr -- i.e., the greatest common divisor of the integer coefficients in the case of an expanded polynomial. If expr is not in expanded form then the *icontent* function is mapped onto its components to obtain the result. For the common case of an expanded polynomial with integer coefficients, this function has a concise definition in terms of the functions *lcoeff*, *map*, *igcd*, and *op* as follows:

igcd(op(map(lcoeff, [op(expr)])))

8.1.27. ifactor (n)

This function returns the complete factorization of its integer argument n. The answer is in the form of a product of powers, where the exponent of the powers is an integer and the base of the powers is the null string function, whose argument is an integer.

8.1.28. igcd (i, j, k, . . .)

This function takes an arbitrary number of arguments which must evaluate to integers, and it computes the nonnegative ***greatest common divisor*** of these integers. If *igcd* is called with no arguments then the value 0 is returned.

Examples:

```
igcd();                    yields    0
igcd( 3 );                 yields    3
igcd( −10, 6, −8 );        yields    2
```

8.1.29. ilcm (i, j, k, . . .)

This function takes an arbitrary number of arguments which must evaluate to integers, and it computes the nonnegative ***least common multiple*** of these integers. If *ilcm* is called with no arguments then the value 0 is returned.

Examples:

```
ilcm();                    yields    0
ilcm( −5 );                yields    5
ilcm( 7, −6, 14 );         yields    42
```

8.1.30. imodp (n, p)

The functions *imodp* and *imods* are two functions for computing the integer modular operation

n mod p .

The final letter 'p' or 's' in the function name stands for 'positive range' or 'symmetric range'. If n and p are integers then the function imodp(n,p) returns an integer r lying in the 'positive range':

$$0 \leq r < \mathrm{abs}(p),$$

where n=p*q + r for some integer q. If p is zero then an error occurs. Note that the *imodp* function satisfies the property:

imodp(n, p) = imodp(n, −p) .

Examples:

```
imodp( 7, 5 );             yields    2
imodp( 8, 5 );             yields    3
imodp( −8, −5 );           yields    2
imodp( 7, −6 );            yields    1
imodp( −7, 6 );            yields    5
```

8.1.31. imods (n, p)

The functions *imods* and *imodp* are two functions for computing the integer modular operation

n mod p .

The final letter 's' or 'p' in the function name stands for 'symmetric range' or 'positive range'. If n and p are integers then the function imods(n,p) returns an integer r lying in the 'symmetric range':

$$-\text{abs}(p)/2 < r \leq \text{abs}(p)/2,$$

where n=p*q + r for some integer q. If p is zero then an error occurs. Note that the *imods* function satisfies the property:

imods(n, p) = imods(n, −p) .

Examples:

```
imods( 7, 5 );         yields    2
imods( 8, 5 );         yields   -2
imods( -8, -5 );       yields    2
imods( 9, -6 );        yields    3
imods( -9, 6 );        yields    3
```

8.1.32. indets (expr)

The purpose of this function is to determine the indeterminates which appear in expr. The value of the function is a set whose elements are the indeterminates. The concept of 'indeterminate' is that expr is viewed as a rational expression (i.e. an expression formed by applying only the operations +, −, *, / to some given symbols) and therefore unevaluated functions such as sin(x), exp(x^2), f(x,y), and x^(1/2) are treated as indeterminates. When an indeterminate which is not a name appears in the set then so will all of its component indeterminates. Expressions of type 'constant' such as sin(1), f(3,5), and 2^(1/2) are not considered to be indeterminates. Note that if expr is a sum or product of terms t_1, t_2, · · · , t_n then the result of applying indets(expr) will be identical to the result of applying the set union:

indets(t_1) + indets(t_2) + . . . + indets(t_n) .

Examples: If the following statements are executed:

```
p := 3*x^3*y^4*z - 2*x^2*z^2 + y^3*z - 7*y + 5;
r := (2*x^2 - 5) * (x - 2)^(1/3) / (x*exp(x^2));
```

then

```
indets( p );           yields    { z, y, x }
indets( r );           yields    { exp(x^2), (x+(-2))^(1/3), x }
```

Furthermore,

```
indets( exp(x^2) );          yields    { exp(x^2), x }
indets( x^(1/2) );           yields    { x^(1/2), x }
indets( 2^(1/2)*f(9) );      yields    {}
```

8.1.33. indices (tbl)

This function takes a table as its parameter and constructs an expression sequence containing the indices of all entries in that table. Each index is made into a list and the expression sequence returned has these lists as its components. The indices are placed in lists to prevent indices that are themselves expression sequences from merging.

Examples:

```
table([(1,2)=A, (2,1)=B, 9=C]);
indices(");                         yields  [1,2],[9],[2,1]
indices( table( ) );                        yields  the value of NULL
array([11, 22, 33, 44]);
indices(");                         yields  [2],[3],[4],[1]
```

The indices will appear in the expression sequence in an apparently arbitrary order. The order in which they appear can not easily be controlled by the user.

The *indices* function is useful for performing actions which use all entries in a given table. For example, the following procedure will remove all zero-valued entries from a table:

```
remove_zeros :=
proc( tbl )
        local i, ix_set, index;
        ix_set := {indices(tbl)};
        for i to nops(ix_set) do
                op(i, ix_set);      # get the i-th list
                index := op(");  # convert it to an expression sequence
                if tbl[index] = 0 then
                        tbl[index] := evaln(tbl[index])
                fi
        od
end;
```

8.1.34. int (expr, x) or int (expr, x = a..b)

If the second argument is not an equation then this function attempts to compute the indefinite integral of expr with respect to the second argument 'x' which must evaluate to a name. If the second argument is an equation then its left-hand-side must evaluate to a name 'x' and its right-hand-side must evaluate to a range 'a..b', and this function attempts to compute the definite integral of expr with respect to 'x' over the interval specified by the range 'a..b'. If Maple is not successful in performing the integration then a FAIL return occurs, meaning that the value of the function invocation is the unevaluated function invocation.

Examples: If

```
        f := 1/2*x^(-2) + 3/2*x^(-1) + 2 - 5/2*x + 7/2*x^2;
```

then

```
        int( f, x );            yields    -1/2*x^(-1)+3/2*ln(x)+2*x-5/4*x^2+7/6*x^3
        int( f, x = 1..2 );     yields    20/3 + 3/2*ln(2)
        evalf(");               yields    7.706387438

        int((x-1)/(x+1), x);    yields    (x-1)*ln(x+1)-(x+1)ln(x+1)+x+1
        expand(");              yields    -2*ln(x+1)+x+1

        int( tan(x), x );       yields    -ln(cos(x)
        int( sin(t)*cos(t), t );yields    1/2*sin(t)^2
        int( x*cos(x), x );     yields    x*sin(x)+cos(x)

        int( exp(x^2), x );     yields    int(exp(x^2),x)
```

8.1.35. iquo (m, n)

This function computes the *integer quotient* of 'm' divided by 'n'. The result of this function is identical with the result of applying trunc(m/n). The *iquo* function will be more efficient than the latter when 'm' and 'n' are long integers because it avoids first simplifying the rational number m/n to lowest terms. Specifically, if m and n are integers then the function iquo(m,n) returns an integer q satisfying

```
        m = n*q + r
```

for some integer r such that

        abs(r) < abs(n)   and   m*r ≥ 0 .

If n is zero then an error occurs.

Examples:

```
iquo( 7, 5 );          yields    1
iquo( -7, 5 );         yields   -1
iquo( 7, -5 );         yields   -1
iquo( -7, -5 );        yields    1
```

8.1.36. isprime (n) or isprime (n, iter)

This function uses a heuristic method to determine whether n is prime. It returns false if it can prove in iter (ten, if no second argument is given) iterations that n is composite; it returns true otherwise.

8.1.37. isqrt (n) or isqrt (n, x)

This function computes the closest integer to the square root of n. If called with two arguments, the second one is used as a first approximation for the square root of n.

8.1.38. ithprime (i)

This function returns the ith prime.

8.1.39. lcm (a, b)

This function computes the *least common multiple* of the multivariate polynomials 'a' and 'b'. It is an error if 'a' and 'b' are not polynomials in their indeterminates. This function invokes the *gcd* function, using the definition

```
lcm( a, b )  =  a*b / gcd(a,b)
```

(with an adjustment of the *sign* of the result to make the leading coefficient positive). Restrictions on the input expressions are therefore the restrictions of the *gcd* function.

8.1.40. lcoeff (expr) or lcoeff (expr, x) or lcoeff (expr, x, y, ...)

This function returns the *leading coefficient* of the multivariate polynomial expr, with respect to the set of indeterminates of expr, i.e., indets(expr), if lcoeff is called with only one argument. If lcoeff is called with more than one argument, the second and succeeding arguments are taken to be the set of indeterminants of the first argument. If expr is in expanded form and if the set of indeterminants are {x.1, x.2, . . . , x.n} then the result of the *lcoeff* function can be expressed as follows:

```
u := expr;
for i to n while not type(u, constant) do
  u := coeff( u, x.i, degree(u, x.i) )
od;
u
```

It is an error if expr is not a polynomial in its indeterminates.

Examples:

```
p := 31*x^4*y^4 + 2*x^3*y^3*z - y^2*z^5;
indets( p );                              yields    { z, y, x }
lcoeff( p );                              yields    -1

q := 17*x^5 + x^3 - 5*x^2 + 111;
indets( q );                              yields    {x}
lcoeff( q );                              yields    17

r := 3/2*y^3 - 5/2*ln(2)*x^5*y + x^4*y - 1;
indets( r );                              yields    { x, y }
lcoeff( r );                              yields    -5/2*ln(2)

s := u*v^2*w^3*x^4;
lcoeff( s );                              yields    1
lcoeff( s, u, w );                        yields    v^2*x^4
```

8.1.41. ldegree (expr, x)

This function is a companion to the *degree* function. If expr is a polynomial in x (allowing both positive and negative exponents) then this function returns the *low degree* of expr in x, which is the least exponent of x in expr. It is not necessary that expr be in expanded form. This function may be applied as well to the series data structure. If expr is neither a series in the indeterminate x nor a polynomial in the indeterminate x then the value returned is @FAIL.

8.1.42. length (n)

For this function, the argument 'n' must evaluate to either a string or an integer. The value returned is the length of the string or of the integer respectively. The length of the integer is the number of digits in its base-10 representation, with the sign of the integer being irrelevant.

8.1.43. lexorder ($name_1$, $name_2$)

This function tests to determine whether $name_1$ and $name_2$ are in lexicographical order. It returns *true* if $name_1$ occurs before $name_2$ in lexicographical order, or if $name_1$ is equal to $name_2$. Otherwise, it returns *false*. The lexicographical order depends in part upon the collation sequence of the underlying character set, which is system-dependent. For names consisting of ordinary letters, lexicographical order is the standard alphabetical order.

Examples: For a typical implementation of the ASCII character set the following results are obtained:

```
lexorder( a, b );                          yields    true
lexorder( A, a );                          yields    true
lexorder( `  a`, a );                      yields    true
lexorder( John, Harry );                   yields    false
lexorder( determinant, determinate );      yields    true
lexorder( greatest, great );               yields    false
lexorder( `*`, `^` );                      yields    true
lexorder( `^`, `+` );                      yields    true
```

8.1.44. limit (expr, x = a)

This function attempts to compute the limiting value of expr as x approaches a. The second argument must be an equation and the left-hand-side of the equation must evaluate to a name. This function applies the *taylor* function and deduces the limit from the form of the taylor series. The limit point 'a' may take the special value 'infinity' in which case the limiting value is determined by applying the change of variable x = 1/x in expr and then computing the taylor series about x = 0.

Examples:

```
limit( sin(x)/x, x=0 );                    yields    1
limit( (tan(x)-x)/x^3, x=0 );              yields    1/3
limit( (5*x-3)/(x^2+x+1), x=infinity );    yields    0

r := (x^2 - 1) / (11*x^2 - 2*x - 9);
limit( r, x=0 );                           yields    1/9
limit( r, x=infinity );                    yields    1/11
limit( r, x=1 );                           yields    1/10

limit( 1/x, x=0 );                         yields    infinity
limit( 1/x, x=infinity );                  yields    0
```

8.1.45. lprint (a, b, ...);

This function prints its arguments in line-print mode. Each argument is printed as well as possible on a single line.

8.1.46. map (f, expr, arg_2, arg_3, ..., arg_n)

For this function, f must evaluate to a name or to a procedure definition. The purpose of this function is to map f (as a function name or as a procedure invocation) onto the components of expr. The result of the *map* function is a new expression which can be defined as follows: replace the ith operand in expr by the result of applying f to the i^{th} operand, for i = 1, 2, . . . , nops(expr). If f takes more than one argument then there must be additional arguments to the *map* function: arg_2, arg_3, · · · , arg_n which are simply passed through as the 2^{nd}, 3^{rd}, · · · , n^{th} arguments to f.

Examples:

```
map( f, x + y*z );                        yields    f(x) + f(y*z)
map( f, y*z );                            yields    f(y)*f(z)
map( f, {a,b,c} );                        yields    {f(a), f(b), f(c)}

map( proc (x) x^2 end,  x + y );          yields    x^2 + y^2
map( proc (x) x^2 end, [1,2,3,4] );       yields    [1,4,9,16]

map( int, [exp(t),ln(t),tan(t)], t );     yields    [exp(t), t*ln(t)-t, -ln(cos(t))]

expr := 2/3 * x/sin(x) - 1/x + sin(x);
den := 3*x*sin(x);
map( proc (e,m) m*e end, expr, den );     yields    2*x^2 - 3*sin(x) + 3*x*sin(x)^2

mult := proc (e,m) m*e end;
map( mult, h(u,v,w), 10 );                yields    h( 10*u, 10*v, 10*w )
```

8.1.47. maparray (f, A, arg2, arg3, ...)

This procedure applies a function to each component of an array. The parameter 'f' must evaluate to a procedure or to a name and the parameter 'A' must evaluate to an array. There may be zero or more additional parameters and they may be of any type.

The procedure *maparray* makes one call

```
f(evaln(A[index]), arg2, arg3, ... )
```

for each index within the index bounds of A. The value 'A' is returned. At the present time, *maparray* must be loaded by the user from the maple library.

As an example, to assign zero to all components of a 3 by 3 array, the following statement may be used:

```
maparray( proc(a) a := 0 end,  array(1..3, 1..3) );
```

To print the elements of an array, A, in a readable order, one could use

```
maparray( proc(a) a; print(a, ` = `, '') end,  'A' );
```

The procedure below does component-wise addition of an arbitrary number of arrays:

```
# Use A := addarray(X, Y, ...) to give A := X + Y + ...

addarray := proc()
        local bd, i;
        if nargs = 0 then ERROR(`nothing to add`) fi;
        bd := op(2,param(1));
        for i from 2 to nargs do
                if op(2,param(i))<>bd then ERROR(`different shapes`) fi
        od;
        proc(A) local i, ix, sum;
                ix := op(A);
                sum := 0;
                for i from 2 to nargs do param(i); sum := sum + "[ix] od;
                A := sum
        end;
        maparray(", array(bd), paramseq)
end;
```

8.1.48. max (a, b, c, . . .)

This function takes an arbitrary number of arguments. If each of the arguments evaluates to an integer, a rational number, or a floating point number, then the value of this function is the maximum of these numbers. If one or more of the arguments does not evaluate to such a constant then a FAIL return occurs. It is an error if *max* is called with no arguments.

Examples:

max(3/2, 1.49);	yields	3/2
max(3/5, evalf(ln(2)), 9/13);	yields	.6931471805
max(5);	yields	5
max(−1001, 1/2, −1/2, −9);	yields	1/2
max(x, y);	yields	max(x,y)

8.1.49. member (expr, set_or_list, 'position')

The purpose of this function is to test for set membership or to test for list membership and (optionally) to locate the position of expr in a list. The second argument 'set_or_list' must be either a set or a list, and this function returns *true* if expr is one of the elements in set_or_list, *false* otherwise. If the third argument is present then it must evaluate to a name, the second argument must be a list, and, in the case where the value of this function is *true*, the position of expr in the list will be assigned to the third argument.

Examples:

```
member( y, {x,y,z} );                           yields   true
member( y, {x*y, y*z} );                        yields   false
member( x, {} );                                yields   false
member( 3*exp(x/2), {sin(x), 3*exp(x/2)} );     yields   true

member( w, [x,y,w,u] );                         yields   true
member( w, [x,y,w,u], 'k' );                    yields   true
k;                                              yields   3
member( x, [x+y, x-y, x*y, x/y], 'k' );         yields   false
member( x+y, [x+y, x-y, x*y, x/y], 'k' );       yields   true
k;                                              yields   1
```

8.1.50. min (a, b, c, . . .)

This function takes an arbitrary number of arguments. If each of the arguments evaluates to an integer, a rational number, or a floating point number, then the value of this function is the minimum of these numbers. If one or more of the arguments does not evaluate to such a constant then a FAIL return occurs. It is an error if *min* is called with no arguments.

Examples:

```
min( 3/2, 1.49 );                               yields   1.49
min( 3/5, evalf(ln(2)), 9/13 );                 yields   3/5
min( evalf(ln(2)), evalf( (5^(1/2)-1)/2 ) );    yields   .6180339890
min( -1001, 1/2, -1/2, 9 );                     yields   -1001
min( x, y );                                    yields   min(x,y)
```

8.1.51. minv (n, p)

This function returns the multiplicative inverse of the integer n in Zp (the ring of integers modulo p). It uses the symmetric range.

Examples:

```
minv( 5, 11 )    yields    -2
minv( 3, 4 )     yields    -1
```

8.1.52. modp (a, p)

This function applies imodp (with modulus p) to the coefficients of the expanded polynomial a. (Also see imodp(), mods().)

Examples:

```
modp( −1, 13 )              yields    12
modp( 1/3, 7 )              yields    5
modp( x^2 − 5*x + 8, 5 )    yields    x^2+3
modp( 17*x*y^3, 11 )        yields    6*x*y^3
```

8.1.53. mods (a, p)

This function applies imods (with modulus p) to the coefficients of the expanded polynomial a. (Also see imods(), imodp().)

Examples:

```
mods( 9, 13 )               yields    −4
mods( 1/3, 7 )              yields    −2
mods( x^2 − 5*x + 8, 5 )    yields    x^2 − 2
mods( 17*x*y^3, 11 )        yields    −5*x*y^3
```

8.1.54. mquo (a, b, p) or mquo (a, b, p, r)

This function returns the quotient q of a/b and optionally the remainder r where a = b*q + r and where a, b, q, and r are in Zp[x] (polynomials in x with coefficients in the ring of integers modulo p). If the fourth argument is specified, it must evaluate to a name into which the remainder is assigned. (See also mrem(), rem().)

Example:

```
mquo( x^4 + 5*x^3 + 6, x^2 + 2*x + 7, 13 )      yields     3*x + x^2
```

8.1.55. mrem (a, b, p) or mrem (a, b, p, q)

This function returns the remainder r of a/b and optionally the quotient q where a = b*q + r and where a, b, q, and r are in Zp[x] (polynomials in x with coefficients in the ring of integers modulo p). If the fourth argument is specified, it must evaluate to a name into which the quotient is assigned. (See also mrem(), rem().)

Example:

```
mrem( x^4 + 5*x^3 + 6, x^2 + 2*x + 7, 13 )      yields     3*x + x^2
```

8.1.56. nextprime(n)

This function returns the smallest prime that is larger than n. The argument n must evaluate to an integer value. (Also see prevprime().)

8.1.57. nops (expr)

The purpose of this function is to determine the number of operands appearing in expr. The manner in which expr is viewed by this function corresponds to the manner in which an expression is viewed by the function *op*. In the most common case, expr has operands indexed from 1 to n (such as in a general algebraic expression, a set, or a list) and nops(expr) is n. If expr is a function invocation with operands indexed from 0 to n then nops(expr) is n. If expr is a series with operands indexed from 0 to n then nops(expr) is n.

Examples: Assuming that f, x, y, and z are names which stand for themselves, if the following statements are executed:

```
g := f(x, y, z);
a := (3*sin(x^3) - (2/3)*x + y) / (2*x^2 - 1);
```

then

```
nops( g );              yields   3
nops( a );              yields   2
nops( op(1, a) );       yields   3
nops( op(2, a) );       yields   2 .
```

Note that the latter result of 2 is not because the denominator of 'a' is the expression

```
2*x^2+(-1)
```

which is an addition of two terms but rather op(2,a) is the expression

```
(2*x^2+(-1))^(-1)
```

which is a power (and a power necessarily consists of exactly two operands).

8.1.58. normal (expr)

This function normalizes expr into the *factored normal form*, i.e., into the form

```
numerator / denominator
```

where numerator and denominator are relatively prime. In the general case, each of 'numerator' and 'denominator' will be left in factored form as far as possible (without actually performing any factorization) subject to the condition that sums of factors will be expanded whenever this is necessary to guarantee that zero will be recognized. In the special univariate case (i.e., when there is only one indeterminate in expr) each of 'numerator' and 'denominator' will be in expanded form.

Examples:

```
normal( 2/x + y );                       yields   (2+y*x)/x
normal( 3*y*(x-5)^2 / (x^2-25) );        yields   3*y*(x-5)/(x+5)
normal( (sin(1)^2-1) / (sin(1)-1) );     yields   sin(1)+1
normal( 2.3*x+4.5*x );                   yields   6.8x

num := (3*x^2 - 5*x*y)^2 * (x^2 - 2*x*y + y^2);
den := x * (y-x)^3;
normal( num/den );                       yields   x*(3*x-5*y)^2/(y-x)
normal( (sin(x)^3 - 27) / (sin(x) - 3) );  yields   sin(x)^2+3*sin(x)+9
normal( x^2/(1-x) - x/(1-x) );             yields   -x
```

8.1.59. numer (expr)

This function computes the numerator of expr that results from first forming a common denominator for the terms in expr. It calls the *denom* function. (See also denom().)

Examples:

```
numer( 2/x + y );                          yields   2+y*x
numer( 3*y*(x-5)^2 / (x^2-25) );           yields   3*y*(x-5)^2
numer( x^2/(1-x) - x/(1-x) );              yields   x^2-x
```

8.1.60. op (i, expr) or op (i..j, expr) or op (expr)

The purpose of this function is to extract one or more operands from the expression expr. If *op* is called with two arguments and if the first argument evaluates to a nonnegative integer, say i, then the value of the function is the ith operand in expr. General algebraic expressions have operands indexed from 1 to n (for some positive integer n). A function invocation

```
<name> ( <expression sequence> )
```

is considered to have as its 0th operand <name> and the arguments are operands 1 through n (for some integer n). If expr is a series formed by expansion about the point x=a (where x is the name of the indeterminate) then the 0th operand of expr is x-a, the first, third, . . . operands are the coefficients (which may be arbitrary expressions), and the second, fourth, . . . operands are the corresponding exponents (with the exponents ordered from least to greatest). For a more detailed description of the operands corresponding to each of Maple's data types, see section 4.1.

If *op* is called with two arguments and if the first argument evaluates to a range then the value returned is an expression sequence (i.e., a sequence of expressions separated by commas) consisting of the operands specified by the range.

If *op* is called with only one argument, say expr, then the result is equivalent to the result of the invocation

```
op ( 1..nops( expr ), expr ) .
```

For general algebraic expressions, this value is an expression sequence consisting of all of the operands in expr. Note, however, that if expr is one of the structures for which operand 0 is defined (e.g., a series or a function invocation) then the 0th operand will be missing from the expression sequence op(expr).

The special case where expr evaluates to a name must be noted. A name is defined to have exactly one operand, which is the value assigned to the name. If no value has been explicitly assigned to the name then its value is its own name. Note that in the case where a value has been assigned to a name, say x, the *op* function must be called in the form

```
op( 'x' )
```

(or equivalently, op(1, 'x')) if it is desired to see what value was assigned to x; otherwise, if the argument is not quoted then it will be the *value* of x which is passed to the *op* function.

Examples:

```
g := f(x, y, z);
op(0, g);                          yields    f
op(2, g);                          yields    y
op(0..2, g);                       yields    f, x, y
op(g);                             yields    x, y, z

e := [2*x, y+1];
[op(e), z];                        yields    [ 2*x, y+1, z ]

a := (3*sin(x^3) - 2/3*x + y) / (2*x^2 - 1);
op( 2, a );                        yields    (2*x^2-1)^(-1)
op( 1, a );                        yields    3*sin(x^3)-2/3*x+y
op( 2, op(1,") );                  yields    sin(x^3)

w := 3*x^2 - 2*x*y + y^2;
x := 1/2;
op( 'w' );                         yields    3*x^2 - 2*x*y + y^2
op( 'x' );                         yields    1/2
op( w );                           yields    3/4, -y, y^2
op( x );                           yields    1, 2
```

8.1.61. prem (a, b, x, 'm')

This function computes the *pseudo-remainder* of 'a' divided by 'b' with respect to the variable x, where 'a' and 'b' must be polynomials with integer coefficients. Specifically, the value of this function is the unique polynomial r with integer coefficients such that

```
m*a = b*q + r
```

for some polynomial q with integer coefficients, with r = 0 or degree(r,x) < degree(b,x), where the *multiplier* m is defined by

```
m = c ^ (degree(a,x) - degree(b,x) + 1)
```

where c = coeff(b, x, degree(b,x)) -- i.e., c is the leading coefficient in b with respect to x. If the fourth argument is present then it must evaluate to a name and it will be assigned the value of the *multiplier* m defined above.

8.1.62. prevprime (n)

This function returns the largest prime that is less than n. The argument n must evaluate to an integer greater than 2.

8.1.63. print ($expr_1$, $expr_2$, ...)

The effect of this function is to print the values of the expressions appearing as arguments. The global variable 'prettyprint' is checked to determine the fashion in which the expressions are to be printed. If prettyprint has a value of 1, then the expressions are displayed as nicely as possible, perhaps on several lines. If pretty-print has a value of 0, then the expressions are displayed as best as is possible on a single line. The default value for prettyprint is 1. If this function is called with no arguments then the effect is to create a blank line in the output stream.

8.1.64. product (expr, i = m..n)

This function forms the product of the factors obtained by substituting for i in expr the values m, m+1, . . ., n. The second argument must be an equation and its left-hand-side must evaluate to a name, its right-hand-side must evaluate to a range. It is an error if n − m does not evaluate to an integer and it is an error if m > n+1. If m = n+1 then the value of the product is 1.

8.1.65. quo (a, b, x) or quo (a, b, x, r)

This function returns the quotient q of a/b and optionally the remainder r where a, b, q, and r are in Q[x] (polynomials with rational coefficients) and where a = b*q + r. If the argument r is specified, it must evaluate to a name; it is to this name that the remainder is assigned. (Also see rem()).

Example:

```
quo( x^3 + x + 1, x^2 + x + 1, x )        yields     x-1
```

8.1.66. radsimp (expr) or radsimp (expr, 'ratdenom')

This function simplifies expr which may contain radical expressions. If the second argument is present, then it must evaluate to a name and it will be assigned the simplified expression with its denominator rationalized.

```
radsimp( (1 + 2*x + x^2) ^ (1/2) );        yields     1+x
radsimp( (1 + 2*x + x^2) ^ (-1), 'd' );    yields     (1 + 2^(1/2)) ^ (-1)
d;                                         yields     -1+2^(1/2)
```

8.1.67. rand () , rand (n) , or rand (m .. n)

This function returns a random ten digit integer if it was called without arguments; otherwise it returns a procedure that will generate random integers in the range 0..n or in the range m..n if called with an integer or a range argument respectively.

bit := rand(2)	yields	a procedure
bit();	yields	a first random bit
bit();	yields	a second random bit

8.1.68. readlib ('f') or readlib ('f', file$_1$, file$_2$, ..., file$_n$)

Each argument to this function must evaluate to a name. If there is only one argument then the following **read** statement is executed:

read `` . libname . f . `.m`; (or read cat(libname, f, `.m`);)

and the value returned is the value of the argument 'f' after the **read** statement has been executed. If there is more than one argument then the following **read** statements are executed:

read file$_1$; read file$_2$; ...; read file$_n$;

and the value returned is the value of the first argument 'f' after the the **read** statements have been executed. It is an error if 'f' is not assigned a value in the file (or one of the files) being read. For further details, including a complete definition in Maple code, see section 5.7.

8.1.69. rem (a, b, x) or rem (a, b, x, q)

This function returns the remainder r of a/b and optionally the quotient q where a, b, q, and r are in Q[x] (polynomials with rational coefficients) and where a = b*q + r. If the argument q is specified, it must evaluate to a name; it is to this name that the quotient is assigned. (Also see quo()).

Example:

rem(x^3 + x + 1, x^2 + x + 1, x) yields x+2

8.1.70. remember (f (x, y, . . .) = result)

This is a special function to be used in conjunction with the *option remember* facility in procedures (see section 5.3). The argument to this function must be an <equation> and its left-hand-side must take the form of a procedure invocation. The name 'f' must evaluate to a procedure definition in which *option remember* has been specified. The effect of this function is to place an entry in the system table known as the *partial computation table* which associates the specified procedure invocation with the specified 'result'. If there is ever another invocation of this procedure with actual parameters that have the same values as those specified here then the Maple system will immediately retrieve the 'result' from the partial computation table without performing any computation. This function generalizes the *option remember* facility since it may be invoked either from within the body of the procedure 'f' or externally.

Note that this function has special rules for the evaluation of its arguments. The name 'f' will be evaluated to a procedure definition and each of the specified arguments x, y, . . . will be evaluated, but the procedure will not be invoked. The right-hand-side of the equation, 'result', will be evaluated.

8.1.71. RETURN ($expr_1$, $expr_2$, . . .)

This function is a special function whose purpose is to cause an immediate return from a procedure (see section 5.5). Upon execution of this function, control returns to the point where the current procedure was invoked and the value of the procedure invocation is the expression sequence $expr_1$, $expr_2$, ... It is an error if a call to the function RETURN occurs at a point which is not within a procedure definition.

8.1.72. saveonly (filename, '$name_1$', '$name_2$', ... '$name_n$')

This function selectively saves the values of the list of variables specified in the parameter list. If the special name 'notprocs' is given as the second parameter, then ALL names which do not evaluate to a procedure are saved. The value returned by *saveonly* is the NULL expression sequence.

If the filename given ends in '.m', the information is saved in internal format; otherwise, the external format is used.

A call to *saveonly* has the side-effect of creating a file with the name '.temp.m'.

Examples:

After executing the following statements

```
a := 2;  b := x+y;
p1 := proc( x ) x end;
p2 := proc( x ) x^2 end;
saveonly( mydata, 'a', evaln( p.a ));
```

the file 'mydata' contains

```
a := 2;
p2 := proc(x) args[1]**2 end;
```

Subsequently executing

```
saveonly( mydata2, notprocs )
```

places the following information in 'mydata2'

```
a := 2;
b := x+y;
```

8.1.73. seq (expr, i = range)

This function returns the expression sequence produced by substituting each value of the range for i inside the expression expr. The variable The range specified may be any valid increasing range the bounds of which are an integer value apart.

Examples:

```
seq( i^2, i = 1..4)    yields          1, 4, 9, 16
seq( 'a.1', i = 1..3 ) yields          a1, a2, a3
r := x..x+2;
seq( 5, j = r )        yields          5, 5, 5
seq( j^2, j = r )      yields          x^2, (x+1)^2, (x+2)^2
```

8.1.74. sign (expr) , sign (expr, x) , or sign (expr, x, y, ...)

This function computes the *sign* of expr in the sense of the sign of the leading coefficient of expr. The leading coefficient of expr is determined with respect to a set of indeterminants. If sign is called with only one argument, then indets(expr) is used as the set of indeterminants for expr. Otherwise, the second and succeeding arguments are used as the set of indeterminants for expr. (See the function *lcoeff*). Specifically, the definition of the *sign* function when called with one argument is:

if lcoeff(expr) < 0 then −1 else 1 fi

The definition for *sign* called with more than one argument is similar to the one above.

```
sign( x*y-x^2, x, y );      yields    -1
sign( x*y-x^2, y, x );      yields    1
sign( -3.4 );               yields    -1
sign( 0 );                  yields    1
```

8.1.75. solve (eqn, var) or solve ({eqn$_1$, ..., eqn$_k$}, {var$_1$, ..., var$_k$})

This function takes two arguments. The first argument is either a single equation or a set of equations, and correspondingly, the second argument is either a single name which is the variable to be solved for or a set of names which are the variables to be solved for. Whenever an equation is expected in the input arguments, if it is instead an ordinary algebraic expression e then the equation e = 0 is understood. In the case of a single equation and a single variable, it is valid to specify one of the arguments as a set or both arguments as sets. The value of this function is an expression sequence of the solutions, and in the case where the second argument is a set the value is a sequence of solution sets.

As of this writing, the *solve* function is able to solve single equations involving elementary transcendental functions, systems of linear equations, single polynomial equations, and equations requiring the inversion of taylor series.

Examples:

```
solve( cos(x) + y = 9, x );              yields    arccos(9-y)
solve( 2^a + G, a );                     yields    ln(-G)/ln(2)
solve( taylor(arcsin(x)-y, x), x );      yields
        1*y+(-1/6)*y^3+1/120*y^5+O(y^6)
solve( x^2 - 46*x + 529, x );            yields    23,23
solve( 1/2*a*x^2 + b*x + c, x );         yields
        (-b+(b^2-2*a*c)^(1/2))/a,(-b-(b^2-2*a*c)^(1/2))/a

eqn1 := x + 2*y + 3*z + 4*t + 5*u = 6;
eqn2 := 5*x + 5*y + 4*z + 3*t + 2*u = 1;
eqn3 := 3*y + 4*z - 8*t + 2*u = 1;
eqn4 := x + y + z + t + u = 9;
eqn5 := 8*x + 4*z + 3*t + 2*u = 1;
solve( {eqn.(1..5)}, {x,y,z,t,u} );      yields
        {u=8589/110,x=56,z=-13983/110,y=168/5,t=-1736/55}
```

8.1.76. **subs (old$_1$ = new$_1$, ..., old$_k$ = new$_k$, expr)**

This function takes an arbitrary number of arguments and each argument except the last one must be an equation. The value of this function is the expression resulting from applying the substitutions specified by the equations to the last argument, expr. The substitutions are performed sequentially starting with the first argument old$_1$ = new$_1$. Thus, the following two statements are equivalent:

subs(old$_1$ = new$_1$, old$_2$ = new$_2$, expr)
subs(old$_2$ = new$_2$, subs(old$_1$ = new$_1$, expr))

More specifically, the semantics of the function subs(old = new, expr) are that every occurrence in 'expr' of the subexpression 'old' is replaced by the expression 'new'.

Examples:

```
subs( x=1, 3*x*ln(x^3) );              yields    0
subs( a+b = y, (a+b)^(4/3) );          yields    y^(4/3)
subs( a=b+1, b=3, a+b );               yields    7
subs( x^2=9, x^2*y^3 );                yields    9*y^3
```

8.1.77. **subsop (i = newexpr, expr)**

The value of this function is the expression resulting from replacing op(i, expr) by newexpr in expr. The first argument must be an equation and the left-hand-side of the equation must evaluate to a nonnegative integer not greater than nops(expr). In the special case where op(i, expr) does not occur anywhere in expr except as the ith operand, the result of this function will be equivalent to the result of

subs(op(i,expr) = newexpr, expr) .

8.1.78. substring (string, m..n)

This function returns a substring (which is of type 'name') of the first parameter which must of type 'name', i.e., which must be a string. If m and n do not both evaluate to integers, then the function call remains unevaluated. If they do evaluate to integers, then they must both be greater than zero and m must be less than n. Moreover, m must be no greater than the length of the first parameter. If n is greater than the length of the string then *substring* will return the substring from position m o the last position of the string.

8.1.79. sum (expr, i) or sum (expr, i = m..n)

If the second argument is not an equation then this function attempts to compute the 'indefinite summation' of expr with respect to the second argument 'i' which must evaluate to a name. Specifically, if we denote the functional dependency of expr on the variable 'i' by the notation expr(i) then the *indefinite summation* is defined to be an expression g(i), containing the variable 'i', such that

$$g(i+1) - g(i) = \mathrm{expr}(i) .$$

In other words, 'indefinite summation' is the inverse of the forward difference operator.

If the second argument is an equation then its left-hand-side must evaluate to a name 'i' and its right-hand-side must evaluate to a range 'm..n', and this function attempts to compute the definite summation of expr with respect to 'i' with lower limit i = m and upper limit i = n. The limits 'm' and 'n' may evaluate to arbitrary expressions. Note that the definite summation over the range m..n can be obtained from the 'indefinite summation' g(i) as the value:

$$g(n+1) - g(m)$$

and this method is used whenever m − n does not evaluate to an integer or m − n is a very large integer; otherwise, direct summation is performed. Note that the value of the definite summation is zero whenever m = n+1.

If Maple is not successful in performing the summation then a FAIL return occurs, meaning that the value of the function invocation is the unevaluated function invocation.

Examples:

```
sum( i^2, i );                     yields    1/3*i^3-1/2*i^2+1/6*i
expand( subs( i=i+1, ") - " ); yields    i^2

e := (5*i - 3)*(2*i + 9);
sum( e, i = 1..5000 );             yields    417279137500
sum( e, i = 1..n );                yields
             10/3*(n+1)^3 + 29/2*(n+1)^2 - 269/6*n + (-107/6)
expand(");                         yields    10/3*n^3 + 49/2*n^2 - 35/6*n
```

```
sum( i^2 - 2*a*i, i = a..5 );   yields
          55 - 181/6*a - 1/3*a^3 + 1/2*a^2+2*a*(1/2*a^2-1/2*a)
expand(");                      yields    55 - 181/6*a - 1/2*a^2 + 2/3*a^3

sum( x^i, i = 0..n );           yields    x^(n+1)/(x+(-1))-(x+(-1))^(-1)
```

8.1.80. system (s)

This function takes one argument which must of the type name. It transmits this name as a command to be executed by the host system on which Maple runs. The value returned by this function is the return code given by the host system after executing the command. (Also see *escape character.*)

Examples:

```
system( who );
system( `msg mbmonagan "Where's your plot package?"` );
```

8.1.81. table (indexing_function, init_list)

To create a table which is not an array, a call is made to this function. The parameters are an indexing function, and a list for initializations. Both of these are optional and they may appear in either order in the parameter sequence.

The indexing function is given as either a procedure or as a name. If one is not given, then a default of NULL is used. (Actually, that is the *only* way to obtain a NULL indexing function.)

The initializations are given either as a list of equations or as a list of values. (To avoid ambiguity, if a list of values is used, none of the values may itself be an equation.) If a list of equations is given, then for each equation, the left-hand side is used as the index of a component and the right- hand side is used as its value. With a list of values, consecutive integer indices are used starting at 1. The default for initializations is the empty list.

Examples:

```
table( );                                  yields  table([ ])
table([22,33]);                            yields  table([(1)=22,(3)=33])
table([2=22,3=33]);                        yields  table([(3)=33,(2)=22])
table([-9=-99, sin(s)=cos(x)]);            yields  table([(-9)=-99,(sin(s))=cos(x)])
table([(1,2)=12, (2,1)=21]);               yields  table([(2,1)=21,(1,2)=12])
table(symmetric,[(0,1)=a, (c,b,c)=x]);     yields  table([(1,0)=a,(b,c,c)=x])
```

8.1.82. taylor (expr, x = a) or taylor (expr, x = a, n)

The purpose of this function is to compute a Taylor series (more generally, Laurent series) expansion of expr. If the second argument evaluates to an equation then its left-hand-side must evaluate to a name 'x' which will be the variable of expansion and its right-hand-side 'a' will be the point about which the expansion is taken. If the second argument is not an equation then it must be a name 'x', and the effect is the same as if the second argument had been the equation x = 0 . If there is a third argument 'n' then it must evaluate to an integer which specifies the 'truncation degree' to be used. If there is no third argument then the 'truncation degree' is specified by the current value of the global variable *Degree* (which initially has the value 5 in the Maple system). An 'order term' appears in the result of the *taylor* function whenever the result is not known to be exact. (See section 4.1.8 for a description of the series data structure).

Examples:

```
f := (3*x^2 - 5*x) / (x^3 - x + 7);
taylor( exp(f), x=0 );          yields
 1+(-5/7)*x+57/98*x^2+(-509/2058)*x^3+12841/57624*x^4+
                                   (-18971/134456)*x^5+O(x^6)
taylor( f, x=1, 2 );            yields
 (-2/7)+11/49*(x+(-1))+167/343*(x+(-1))^2+O((x+(-1))^3)

e := (x^2 + a*x - 1) / (a+1-x);
taylor( e, x=a, 2 );            yields
 (2*a^2+(-1))+(3*a+2*a^2+(-1))*(x-a)+(3*a+2*a^2)*(x-a)^2+O((x-a)^3)

h := y*exp(y)*sin(x)/x^3 + y*ln(sin(x));
taylor( h, x=0 );               yields
 y*exp(y)*x^(-2)+(-1/6*y*exp(y)+y*ln(x))+(-1/6*y+1/120*y*exp(y))*x^2+O(x^3)

taylor( 1/x + y + x^3, x );     yields     1*x^(-1)+y+1*x^3
taylor( x + x^3 + O(x^2), x );  yields     1*x+O(x^2)

`diff/g` := proc (a,x) `g'`(a) * diff(a,x) end;
`diff/g'` := proc (a,x) `g''`(a) * diff(a,x) end;
Degree := 2;
taylor( sin(g(x)), x=0 );       yields
 sin(g(0))+cos(g(0))*g'(0)*x+(-1/2*sin(g(0))*g'(0)^2+1/2*cos(g(0))*g''(0))*x^2+O(x^
```

User Interface:

New functions can be made known to Maple's *taylor* function by the following mechanism. If the user assigns a procedure to the name `tayl/newfcn` (where 'newfcn' is any name chosen by the user) as in

```
`tayl/newfcn` := proc (expr, x)
                     taylor(expr, x);
                     # Code to compute taylor series for
                     #          newfcn(expr)
                     # from the taylor expansion of expr
                     # about x = 0 using global variable Degree
                     # to specify the 'truncation degree'.
                     . . .
                     end;
```

then the function invocation

```
taylor ( newfcn(expr), x )
```

will cause the function invocation

```
`tayl/newfcn` ( expr, x ) .
```

In the case of a more general invocation of the *taylor* function:

```
taylor ( newfcn(expr), x = a, n )
```

the internal *taylor* function will perform a transformation of the variable x, and will set the global variable ***Degree***, before invoking the `tayl/newfcn` procedure. If `tayl/newfcn` is not assigned then Maple looks for it in the Maple system library at the pathname

```
cat( libname, `tayl/newfcn.m` )
```

and if it is not found then the mechanism described below comes into effect.

A second mechanism for making a new function known to Maple's *taylor* function is to define the derivatives of the function via the user interface for the *diff* function. If Maple is not able to find a definition for the name `tayl/newfcn` then it looks for a definition of the name `diff/newfcn` and, if it is found, then the taylor expansion is generated via differentiation and substitution.

Functions whose series expansions are currently defined in the Maple system library include the elementary functions (all of the circular, inverse circular, hyperbolic, and inverse hyperbolic functions, as well as the functions exp and ln), and the factorial function (which interfaces to the GAMMA function known to *diff*).

8.1.83. trunc (expr)

The value of this function when expr evaluates to an integer, a rational number, or a floating point number is the 'integer part' of expr which would be obtained if expr was expanded in a decimal expansion. For example,

```
trunc( 8/3 );                 yields    2
trunc( -8/3 );                yields    -2
trunc( -2.4 );                yields    -2
```

8.1.84. type (expr, typename) or type (expr, ratfunc, arg)

This is Maple's type-checking function. The value returned is *true* if expr is of type typename and the value returned is *false* otherwise. Except when the second parameter is 'ratfunc', the first form of the function calls shown above is used. The following typenames known to the *type* function correspond to Maple's data types which are described in section 4.1:

`.`, `<`, `<=`, `<>`, `+`, `*`, `**`, `^`, `!`, `and`, `not`, `or`, array, equation (alternatively `=`), range (alternatively `..`), float, function, indexed, integer, list, name, procedure, rational, series, set, table, uneval.

Additionally, the following typenames are known to the *type* function and they are defined in terms of the basic data types as indicated:

algebraic (any of the following types: `.`, `+`, `*`, `^`, `!`, float, function, integer, name, rational, series)

constant (any of the following types: float, integer, rational, or any expression whose operands are all of type constant)

Any object which is of type array is also of type table since arrays are a subclass of tables.

If the second parameter is 'ratfunc', then *type* determines whether the first argument is a rational function of the third argument.

User Interface:

New type-checking procedures can be made known to the *type* function by the following mechanism. If the user assigns a procedure to the name `type/newtype` (where 'newtype' is any name chosen by the user) as in

```
`type/newtype` := proc ( expr, <extra parameters> ) . . . end
```

then the function invocation

```
type ( expr, newtype, <extra parameters> )
```

will cause the function invocation

```
`type/newtype` ( expr, <extra parameters> ) .
```

If `type/newtype` is not assigned then Maple looks for it in the Maple system library at the pathname

```
cat( libname, `type/newtype.m` )
```

and if it is not found then an error occurs.

One additional typename is currently defined in the Maple system library: polynom. It can be used in either of the two forms:

```
type( expr, polynom )
type( expr, polynom, domain )
```

where in the latter case, the extra parameter 'domain' can take any one of five possible forms:

```
typename [ x, y, . . . ]
typename [ { x, y, . . . } ]
[ x, y, . . . ]
{ x, y, . . . }
x
```

The expression expr is checked as a polynomial in the indeterminate(s) specified by 'domain'. In the case of the first two forms of 'domain', there is an additional check that expr, as a polynomial in the specified indeterminates, has coefficients of type 'typename'. In the case where the argument 'domain' is omitted, the implied value of 'domain' is indets(expr). In all cases, the concept of a 'polynomial' is that expr is not of type 'series' and the *degree* of expr in each of the indeterminates is finite (i.e., not equal to the largest word-size negative integer).

8.1.85. unames ( )

This function takes no arguments. It returns an expression sequence consisting of all of the active names in the current Maple session which are *unassigned names*, meaning names which have no value other than their own name. Note that in Maple every 'string' is equivalent to a 'name', so the result of the *unames* function will include every 'string' that has been defined in the session (including file names and error messages). (See also the function *anames*).

8.1.86. whattype (expr)

This function returns the typename of expr. The typename returned for both exponentiation operators '**' and '^' is '**'.

EXAMPLES

```
whattype( x+y )    yields    +
whattype( x*y )    yields    *
```

8.1.87. writeto (filename)

This function has the effect of making the Maple system redirect standard output to the file the name of which is passed as the argument. If the file already exists, its previous data is overwritten. If the file does not exist, it is created. The special name 'terminal' can be used to redirect output to the user's terminal device.

8.2. Miscellaneous Library Functions

The following functions reside in the Maple system library but they are not automatically loaded. In other words, the names of these functions are not initially defined in the Maple system. If the user wishes to load one of these functions, named 'fname', then he may use the **read** statement:

```
read cat( libname, `fname.m` )
```

or he may use the *readlib* function to initiate an 'automatic loading' facility by specifying the assignment:

```
fname := 'readlib('fname')' .
```

8.2.1. bernoulli(n)

This function computes the nth Bernoulli number. The argument n must evaluate to an integer value.

8.2.2. binomial(n, r)

This function computes the binomial coefficient n!/(r!*(n−r)!). The two arguments must have integer values.

8.2.3. cfrac (f) or cfrac (f, maxit)

This function prints the sequence of quotients and convergents to f. If maxit is given as a second argument, then no more than maxit values are printed.

8.2.4. content (a, varlist)

This function returns the contents of the multivariate integer coefficient polynomial a with respect to the variables in varlist. The argument varlist may be a list, set, or expression sequence of variables and it may be empty.

Examples:

content(w^2*x^3 + w^3*x^4*y^5*z^6, [y, z])	yields	w^2*x^3
content(3*x*y + 6*x^2*y^2, x)	yields	3*y
content(4 + 2*x, {x})	yields	2
content(3 + 3*x)	yields	3+3*x

8.2.5. convergs (a, b, n) or convergs(a, b)

This function finds and prints the convergents of a continued fraction. The continued fraction

$$a_1 + \frac{b_2}{a_2} + \frac{b_3}{c_3} + \frac{b_4}{a_4} + \cdots$$

is entered as: convergs(a, b, n); where a and b are either lists ([a_1, a_2, ..], [1, b_2, ..]) or functions which compute the respective coefficients. If all the b coefficients are 1, b can be omitted. n is an optional parameter which indicates the number of convergents to compute.

8.2.6. E_ML (f, x, n)

E_ML(f, x, n) computes an n^{th} degree Euler-Maclaurin summation formula of f (an expression in x). In general, E_ML(f, x, n) is an asymptotic approximation of sum(f, x).

8.2.7. expandoff (<fcn $name_1$>, <fcn $name_2$ >, ... <fcn $name_n$>)

This function can be used to supress Maple's knowledge of how to expand the functions listed in the parameter list. If the parameter list is empty, then knowledge of all functions is supressed. The function *expandoff* returns the NULL expression sequence as its result. (See also *expand* and *expandon.*)

Examples:

expandoff(exp, ln)	yields	NULL
expand(exp(a+b))	yields	exp(a+b)
expandon()	yields	NULL
expand(exp(c+d))	yields	exp(c)*exp(d)
expand(exp(a+b))	yields	exp(a+b)
gc()	yields	NULL
expand(exp(a+b))	yields	exp(a)*exp(b)

WARNING: The *expand* function uses the *remember* option which is why the second call of expand(exp(a+b)) did not expand. Garbage collection (c.f. *gc*) clears the partial computation table; subsequently the third call of expand(exp(a+b)) does expand its argument.

8.2.8. expandon (<fcn $name_1$>, <fcn $name_2$ >, ... <fcn $name_n$>)

This function can be used to reassert Maple's knowledge of how to expand the functions listed in the parameter list. If the parameter list is empty, then knowledge of all functions is reasserted. The function *expandon* returns the NULL expression sequence as its result. (See also *expand* and *expandoff.*)

8.2.9. invfunct (fname)

This function returns the name of the inverse of the function fname.

Examples:

invfunct(cosh)	yields	arccosh
invfunct(exp)	yields	ln
invfunct(arctan)	yields	tan

8.2.10. mgcd (a, b, p)

This function returns the modular gcd of the univariate polynomials a and b with respect to the modulus p.

Example:

```
mgcd( x+2, x+3, 7 )          yields     1
```

8.2.11. mgcdex(a, b, p) or mgcdex(a, b, p, k)"

This function applies the Extended Euclidean algorithm in $Z/p^k[x]$. The arguments a and b are interpreted as polynomials in the domain $Z/p^k[x]$ and p must evaluate into a prime integer. If the fourth parameter k is not given in the argument list, then the default value of 1 is used for k. This function produces a list [s, t, g] with the following properties:

1) g = GCD(a mod p, b mod p) in Zp[x].

2) If k=1, then s and t are polynomials in Zp[x] such that

$$s*a + t*b = g \pmod p$$

3) If k>1 and g=1, then s and t are polynomials in $Z/p^k[x]$ such that

$$s*a + t*b = 1 \pmod{p^k}$$

4) It is an error if k>1 and g≠1.

If k=1, then this is the standard extended Euclidean algorithm in the Euclidean domain Zp[x] for a prime integer p. If k>1, then first the standard extended Euclidean algorithm is applied to a mod p and b mod p in Zp[x] and if the gcd is 1 in Zp[x] then p-adic lifting is applied to lift the solution to the domain $Z/p^k[x]$.

8.2.12. mpower (x, n, p)

This function returns imodp(x^n, p). (Also see *imodp*.)

Examples:

```
mpower( 2, 10, 1000 )        yields     24
mpower( 3, 4, 11 )           yields     4
```

8.2.13. orthog.p

All the orthogonal polynomials in this package are generated by their recurrences. This seems to be the most efficient procedure. It is better than the generating function, even if you want all of the polynomials from 1 to n.

H(n, x) generates the nth orthogonal Hermite polynomial.

P(n, x) generates the nth orthogonal Legendre polynomial.

T(n, x) generates the nth orthogonal Tschebysheff polynomial of the first kind.

U(n, x) generates the nth orthogonal Tschebysheff polynomial of the second kind.

8.2.14. psqrt (a)

This function returns a square root of the multivariate integer coefficient polynomial a, if a is a perfect square; otherwise it returns the name @NOSQRT.

Examples

```
psqrt(9);                       yields   3
psqrt( x^2 + 2*x + 1 );         yields   x+1
psqrt( x^2 + 2*x*y +y^2 );      yields   x+y
psqrt( x+y );                   yields   @NOSQRT
```

8.2.15. randlcm ()

This function returns random integers after it has been initialized by the call

randlcm_init (m, a, c, X);

where the linear congruential method is used with

X[n+1] = (a*X[n] + c) mod m .

Example:

```
randlcm_int( 10^10, 4219755981, 9893258573, 4806771896 )   yields   NULL
randlcm()                                                  yields   7341968549
randlcm()                                                  yields   6849900142
```

9. MISCELLANEOUS FACILITIES

9.1. Debugging Facilities

The current version of Maple does not have the sophisticated syntax error messages that we envision for Maple in the future. The best mode of operation for detecting syntax errors in procedure definitions is to develop the procedure definition into a file (using a text editor external to Maple) and then to use the **read** statement to read the file into Maple. In this mode, when a syntax error is encountered the corresponding line number in the file is displayed with the syntax error message.

One name whose value determines the amount of information displayed to the user during execution of a Maple session is *yydebug*. The default value for yydebug is 0. If the user assigns the value 1 to yydebug as in the statement

```
yydebug := 1;
```

then the system displays a very large amount of information which is a trace of the Maple session from the basic system viewpoint.

A more useful facility from the user viewpoint is the *printlevel* facility. The default value for printlevel is 1. Any integer may be assigned to the name printlevel and, in general, higher values of printlevel cause more information to be displayed. Negative values indicate that no information is to be displayed. More specifically, there are *levels* of statements recognized within a particular procedure (or in the main session) determined by the nesting of selection and/or repetition statements. If the user assigns

```
printlevel := 0;
```

then the following statements within the main session

```
b :=2;
for i to 5 do a.i := b^i od;
```

will generate the printout b := 2 after execution of the first statement and there will be no printout caused by the for-statement (the value of the for-statement is null). If the user assigns

```
printlevel := 1;
```

before the above statements are executed (or equivalently, if no assignment to printlevel has been made) then each statement within the for-statement will be displayed as it is executed (in the same manner as if these statements appeared sequentially in the 'mainstream'), yielding the following printouts for the above statements:

```
b := 2
a1 := 2
a2 := 4
a3 := 8
a4 := 16
a5 := 32
```

The statement b := 2 is considered to be at level 0 while the other assignment statements in this example are at level 1 because they are nested to one level in a repetition statement. If statements are nested to level *i* then the value of printlevel must be *i* if the user wishes to see the results of these statements displayed.

More generally, statements are nested to various levels by the nesting of procedures. The Maple system decrements the value of printlevel by 2 upon each entry into a procedure and increments it by 2 upon exit, so that normally (with printlevel = 1) there is no information displayed from statements within a procedure. If the user assigns

```
printlevel := 2;
```

in the main session then statements within procedures called directly from the main session (but not nested statements) will be displayed as they are executed, because the effective value of printlevel within the procedure is 0. If the user assigns

```
printlevel := 3;
```

in the main session then, in addition, statements nested to one level of selection and/or repetition statements in the procedure will be displayed because the effective value of printlevel within the procedure is 1. Alternatively, the user may explicitly set the value of printlevel within the procedure for which the information is desired.

It is often useful for debugging purposes to set a high value of printlevel in the main session if information is desired from within procedures to various levels of nesting. When the effective value of printlevel upon entry to a procedure is 3 or greater, the printout will display the entry point and exit point for that procedure as well as the values of the arguments at the entry point. It is not uncommon to use a debug setting such as

```
printlevel := 1000;
```

in which case entry and exit points and statements will be displayed for procedures up to 500 levels deep. For more selective debugging information, the value of printlevel should be assigned within specific procedures.

A program called *profile* is available for processing the output produced by Maple with a high setting of printlevel. This program is separate from the Maple system and is available under the same directory where the Maple system resides. It is used in the form:

```
profile <outfile
```

where 'outfile' is a file containing Maple output produced with a high setting of printlevel (in particular, entry and exit points of Maple functions must be displayed). The output from the *profile* program is a table showing the name of each Maple procedure (including the Main Routine) that was entered, the number of entries to the procedure, and the number of lines (also the number of characters) of output in 'outfile' originating from the procedure. This information can be useful to pinpoint 'bottleneck' procedures which should be candidates for efficiency improvements.

9.2. Monitoring Space and Time

As execution proceeds in a Maple session the user will see lines displayed in the form "words used n" for integer values n. This information indicates the number of *words* of memory that have been requested up to that point in the execution of the session. This information is also displayed at the end of a session when the **quit** statement is executed, where the phrase "Final 'words used'=n" is displayed. It should be noted that this measure of memory usage is not directly related to the actual memory requirements of the Maple session at any point, but rather is a cumulative count of all memory requests made to the internal Maple memory manager during execution of the session. Typically, a significant proportion of the 'words used' at any point may have been re-allocations of actual memory that was previously used and then released to Maple's memory manager.

A second measure of memory requirements is displayed at the end of a session when the **quit** statement is executed, in the form of the phrase "storage=n" for some integer n. This measures the memory space actually occupied by the Maple system plus the data area, and the unit of measurement is the 'natural' unit of memory for the particular host system (e.g., *bytes* on the Vax machine and *words* on the Honeywell machine). Note that Maple's internal memory manager requests 'storage' from the host system in large chunks and then allocates it as needed, so that the final "storage=n" measure typically includes a significant number of memory units that were never actually required by the Maple session.

Monitoring timing information for a Maple session can be accomplished by using the timing facilities of the host system. Typically there is a **time** command on the host system and it is often convenient to use this command along with the host system's facilities for re-direction of input and output. For example, if *infile* denotes the pathname of a file containing the Maple session to be timed and if *outfile* denotes the pathname of the file where the Maple output is to be directed then the UNIX command

```
time /u/maple/bin/maple <infile >outfile
```

or the Honeywell TSS command

```
time : maple/maple <infile >outfile
```

will cause the Maple session in *infile* to be executed, with output into *outfile*, and after completion the host system will display the timing information for that session. Of course, the **time** command may also be used without necessarily using re-direction of input and/or output.

9.3. Session Initialization

When a Maple session is begun, the Maple system first searches for an initialization file before it starts receiving any input. Any Maple statements may be placed in this file and these statements are executed before any statements from the input file. If there is no initialization file, then the Maple system begins directly by reading statements from the input file. If there is an initialization file, the printlevel variable is set to -1 before the initialization statements are executed so that initialization will proceed silently. We recommend that the last statement of the initialization file be "printlevel := 1:" to reset the variable. The silent statement terminator (:) is used so that the result of the assignment statement is not printed either.

On UNIX systems, the initialization file must be placed in a user's home directory under the name ".mapleinit". On the Honeywell system, the initialization file must be placed in the file "userid/_sysfiles/maple" where "userid" is the userid of the user.

The use of the load option (c.f. section 9.4) on the command line invoking Maple will prevent any initialization file from being read.

9.4. Other Facilities

Escape Character

The character ! when it appears as the first character in a line is treated as an 'escape to host' operator. This allows one to execute any command in the host system from within a Maple session. The line need not terminate with a Maple statement separator (':' or ';') before the carriage-return character terminates the line.

Garbage Collection

Maple's automatic garbage collection facility has not been implemented at the time of this writing. However, there is a function *gc()* which can be invoked by the user. This has the effect of deleting all data structures to which no references are made and also of deleting the partial computation table. The function returns the NULL expression sequence and it also prints a message showing three values:

```
words returned      words available      words allocated
```

The first value is the number of words that were released during garbage collection. The second one is the total number of free words available. The third value is the total amount of memory that has been allocated by Maple so far.

Wrap Program

There is a program called *wrap* which will insert <newline> characters at appropriate intervals in files containing very long lines of output. (Note that it is not uncommon for Maple to produce very long expressions in its output). This program is necessary on some host systems as a pre-processor before the host system's editor will accept the file for editing. The *wrap* program is separate from the Maple system and is available under the same directory where the Maple system resides (on those host systems where it is required). It is used in one of the following two forms:

```
wrap <file1 >file2
wrap n <file1 >file2
```

where 'file1' is the original Maple output file, 'file2' is the file into which the 'wrapped' output will be deposited, and 'n' (if present) specifies the maximum number of characters to be allowed in a line before a <newline> character. (The default value of 'n' is 240). The *wrap* program uses some knowledge about mathematical expressions in attempting to insert the <newline> characters at 'natural' break points (when possible), rather than breaking after exactly 'n' characters.

Load Option

Maple has a *load* option which must be used whenever functions are being loaded into the Maple system library, and which should be used whenever an internal-format ('.m') file is created by a user. This option is activated by specifying '−l' immediately following the 'maple' command. For example, on the Vax UNIX system a typical command for loading a library function named 'f' would be

```
/u/maple/bin/maple -l </u/maple/lib/src/f
```

where the source file for 'f' should end with the statements

```
save `` . libname . `f.m`;
quit
```

In Maple's normal mode of operation (without the *load* option), when an internal-format **save** is done the Maple <name>s which correspond to automatically-loaded library functions (*readlib*-defined functions) are not saved. Therefore, in this normal mode it is impossible to update the '.m' files which define the Maple library functions. The effect of the *load* option is to initiate a Maple system in which none of the library function names is initially defined (and the global variable names *printlevel*, *Digits*, and *Degree* are also undefined). It follows that such a Maple system is of limited value for ordinary use; its sole purpose is for loading '.m' files.

It is recommended that every user should use the *load* option when creating a '.m' file. Otherwise, the *readlib* definition of each Maple library function which is referenced, and also the current values of any of the above-mentioned global variables which are referenced, will be stored in the user's file. This may lead to several undesirable effects: the value of the global variables will be 'mysteriously' redefined when the user's '.m' file is loaded; there may be unwanted re-loading of library functions (which not only is costly but also destroys previously-remembered values for functions with option remember); and, even more seriously, there may arise a circular loop loading and re-loading files!

10. APPENDIX A

UNITS USED BY METRIC CONVERSION FUNCTION

The following units are the ones which are known to the *convert* function when it is called upon to convert from a non-metric expression to a metric one.

acre
acres
bu
bushel
bushels
chain
chains
cm
cord
cords
feet
foot
ft
furlong
furlongs
gal
gallon
gallons
Gals
gill
gills
gr
hr
in
inch
inches
kg
km
Lb
lbs
light_year
light_years
mi
Mile
miles
MPG
MPH
ounce
Ounces
oz
Ozs
pint
pints
pole
poles
pound
pounds
quart
quarts
yard
yards
yd
yds

11. APPENDIX B

EXAMPLES OF TABLES

This appendix contains edited versions of the procedures `table/initbds` and `table/initvals` from the Maple system library. These are given as model programs upon which a user may wish to base his own procedures.

```
# This function is called to deduce the bounds when an array is being
#       created and no bounds are given but some initializing values are.
# Input        -- the list of initializations supplied in the call to 'array'.
# Output       -- a sequence of zero or more integer ranges.
#
# The action taken depends on whether we have values or equations:
# --    If only values, then give bds for a 1-dimensional array.
# --    If only equations, then deduce the dimensions from the LHS's.
#       (All LHS's must have the same number of components.)
# --    Otherwise, the input is erroneous.

`table/initbds` :=
proc(init_list)
        local i, j, rank, bds, lo, hi, err_msg;
        err_msg := `improper array initializations`;

        # determine whether initializations are equations or values
        {op(map(type, init_list, `=`))};

        if " = {false} then        # list of values
                1..nops(init_list)

        elif " = {true} then       # list of equations
                # verify that all indices have same number of components
                {op(map( proc(eqn) nops([op(1,eqn)]) end, init_list))};
                if nops( " ) <> 1 then ERROR(err_msg) fi;

                # find smallest "box" containing the indices
                rank := op( " );
                bds  := NULL;
                for i to rank do
                        lo := op(i,[op(1,op(1,init_list))]);      hi := lo;
                        for j to nops(init_list) do
                                op(i,[op(1,op(j,init_list))]);
                                if not type(", integer) then ERROR(err_msg)
                                elif " < lo then lo := "
                                elif " > hi then hi := "
                                fi
                        od;
                        bds := bds, lo..hi
                od;
                bds

        else    ERROR(err_msg)      # list has some equations AND some values
        fi
end;
```

```
# This procedure is used to install the initial values in a table.
# The first parameter is the table, the second is the list of
# initializations from the call to 'array' or 'table'.
# This function is not called if the list of values is null.
#
# The action depends on whether we have values or equations:
# --   If only values, install with integer indices.
# --   If only equations, install with the LHS's as indices.
# --   Otherwise, the input is erroneous.

`table/initvals` :=
proc(tbl, init_list)
        local i, `lo-1`, err_msg;
        err_msg := `improper initializations for table or array`;

        {op(map(type,init_list, `=`))};

        if " = {true} then
                for i to nops(init_list) do
                        tbl[op(1,op(i,init_list))] := op(2,op(i,init_list))
                od
        elif " = {false} then
                if type(tbl,array) then
                        if nops([op(2,tbl)])<>1 then ERROR(err_msg) fi;
                        `lo-1` := op(1,op(2,tbl)) - 1
                else
                        `lo-1` := 0
                fi;
                for i to nops(init_list) do tbl[`lo-1`+i] := op(i,init_list) od
        else
                ERROR(err_msg)
        fi
end;
```

12. REFERENCES

Bou71a. S.R. Bourne and J.R. Horton, ``The Design of the Cambridge Algebra System,'' pp. 134-143 in *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, ed. S.R. Petrick, Special Interest Group on Symbolic and Algebraic Manipulation, Association for Computing Machinery (1971).

Hal71a. Andrew D. Hall, Jr., ``The ALTRAN System for Rational Function Manipulation - A Survey,'' in *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, ed. S.R. Petrick, Special Interest Group on Symbolic and Algebraic Manipulation, Association for Computing Machinery (1971).

Hea71a. Anthony C. Hearn, ``Reduce 2: A System and Language for Algebraic Manipulation,'' pp. 115-127 in *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, ed. S.R. Petrick, Special Interest Group on Symbolic and Algebraic Manipulation, Association for Computing Machinery (1971).

Joh83a. Howard Johnson and the 28 flavors, *Margay reference manual*, (Margay is a type of cat native to South America.), 1983.

Mar71a. W.A. Martin and R.J. Fateman, ``The MACSYMA System,'' pp. 59-75 in *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, ed. S.R. Petrick, Special Interest Group on Symbolic and Algebraic Manipulation, Association for Computing Machinery (1971).

Mera. Meg Merrill, *Know Your Ocelots and Margays*, The Pet Library Ltd, New York.

INDEX