MAPLE User's Manual

Keith O. Geddes
Gaston H. Gonnet

CS-81-25

July, 1981

M A P L E

User's Manual

Keith O. Geddes

Gaston H. Gonnet

July 1981

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 document is a working document for the project.

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 all current symbolic systems were made ten to fifteen years ago, it would be wise to design a new system from scratch.

An important property of the Maple system is its modular design. 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. On top of this basic system are successive levels of 'function packages' each of which adds more facilities to the system as may be required by a particular user, such as polynomial factorization or the Risch integration algorithm. The systems implementation language for Maple is primarily C running under the UNIX operating system. Maple is also available in Honeywell TSS where B is used as the implementation language, and work is progressing to bring Maple up in C under VM/CMS. 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 and that the appropriate environment for such use will be a dedicated microprocessor with one or more megabytes of main memory. Current plans are to use a Motorola 68000-based microsystem for the first implementation on dedicated hardware.

TABLE OF CONTENTS

Preface

1. Introduction 1

2. Language Elements 2
   2.1. Character Set 2
   2.2. Tokens 3
   2.3. Blanks, Lines, and Comments 6
   2.4. Files 6

3. Statements and Expressions 8
   3.1. Types of Statements 8
      3.1.1. Assignment Statement 8
      3.1.2. Expression 8
      3.1.3. Read Statement 8
      3.1.4. Save Statement 8
      3.1.5. Solve Statement 8
      3.1.6. Selection Statement 9
      3.1.7. Repetition Statement 9
      3.1.8. Print Statement 9
      3.1.9. Local Statement 9
      3.1.10. Stop Statement 10
   3.2. Expressions 10
   3.3. Sample Maple Session 13

4. Data Types and Formal Syntax 17
   4.1. Data Types 17
      4.1.1. Name 17
      4.1.2. Rational number 17
      4.1.3. General algebraic expression 17
      4.1.4. Polynomial canonical form 17
      4.1.5. Boolean expression 17
      4.1.6. Statement sequence 17
   4.2. Formal Syntax 18

5. Functions 20
   5.1. Boolean Functions 20
      5.1.1. has ? <exp1>, <exp2> ? 20
      5.1.2. assigned ? <name> ? 20
      5.1.3. name ? <exp> ? 21
      5.1.4. constant ? <exp> ? 21
      5.1.5. integer ? <exp> ? 21

5.1.6. addition ? <exp> ? 21
5.1.7. multiplication ? <exp> ? 21
5.1.8. power ? <exp> ? 21
5.2. Algebraic Functions 21
5.2.1. op(<range>, <exp>) 21
5.2.2. subs(<equation>, <exp>) 23
5.2.3. taylor(<exp1>, <equation>, <exp2>) 23
5.2.4. diff(<exp>, <nameseq>) 23
5.2.5. int(<exp>, <name>) 24
5.2.6. int(<exp>, <name> = <range>) 24
5.2.7. sum(<exp>, <name>) 24
5.2.8. sum(<exp>, <name> = <range>) 24
5.2.9. simplify(<exp>, <equatseq>) 25
5.2.10. sin(<exp>) 25
5.2.11. cos(<exp>) 25
5.2.12. exp(<exp>) 25
5.2.13. ln(<exp>) 25
5.2.14. trunc(<exp>) 25
5.2.15. igcd(<exp1>, <exp2>) 26
5.2.16. expand(<exp1>, <exp2>) 26
5.2.17. coeff(<exp1>, <exp2>, <exp3>) 26
5.2.18. degree(<exp1>, <exp2>) 26
5.2.19. nops(<exp>) 27
5.2.20. indets(<exp>) 28
5.3. User-defined Functions 28

6. Internal Representation and Manipulation 35
6.1. Internal Representations ....
6.2. Rules of Evaluation ....
6.3. Automatic Simplification ....
6.4. Common Expression Handling ....

7. Miscellaneous Facilities 36
7.1. Debugging Facilities 36
7.2. Echoing and Timing 37
7.3. Facilities for Names 38
7.4. Other Facilities 39

8. The Maple Library 40

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 b+1 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 to real’ which would yield the floating-point value of sin(1) to the accuracy of the floating-point number system used).

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 24 *special characters* are

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

leaving the following ASCII characters as yet unused:

$	dollar	|	vertical bar

&	ampersand	%	percent
~	tilde	@	at-sign
\	back slash	^	circumflex
`	grave accent		

2.2. Tokens

The tokens consist of keywords, reserved function names, operators, strings, constants, and punctuation marks.

The *keywords* are the following reserved words which are used in forming statements:

by	od
do	print
done	quit
elif	read
else	save
end	solve
fi	stop
for	then
from	to
if	while
local	

The *reserved function names* are the following reserved words:

diff	subs
int	sum
op	taylor
simplify	

The *operators* consist of the *binary* operators

+	addition; set union
−	subtraction; set difference
*	multiplication; set intersection
/	division
**	exponentiation
<	less than
<=	less than or equal

=	equal
>	greater than
>=	greater than or equal
and	logical and
or	logical or
:=	assignment
.	concatenation
=	equation symbol
..	ellipsis (more generally, ...*)

the *unary* operators

+	unary plus (prefix)
−	unary minus (prefix)
!	factorial (postfix)
not	logical not (prefix)

and the *nullary* operators

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

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

A *string* is a letter followed by zero or more letters, digits and underscores, with a maximum length of 43 characters. A string is a valid name (e.g. a variable name or a function name) but we shall see that a name may also involve the concatenation operator.

The *constants* are integers, rational numbers, and reals. An integer is either a natural integer (i.e. an unsigned integer) or a signed integer. A rational number is of the form <integer>/<natural integer>. The length of integers (and hence rational numbers) is restricted only by the address space of the host processor. A real is an (optionally signed) sequence of digits containing a period (decimal point). [Note: Reals are not yet implemented in Maple.]

The *punctuation marks* are

;	semicolon	?	question mark
,	comma	[	left bracket
'	single quote	]	right bracket
(	left parenthesis	{	left brace

)	right parenthesis	}	right brace

The semicolon is used to separate statements and the comma is used to separate expressions in a function call and in specifying a set. The single quote is used to specify that an expression, a statement, or a sequence of statements is to be ***unevaluated,*** by enclosing the expression or statement sequence in a pair of single quotes. For example, the statements

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

cause the value b+1 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;
```

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

```
g := 'g';
```

is executed, then even if g 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.

The remaining seven punctuation marks are used for various kinds of bracketing. The left and right parentheses have their familiar use in grouping terms in an expression. These parenthesis also have a familiar use in grouping parameters in a function call for the case of *algebraic* (non-boolean) *functions*. For the case of *boolean functions* the question mark is used to group the parameters. For example, a call to the built-in algebraic function igcd (which computes the greatest common divisor of two integers) can be contrasted with a call to the built-in boolean function integer (which has the value true if its argument is an integer and has the value false otherwise) in the following sample statement:

```
if  integer ?a?  and  integer ?b?  then  g := igcd(a,b)  fi;
```

The left and right brackets are used to form *subscripted names* (as in the use of arrays). For example, the following are valid distinct names in Maple:

```
a[3,1,5];  a[-4];  a[1];
i := 2;  j := 1;  a[i, j].
```

(Note that arrays are not declared in Maple and the above may be viewed simply as a facility for forming new names by the concatenation of symbols, where the subscripts may be arbitrary expressions which evaluate to integers. Note also that a specific string, such as a above, may be used with one subscript and with several subscripts in the same session, although it is probably bad programming practice to do so). The left and right braces are used to form *sets* in Maple. An example of a set would be

```
a := {x,y,z} .
```

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.

Input to the Maple system consists of a stream (sequence) of statements separated by semicolons. 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 is complete if it is a syntactically valid statement followed by a semicolon. When a line is entered, the system evaluates (executes) the statements (if any) which have been completed on that line.

When a sharp (#) is encountered, all subsequent characters on the line are considered to be a *comment*. The comment is echoed by the system.

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. All files which are in Maple internal format are distinguished by the fact that the file name ends with the characters '.m' . Some valid file names for files in user format are:

```
temp
/mycat/file1
```

and some valid file names for files in Maple internal format are:

```
temp.m
/lib/integration/risch.m
```

A file is created using system facilities external to Maple. 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. The contents of a file in Maple internal format are written into the file from Maple by using the 'save' statement. 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 Maple internal format. When one of these functions is encountered by the Maple system, the corresponding file is automatically read in to the Maple session.

3. STATEMENTS AND EXPRESSIONS

3.1. Types of Statements

There are ten 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 result of this statement is that the expression is evaluated.

3.1.3. Read Statement

The statement

```
read <filename>
```

causes the file named <filename> to be read in to the Maple session. The <filename> may be one of two types as discussed in section 2.4.

3.1.4. Save Statement

The statement

```
save <filename>
```

causes the current Maple environment to be written into the file named <filename>. If <filename> 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. Solve Statement

The syntax of this statement is

```
solve(<equation>, <name>)
```

where <equation> has the form

```
<expression> = <expression>
```

(or simply <expression> in which case <expression> = 0 is understood). The result is to solve (if possible) the <equation> for the variable <name>.

3.1.6. Selection Statement

The selection statement takes one of the following three general forms. Here <bool> stands for a boolean expression and <statseq> stands for a sequence of statements.

```
if <bool> then <statseq> fi
if <bool> then <statseq> else <statseq> fi
if <bool> then <statseq> elif <bool> then . . . etc.
```

The sequence of statements in the branch selected (if any) is executed and the value of the selection statement is the value of the last statement executed.

3.1.7. Repetition Statement

The syntax of the repetition statement is as follows, where <bool> and <statseq> are as in 3.1.6 and where <exp> stands for an algebraic expression.

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

where any of 'for part', 'from part', 'by part', 'to part', or 'while part' may be omitted. The sequence of statements in <statseq> is executed zero or more times and the value of the repetition statement is the value of the last statement executed. The 'for 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 'from part' and/or 'by part' are omitted then the default values 'from 1' and/or 'by 1' are used. If 'to part' and/or 'while part' are present then the corresponding tests for termination are checked at the beginning of each iteration, and if neither is present then an infinite loop will occur.

3.1.8. Print Statement

The syntax of this statement is

```
print( <nameseq> )
```

where <nameseq> stands for a sequence of names. The result is that the value associated with each name in <nameseq> is printed out.

3.1.9. Local Statement

The syntax of this statement is

```
local( <nameseq> )
```

where <nameseq> is a sequence of simple names not involving the concatenation operator. The effect of this statement is to make the names in <nameseq> to be 'local from here to the end of the current statement sequence'. In other words, this statement can be viewed as causing a syntactic renaming of every occurrence of the specified names from the local statement to the end of the statement sequence in which the local statement appears. A statement sequence must not contain more than one local statement. (The primary use of this statement will be in a function definition).

3.1.10. Stop Statement

The syntax of the stop statement is any one of the following four forms:

```
quit
done
stop
end
```

The result of this statement is to terminate the Maple session and return the user to the system level from which Maple was entered.

3.2. Expressions

The simplest expressions are called *primaries* and the simplest instance of a primary is a <natural integer>. A <name> is also a primary and it has a value which may be any expression or, if no value has been assigned to it, then the <name> simply stands for itself. A <name> may be simply a <string>, which was previously defined to be a letter followed by zero or more letters, digits, and underscores (with a maximum length of 43 characters). Note that lower case letters and upper case letters are distinct, so that the names

```
g
G
new_term
New_Term
```

are all distinct.

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 more examples of valid <name>'s are:

```
This_is_an_extremely_long_name
x13A
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).

The concatenation operator may be used in another construct to form a *name sequence*, as follows:

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

where <range> takes the form <expression> .. <expression>. For example, to print out the values of a1, a2, . . . , a10 one may use the statement

```
print(a.(1..10)).
```

This is equivalent to the statement print(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10).

Yet another construct for forming a <name> is the *array notation* which takes the form

```
<name> [ <expression sequence> ] .
```

This construct is similar in principle to the concatenation construct. For example, the name a[2,5] is simply a name which the system views as a concatenation of the symbol 'a' with the symbols '[', '2', ',', '5', and ']'. Therefore for the case of single subscripts, the constructs

```
a.i , for i = 1, 2, . . . , n
a[i] , for i = 1, 2, . . . , n
```

are equally general. However when using two or more subscripts the array notation is more powerful. For example if i = 1, j = 27, m = 12, and n = 7 then

```
a[i,j];  a[m,n];
```

evaluate to the distinct names a[1,27] and a[12,7] while

```
a.i.j;  a.m.n;
```

both evaluate to the single name a127. Finally note that since a[1,27], for example, is a valid name it follows that the construct

```
a[1,27][5]
```

is also valid. In general, the array notation may be freely combined with itself and with the concatenation construct as desired.

A sequence of statements enclosed in single quotes is called an *unevaluated statement sequence*. The three examples given in section 2.2 showing the use of single quotes:

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

are examples where the unevaluated statement sequence is a single expression. Here the name x may be later used as an expression and the effect of evaluating x is to strip off the quotes. The third case here is a special case which means that the name g no longer has any value associated with it (g now stands for itself). A different situation arises if the unevaluated statement sequence is not an expression (e.g. it may be an assignment statement, or a sequence of statements, etc.), such as in

```
f := 'a := 1';
```

or

```
g := 't := a;  a := b;  b := t;  a + b';
```

In these examples, the names f and g may not be later used as ordinary expressions because they do not evaluate to valid expressions. These are instances of *function definitions* and they are invoked by using the syntax

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

which is another instance of a primary. For example with f and g defined as above, the effect of the statement f() is that the statement a := 1 is executed, while the effect of the statements

```
a := 7;  b := 3;  g();
```

is that the value 10 is returned for the statement g() and the value of a is now 3, the value of b is now 7. (A discussion of functions with parameters and local variables will be postponed until section 5).

A primary may be formed from any arbitrary expression by enclosing the expression in parentheses. A different type of primary is a *set* which takes the form

```
{ <expression sequence> } .
```

Other expressions are formed using 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 ! following any primary denotes the factorial function of its operand.

All of the expressions discussed so far are classified as primaries. More general expressions are formed from the primaries by using the *algebraic operators* +, −, *, /, and **. + and − may be used as unary (prefix) operators and all five of these operators may be used as binary operators. The usual rules of precedence apply, with ** having the highest binding strength, then * and /, followed by + and −. When the order of evaluation is not determined by the rules of precedence then the order of evaluation is left-to-right. The operators +, −, and * may also be used as set operators in which case they denote set union, set difference, and set intersection, respectively.

A *boolean expression* can appear only as the conditional in a selection statement or in the while-part of a repetition statement. A boolean expression can be formed from algebraic expressions by using the *relational operators* <, <=, =, >, >=. All of these operators except = are invalid if the difference of their operands does not evaluate to a constant. In the case of the relation

```
op1 = op2
```

the operands are arbitrary algebraic expressions. The expression op1 − op2 is formed and 'simplified' in the current 'environment' and if the result is zero then the boolean expression is 'true', otherwise it is 'false'. Another form of boolean expression is a *boolean function* which is invoked using the syntax

```
<name> ? <expression sequence> ?
```

(i.e. the syntax is similar to the invocation of an algebraic function except that question marks are used in place of left and right parentheses). More generally, a boolean expression can be formed from the above relations and boolean functions by using the *logical operators*

```
and
or
not
```

with parentheses used where necessary. A boolean expression is evaluated from left to right and evaluation terminates as soon as the truth value of the entire expression can be deduced. For example, the construct

```
if not(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. Maple is initiated on the Honeywell TSS by entering the command 'maple/sys' to the system prompt '*', as illustrated below. 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>. Notice that one system response is of the form:

```
if ----unable to print---- then . . . .
```

This is due to a current limitation of the output routine which is unable to print out a relation in 'user format'. Note that this limitation of the visible output in no way affects the successful execution of the selection statement.

Exception: If the comments appearing below are entered in an interactive session, the Maple system will respond to each comment line by echoing the comment. For readability, the system response to each comment line has been suppressed in this presentation.

```
*maple/sys

# Integers and rational numbers.

254 + 5280*99999;
527994974
3!;
6
3!!;
720
1 + 1/4 + 1/16 + 1/64 + 1/256;
341/256
(2**50 + 3**20) / 2**100;
1125903393627025/1267650600228229401496703205376
a := ";
a := 1125903393627025/1267650600228229401496703205376
b := 2**100;
b := 1267650600228229401496703205376
a*b;
1125903393627025

# 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;
25
p3;  p5;
9
25
print(p.(1..5));
1  4  9  16  25
x := 333;  x := 'x';
x := 333
x := x

# Polynomials and rational functions.

p := 24*x**2 - 2*x + 7;
p := 24*x**2-2*x+7
q := x**3 + x**2 + x + 1;
q := x**3+x**2+x+1
```

```
r := "" * ";
r := (24*x**2-2*x+7)*(x**3+x**2+x+1)
s := p/q;
s := (24*x**2-2*x+7)/(x**3+x**2+x+1)
r*s;
(24*x**2-2*x+7)**2

# Unevaluated statements.

a; b;
1125903393627025/1267650600228229401496703205376
1267650600228229401496703205376
f := 'a*(b+5)';
f := a*(b+5)
f;
1427252112730298675135602957725914085087021525/1267650600228229401496703205376
max := 'if a>b then a else b fi';
max := if ----unable to print---- then a else b fi
max( );
1267650600228229401496703205376
a := 275/77;  b := 1575/447;
a := 25/7
b := 525/149
max( );
25/7

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

3!!!;
26012189435657951002049032270810436111915218750169457857275418378508356311569473
82240678577958130457082619920575892247259536641565162052015873791984587740832529
10524469038881188412376434119195104550534665861624327194019711390984553672727853
70993456298555867193697740700037004307837589974206767840169672078462806292290321
07161669867260548988445514257193985499448939594496064045132362140265986193073249
36977047760606768067017649166940303481996188145562519559256691883082551494294759
65372748456246288242345265977897377408964665539924359287862125159674832209760295
05696699927284670563747137533019248313587076125412683415860129447566011455420749
58995256354306828863463108496565068277155299625679084523570255218622235813001670
08345234432368219357931847019565107297818043541738905607274280485839959197290217
26612291298420516067579036232337699453964191475175567557695392233803056825308599
97744167578435281591346134039460490126954202883834710136373382448450666009334848
44407119312925376946573543373757247722301815340326471775319845373414786743270484
57983786618703257405938924215709695994630557521063203263493209220738320923356309
```

```
92326750440170176057202601082928804233560664308988871029738079757801305604957634
28386830571906622052911748225105366977566030295740433879834715185526028053338663
57139101046336419769097397432285994219837046979109956303389604675889865795711176
56667003915674815311594398004362539939973120306649060132531130471902889849185620
37666691644687911252491937544258458950003115616829743046411425380748972817233759
55380661719801404677935614793635266265683339509760000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
```

quit
*

4. DATA TYPES AND FORMAL SYNTAX

4.1. Data Types

There are currently six basic data types in Maple.

4.1.1. Name

The role of a <name> in Maple is similar to the role of identifiers in other common programming languages except that in Maple a <name> may not have been assigned any value in which case its value is its own name.

4.1.2. Rational number

Rational numbers are represented by a pair of integers (numerator and denominator) with all common factors removed and with a positive denominator. A Maple integer is represented as a rational number with unit denominator. Integers (and therefore rational numbers) are of arbitrary length, restricted only by the address space of the host processor.

4.1.3. General algebraic expression

A general algebraic expression is any sequence of tokens which forms a valid <expression> as defined by the Maple grammar. It is represented by an expression tree in sum-of-products form.

4.1.4. Polynomial canonical form

The polynomial canonical form in Maple is a special data type for algebraic expressions which represents the expression as a univariate polynomial in one specified indeterminate. The 0th operand in the canonical form is the name of the indeterminate, the 1st, 3rd, . . . operands are the coefficients (generally expressions), and the 2nd, 4th, . . . operands are the corresponding exponents. The exponents are ordered from least to greatest.

4.1.5. Boolean expression

A boolean expression is any sequence of tokens which forms a valid <boolean> as defined by the Maple grammar. When fully evaluated, a boolean expression must yield either the value 1 (true) or the value 0 (false).

4.1.6. Statement sequence

The sixth data type is simply a sequence of one or more valid <statement>'s as defined by the Maple grammar.

4.2. Formal Syntax

This section presents the BNF grammar which describes the syntax accepted by Maple. The non-terminal <name> corresponds to the 'name' data type, the non-terminal <exp> corresponds to the 'general algebraic expression' data type, the non-terminal <bool> corresponds to the 'boolean expression' data type, and the non-terminal <statseq> corresponds to the 'statement sequence' data type. The other two data types – 'rational number' and 'polynomial canonical form' – do not correspond directly to non-terminals in the grammar but rather are special instances of <expression>'s which are treated as special data types for reasons of efficiency.

The non-terminal <filename> which appears in the definition of the read and save statements is not further defined in the formal grammar. The naming conventions for <filename>'s are the file naming conventions of the host system. When a sequence of symbols is enclosed in a pair of daggers (as in †for <name>†) it indicates that this portion of the statement is optional. On the other hand, when the daggers are followed by an asterisk (as in †.†*) it indicates that the enclosed sequence of symbols may appear zero or more times.

```
<session>       ::=   <statseq>

<statseq>       ::=   <statseq> ; <statement> | <statement>

<statement>     ::=   <name> := <exp>  | <exp> | read <filename>   |
                      save <filename> | solve ( <equation>, <name> )   |
                      †for <name>† †from <exp>† †by <exp>†
                          †to <exp>† †while <bool>† do <statseq> od   |
                      if <bool> then <statseq> <elsepart>  |
                      print ( <nameseq> )  | local ( <nameseq> ) | <stop>

<exp>           ::=   <exp> + <term>   | <exp> - <term>  |
                      + <term>  | - <term> | <term>

<term>          ::=   <term> * <factor>  | <term> / <factor> | <factor>

<factor>        ::=   <primary> ** <primary>| <primary>

<primary>       ::=   <natural>  | <name>   | <primary> !   |
                      "   | "" | """| ( <exp> )  | { †<expseq>† } |
                      <name> ( †<expseq>† )  |
                      taylor ( <exp> , <equation> , <exp> )|
                      diff ( <exp> , <nameseq> )   |
                      int ( <exp> , <name> )  |
                      int ( <exp> , <name> = <range> )   |
                      sum ( <exp> , <name> ) |
                                sum ( <exp> , <name> = <range> )  |
                                subs ( <equation> , <exp> )  |
                                simplify ( <exp> , <equatseq> )  |
                                op ( <range> , <exp> )  | ' <statseq> '

<expseq>            ::=         <expseq> , <exp>   | <exp>

<range>             ::=         <exp> ..†.†* <exp>  | <exp>

<equation>          ::=         <exp> = <exp> | <exp>

<equatseq>          ::=         <equatseq> , <equation> | <equation>

<bool>              ::=         <bool> or <bterm>  | <bterm>

<bterm>             ::=         <bterm> and <bfactor>  | <bfactor>

<bfactor>           ::=         not <bfactor>    | ( <bool> ) |
                                <exp> <relop> <exp>   | <name> ? †<expseq>† ?

<relop>             ::=         =  | < | <=    | > | >=

<elsepart>          ::=         fi   | else <statseq> fi     |
                                elif <bool> then <statseq> <elsepart>

<stop>              ::=         quit | done   | stop    | end

<nameseq>           ::=         <nameseq> , <name>    | <name>    | <name> . (<range>)

<name>              ::=         <string>     | <name> . <string> |
                                <name> . <natural> |<name> . ( <exp> )|
                                <name> [ †<expseq>† ]

<string>            ::=         <letter>| <string> <letter>   |
                                <string> <digit>     | <string> _

<natural>           ::=         <digit> | <natural> <digit>

<letter>            ::=         a  | A | b  | B | c  | C | d  | D | e  | E | f  | F |
                                g  | G | h  | H | i  | I | j  | J | k  | K | l  | L |
                                m  | M | n  | N | o  | O | p  | P | q  | Q | r  | R |
                                s  | S | t  | T | u  | U | v  | V | w  | W | x  | X |
                                y  | Y | z  | Z

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

5. FUNCTIONS

One form of an algebraic expression is a function invocation which is generally of the form

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

although some exceptions to this general syntax are noted in section 5.2 below. One form of a boolean expression is a boolean function invocation which has the special syntax

```
<name> ? <expseq> ?
```

Function names are, in general, not reserved words in Maple so that a user may define his own function using the same name as one of the system-supplied functions. The exception to this general rule is that the functions with special syntax in their parameter sequence (as noted in section 5.2 below) have names which appear in the formal syntax and these names are reserved words. There are seven such names and they are listed in section 2.2 as reserved function names.

Functions may be 'built-in functions', 'library functions', or 'user-defined functions'. The built-in functions are part of the basic Maple system and need only to be invoked by the user. The library functions are stored in files and, when invoked, are automatically read in to the current Maple environment. The library functions currently available in Maple are described in section 8 of this manual. Sections 5.1 and 5.2 describe the built-in functions of the Maple system and section 5.3 describes how to write a user-defined function.

5.1. Boolean Functions

There are currently eight built-in boolean functions in the Maple system.

5.1.1. has ? <expl>, <exp2> ?

The value of this function is 'true' if <expl> contains <exp2> as an explicit subexpression, 'false' otherwise. This function is most commonly used when <exp2> evaluates to a <name> in which case the concept of 'explicit subexpression' needs no further explanation. Three examples of the more general situation are:

```
has ?(a+b)**(4/3), a+b? evaluates to 'true' ;
has ?(a+b)**(4/3), a? evaluates to 'true';
has ?a+b+c, a+b? evaluates to 'false' .
```

5.1.2. assigned ? <name> ?

The value of this function is 'true' if <name> has been assigned a value in the current environment, 'false' otherwise.

5.1.3. name ? <exp> ?

The value of this function is 'true' if <exp> evaluates to a name, 'false' otherwise.

5.1.4. constant ? <exp> ?

The value of this function is 'true' if <exp> evaluates to a constant (integer or rational), 'false' otherwise.

5.1.5. integer ? <exp> ?

The value of this function is 'true' if <exp> evaluates to an integer, 'false' otherwise.

5.1.6. addition ? <exp> ?

The value of this function is 'true' if, after evaluation and simplification, <exp> is a sum of two or more terms. Otherwise the value is 'false'.

5.1.7. multiplication ? <exp> ?

The value of this function is 'true' if, after evaluation and simplification, <exp> is a product of two or more factors. Otherwise the value is 'false'.

5.1.8. power ? <exp> ?

The value of this function is 'true' if, after evaluation and simplification, <exp> takes the form

```
<primary> ** <primary> .
```

Otherwise the value is 'false'.

5.2. Algebraic Functions

The first nine built-in algebraic functions described below are functions which have a special syntax in their parameter sequence. The syntax for each of these functions differs from the general syntax

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

in the manner indicated.

5.2.1. op(<range>, <exp>)

The purpose of this function is to extract one or more operands from the algebraic expression <exp>. If <range> is a single expression which evaluates to a nonnegative integer, say i, then the value of the function is the i-th operand in <exp>. General algebraic expressions have operands indexed from 1 to n (for some positive integer n). A function invocation

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

is considered to have as its 0-th operand <name> and the expressions in the parameter sequence <expseq> are operands 1 through n (for some integer n). If <exp> is an expression in polynomial canonical form then the 0-th operand of <exp> is the indeterminate of the canonical form, the 1-st, 3-rd, . . . operands are the 'coefficients' and the 2-nd, 4-th, . . . operands are the corresponding exponents (with the exponents ordered from least to greatest). If <range> is not a single expression but is of the form 0..i where i evaluates to an integer, then <exp> must be a function invocation and the result is a new function invocation consisting of operands 0 through i of <exp>.

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

```
g := f(x, y, z);
op0 := op(0, g);  op2 := op(2, g);
op02 := op(0..2, g);
a := (3*sin(x**3) - (2/3)*x + y) / (2*x**2 - 1);
first := op(1, a);  second := op(2, a);
term := op(2, op(1, first));
```

then

```
op0    has the value  f
op2    has the value  y
op02   has the value  f(x,y)
first  has the value  3*sin(x**3)-2/3*x+y
second has the value  (2*x**2+(-1))**(-1)
term   has the value  sin(x**3) .
```

If the following statements are executed:

```
b := 5/2*x*sin(x)**4 + 3*x**3*sin(x) - 13*sin(x) - 5;
p := expand(b, sin(x));
```

then p is a polynomial canonical form and

```
op(0,p) has the value sin(x)
op(1,p) has the value -5
op(2,p) has the value 0
op(3,p) has the value 3*x**3+(-13)
op(4,p) has the value 1
op(5,p) has the value 5/2*x
op(6,p) has the value 4 .
```

5.2.2. subs(<equation>, <exp>)

The purpose of this function is to substitute <equation> into <exp>. Every occurrence in <exp> of the expression appearing on the left hand side of <equation> is replaced by the expression appearing on the right hand side of <equation>.

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

```
f := 3*x*ln(x**3);
fnew := subs(x=1, f);
g := (a+b)**(4/3);
gnew := subs(a+b = y, g);
```

then

```
fnew  has the value  0
```

and

```
gnew  has the value  y**(4/3) .
```

5.2.3. taylor(<exp1>, <equation>, <exp2>)

The purpose of this function is to compute the Taylor series (more generally, Laurent series) expansion of <exp1> with respect to the indeterminate given by the left hand side of <equation> about the point given by the right hand side of <equation>. (If <equation> is simply an <exp> then the equation <exp> = 0 is understood). The 'truncation degree' to be used in the computation of the series is given by <exp2>.

Example: Assuming that x is a name which stands for itself, if the following statements are executed:

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

then the results of the two function invocations of 'taylor' are, respectively:

```
1+(-5/7)*x+57/98*x**2+(-509/2058)*x**3
```

and

```
(-2/7)+11/49*(x-1)+167/343*(x-1)**2 .
```

5.2.4. diff(<exp>, <nameseq>)

The purpose of this function is to compute the partial derivative of <exp> with respect to the indeterminates name1, name2, . . . where <nameseq> is name1, name2, ...

Example: 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) .

5.2.5. int(<exp>, <name>)

The purpose of this function is to compute the indefinite integral of <exp> with respect to <name>. Except for simple cases, this function will invoke Maple library functions to be read in.

Example: If

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

then

```
int(f,x) has the value -1/2*x**(-1) + 3/2*ln(x) + 2*x -5/4*x**2 + 7/6*x**3 .
```

5.2.6. int(<exp>, <name> = <range>)

The purpose of this function is to compute the definite integral of <exp> with respect to <name> over the interval specified by <range>. Except for simple cases, this function will invoke Maple library functions to be read in.

Example: If f is as in 5.2.5 above then

```
int(f, x=1..2) has the value 20/3 + 3/2*ln(2).
```

5.2.7. sum(<exp>, <name>)

The purpose of this function is to compute the 'indefinite' sum of <exp> with respect to <name>. Except for simple cases, this function will invoke Maple library functions to be read in.

Example:

```
sum(i**2, i) has the value 1/2*i*(i-1) + 1/3*i*(i-1)*(i-2)
```

5.2.8. sum(<exp>, <name> = <range>)

The purpose of this function is to compute the sum of <exp> with respect to <name> over the discrete interval specified by <range>. Except for simple cases, this function will invoke Maple library functions to be read in.

Example: If

```
e := (5*i - 3)*(2*i + 9);
```

then

sum(e, i=1..50) has the the value 477625
sum(e, i=1..n) has the value −27*n + 49/2*(n+1)*n + 10/3*(n+1)*n*(n−1)
expand(", n) has the value (−35/6)*n + 49/2*n**2 + 10/3*n**3.

5.2.9. simplify(<exp>, <equatseq>)

The purpose of this function is to 'simplify' <exp> with respect to <equatseq>. The elements of <equatseq> are applied in order. An element in <equatseq> which is an equation (e.g. i**2 = −1) causes this equation to be applied as a side relation. An element in <equatseq> which is simply an <exp> is interpreted as the equation <exp> = 0 if <exp> is not simply a <name>. When a <name> is encountered as an element of <equatseq> then the system attempts to perform simplification with respect to <name>. (e.g. The <name> may be ln, exp, sin, or cos in which case the standard system simplifications with respect to the functions with these names are applied).

The remaining built-in algebraic functions conform to the normal syntax for functions.

5.2.10. sin(<exp>)

This is the mathematical sine function and it is built-in in the sense that it is known to the Maple simplifier.

5.2.11. cos(<exp>)

This is the mathematical cosine function and it is built-in in the sense that it is known to the Maple simplifier.

5.2.12. exp(<exp>)

This is the exponential function and it is built-in in the sense that it is known to the Maple simplifier.

5.2.13. ln(<exp>)

This is the natural logarithm function (logarithm to the base exp(1)) and it is built-in in the sense that it is known to the Maple simplifier.

5.2.14. trunc(<exp>)

The value of this function when <exp> evaluates to a constant is the 'integer part' of <exp> when expanded in a decimal expansion. For example, trunc(8/3) is 2 and trunc(−8/3) is −2.

5.2.15. igcd(<expl>, <exp2>)

This is the 'integer greatest common divisor' function whose value when <expl> and <exp2> evaluate to integers is the nonnegative greatest common divisor of the two arguments.

5.2.16. expand(<expl>, <exp2>)

The purpose of this function is to expand <expl> into polynomial canonical form with respect to the indeterminate <exp2>. For example,

```
p := (2*x - 5) * (35*x**2 - x + 7);
expand(p, x);
```

yields the result

```
(-35)+19*x+(-177)*x**2+70*x**3 .
```

As another example,

```
q := 3*sin(x) * (x*sin(x) - y*z) * (2*x**2 - 3);
expand(q, sin(x));
```

yields the result

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

5.2.17. coeff(<expl>, <exp2>, <exp3>)

For this function, the expression <expl> must be in polynomial canonical form with respect to the indeterminate <exp2>. The value of this function is the coefficient in <expl> of the term involving <exp2>**<exp3> . For example, let p and q be as in 5.2.16 above and let

```
expandp := expand(p, x);
expandq := expand(q, sin(x));
```

as computed in 5.2.16 above. Then

```
coeff(expandp, x, 2)  yields  -177
```

and

```
coeff(expandq, sin(x), 1)  yields -3*z*y*(2*x**2+(-3)) .
```

5.2.18. degree(<expl>, <exp2>)

The purpose of this function is to determine the degree of <expl> with respect to the indeterminate <exp2>. The result is the degree of the polynomial in <exp2> which would be produced by the function call

```
expand( <expl>, <exp2> ) .
```

For example, if p and q are as in 5.2.16 above then

```
degree(p, x)       has the value  3
degree(q, sin(x))  has the value  2
degree(q, x)       has the value  3
degree(q, z)       has the value  1 .
```

5.2.19. nops(<exp>)

The purpose of this function is to determine the number of operands appearing in <exp>. The manner in which <exp> is viewed by nops corresponds to the manner in which an expression is viewed by the function op. If <exp> is a general algebraic expression with operands indexed from 1 to n then nops(<exp>) is n. If <exp> is a function invocation with operands indexed from 0 to n then nops(<exp>) is again n. If <exp> is a polynomial canonical form with respect to an indeterminate x and if <exp> has m terms as a polynomial in x then nops(<exp>) is 2*m. (See section 4.1.4).

Example: 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);
```

as in the example in 5.2.1 then

```
nops(g)          has the value  3
op(0..nops(g), g)  has the value  f(x,y,z)
nops(a)          has the value  2
nops(op(1,a))  has the value  3
nops(op(2,a))  has the value  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). If expandp and expandq are as in 5.2.17 above then

```
nops(expandp)  has the value 8
nops(expandq)  has the value 4.
```

If <exp> is a constant then nops(<exp>) = 2 even if <exp> is an integer, because integers are stored as rational numbers. For example, if a := 3 then

```
nops(a)  has the value 2
op(1,a)  has the value 3
op(2,a)  has the value 1.
```

5.2.20. indets(<exp>)

The purpose of this function is to determine the indeterminates which appear in <exp>. The value of the function is a set whose elements are the indeterminates. The concept of 'indeterminate' is that <exp> 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), x**(1/2) are treated as indeterminates.

[Note: As of this writing, the value of the indets function is not a set but rather an ordered list using the construct LIST(e1, . . . , en) where e1, . . . , en are the indeterminates. The name LIST is recognized by the system function REST such that the value of REST(LIST(e1, . . . , en)) is LIST(e2, . . . , en). A switch to the use of sets will be implemented soon.]

Example: 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)  has the value LIST(x, y, z)
indets(r)  has the value LIST(x, (x−2)**(1/3), exp(x**2)).
```

5.3. User-defined Functions

In Maple there is the concept of an unevaluated statement sequence:

```
' <statseq> '
```

as a valid expression in the language. The definition of a function in Maple flows very naturally from this concept; namely, a function definition takes the form

```
<name> := ' <statseq> '
```

which we already know to be a valid statement in Maple. For functions which do not require any parameters to be passed and which do not require any variables to be local to the function, the above construction of a function needs no further explanation. Such a function is invoked either as a boolean function using the construct

```
<name> ??
```

or as an algebraic function using the construct

```
<name> ( )
```

which are function invocations with an empty <expseq> for the list of actual parameters. A Maple function which is invoked as a boolean function must evaluate to the value 1 ('true') or the value 0 ('false'). In the special case where <statseq> is a single algebraic expression, the two expressions

```
<name>

<name> ( )
```

evaluate to precisely the same value. However if <statseq> is not a single valid expression then the former is an invalid expression, while the latter is a function invocation whose value is the value of the last statement in <statseq>.

An example of a simple function with no parameters and with no local variables was seen in the sample Maple session of section 3.3. Namely, defining

```
max := 'if a>b then a else b fi';
```

then executing the statements

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

yields 25/7 as the value of the function invocation max(). As an example of a boolean function, let us define

```
a_greater := ' if a>b  then 1 else 0  fi';
```

if a and b have values as above then executing the statement

```
if a_greater?? then print(a) fi;
```

causes the value 25/7 to be printed out.

The mechanism for introducing *parameters* into a Maple function is as follows. A function is defined by a statement of the form

```
<name> := '<statseq>' .
```

The following names have a special meaning when used within the statements in <statseq>:

```
nargs
param1, param2, param3, . . . .
```

Specifically, when the function is invoked with actual parameters as in

```
<name> ( <exp1>, <exp2>, . . . , <expn> )
```

then every occurrence in <statseq> of the special name nargs is replaced by the integer n, where n is the number of actual parameters specified in the function invocation. Similarly, every occurrence in <statseq> of the special name param1 is replaced by the result of evaluating the first actual parameter, every occurrence in <statseq> of the special name param2 is replaced by the result of evaluating the second actual parameter, and so on up to param.nargs. There may be too many actual parameters, in which case the effect is that the extra actual parameters are simply ignored. There may be too few actual parameters, in which case the extra formal parameter names 'parami' remain unreplaced in the function definition (and may show up in the result of the function invocation).

As an example of a function with parameters, let us change the above definition of max so that the two values are passed in as parameters. Defining

```
max := ' if param1 > param2 then param1 else param2 fi ';
```

then executing the statements

```
r := 25/7;  s := 525/149;  max(r,s);
```

yields 25/7 as the value of the function invocation max(r,s). If we specify too many actual parameters as in

```
max(25/7, 525/149, 9/2);
```

then the third actual parameter is simply ignored and the value returned is exactly as before. If we specify too few actual parameters in this case, an execution error will result because the relation > is invalid when the difference of its operands involves an unevaluated name. (In this case the name param2 would remain as an unevaluated name).

Let us now make use of the special name nargs to generalize our function max so that it will be defined to calculate the maximum of an arbitrary number of actual parameters. Consider the following function definition:

```
max :=
' result := param1;
  for i from 2 to nargs do
    if param.i > result then result := param.i fi
  od;
  result ';
```

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

```
max(25/7, 525/149)       has the value  25/7
max(25/7, 525/149, 9/2)  has the value  9/2
max(25/7)                has the value  25/7
max()                    has the value  param1
```

where the latter case is an example of a function being called with too few actual parameters. If we wish to change our definition of max so that the function invocation max() with an empty parameter list will return no value (more precisely, it 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 :=
' if nargs > 0 then
    result := param1;
    for i from 2 to nargs do
      if param.i > result then result := param.i fi
    od;
    result
  fi ';
```

Let us now consider an example of a function where we may wish to return a value into one (or more) 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'

$$a = bq + r$$

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 function 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 :=
' trunc(param1/param2);
  if nargs > 2 then param3 := " fi;
  param1 - " * param2 ';
```

The function rem as defined here may be invoked with either two or three parameters. In either case the value of the function 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 it is crucial to understand that the parameter passing is 'call by evaluated name'. That is to say, 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 such an actual parameter should usually be explicitly quoted (which effects a true 'call by name' parameter passing mechanism). The following statements will serve to illustrate.

```
rem(5, 2);
  yields the value 1
rem(5, 2, 'q'); q;
  yields the values 1 and 2
rem(−8, 3, 'q'); q;
  yields the values −2 and −2
rem(8, −3);
  yields the value 2
rem(8, 3, q);
  yields error (in evalname)
```

The latter 'error (in evalname)' 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 param3 in the function definition yielding an invalid assignment statement. The solution to this problem is to change the actual parameter from q to 'q'.

The mechanism for introducing *local variables* into a Maple function is to use the local statement which was described in section 3.1. The syntax of the statement is

```
local( <nameseq> )
```

and the semantics are that the names appearing in <nameseq> are to be 'local from here to the end of the current statement sequence'. In other words, this statement can be viewed as causing a syntactic renaming of every occurrence of the specified names from the local statement to the end of the statement sequence in which the local statement appears. The use of the local statement is not restricted to functions, but a typical use would be to have a local statement as the first statement in a function and then its 'range of locality' would be the entire statement sequence which defines the function. As an example, let us reconsider the latest definition of max appearing above. There are two global variables appearing in the function definition which we would almost certainly want to make local: result and i. This is effected by the following version of the function definition.

```
max :=
' local(result, i);
  if nargs > 0 then
    result := param1;
    for i from 2 to nargs do
      if param.i > result then result := param.i fi
    od;
    result
  fi ';
```

We give one more version of the max function which will be more efficient in Maple.

```
max :=
 ' if nargs > 0 then
     local(i);
     param1;
     for i from 2 to nargs do
        if param.i > " then param.i fi
     od;
     "
  fi ';
```

Here we have exploited the operator " in order to avoid any assignments and thus improve the efficiency of execution. Another factor in improving the efficiency is that the number of variables declared to be local has been reduced from two to one. It is worth noting here that the final " appearing in the above function would be redundant in some contexts but is necessary in this function. If it were left out then the value of the function invocation would be the value of the last statement executed, which would be the value of the if-statement inside the for-loop when i=nargs. This value will be null unless param.nargs is greater than all of the other actual paremeters (in precisely the same way that the function invocation max() returns the null value). However, the value of " is never updated by a null value and this fact is exploited in the above function definition.

Noting that a function definition for a boolean function is identical in form to a function definition for an algebraic function (the only syntactic distinction arises in the *invocation* of the function), it follows that a particular function definition could be invoked as an algebraic function and as a boolean funcion in the same session. The important semantic property of a boolean function is that it must return the value 1 ('true') or 0 ('false'). Consider the following definition of a function integer_divide which divides a two-component vector by an integer if the division is exact over the integers.

```
integer_divide :=
 ' if integer ?param1.1/param2? and integer ?param1.2/param2? then
     param1.1 := param1.1/param2;
     param1.2 := param1.2/param2;
     1
  else
     0
  fi ';
```

If v.1 := 26 and v.2 := 39 then the construct

```
if integer_divide ?v,13? then . . . fi;
```

will cause the components of v to be changed to 2 and 3, respectively, and the statements in the then-clause will be executed. If the components of v are 26 and 39 as before, the statement

```
integer_divide(v,13);
```

will also cause the components of v to be changed as above.

As a final remark about functions, note that it is usually convenient to use a text editor to develop a function definition and to write it into a file. The file can then be read into a Maple session. For example, the max function might be written into a file named /mlib/max . In a Maple session the statement

```
read /mlib/max;
```

will read in the function definition. Since this function is in 'user format' Maple will echo the statements as they are read in. Once the function 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 function definition in a file named /mlib/max. If we then enter the Maple system and execute the statements

```
read /mlib/max;
save /mlib/max.m;
```

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

```
read /mlib/max.m;
```

which will update the current Maple environment with the contents of the specified file. The user will quickly discover the time-saving advantages of saving function definitions in 'Maple internal format'.

6. INTERNAL REPRESENTATION AND MANIPULATION

This section is not yet written.

7. MISCELLANEOUS FACILITIES

7.1. Debugging Facilities

There are two names whose values determine the amount of information displayed to the user during execution of a Maple session:

```
yydebug
printlevel.
```

The default value for both of these names 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 trace of the Maple session from the basic system viewpoint.

A more useful facility from the user viewpoint is the printlevel option. 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 two levels of statements recognized within a particular function (or in the main session): statements within selection or repetition statements and 'mainstream' statements. Normally (with printlevel := 0) the following statements within the main session

```
b :=2;
for i to 5 do a.i := b**i od;
```

would cause the printout b:=2 after execution of the first statement and the printout 32 after execution of the for-statement (the value of the for-statement is the value of the last statement executed). If the user assigns

```
printlevel := 1;
```

before the above statements are executed then, in addition, 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
32
```

More generally, statements are nested to various levels by the nesting of functions. The Maple system decrements the value of printlevel by 2 upon each entry into a function and increments it by 2 upon exit, so that normally (with printlevel:=0) there is no information displayed from statements within a function. If the user assigns

```
printlevel := 2;
```

in the main session then each statement within a function nested to one level only (but not statements within loops in the function) will be displayed as it is executed (because the effective value of printlevel within the function is 0). If the user assigns

```
printlevel := 3;
```

in the main session then, in addition, statements within selection and repetition statements in the function will be displayed (because the effective value of printlevel within the function is 1). Alternatively, the user may explicitly set the value of printlevel within the function 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 functions to various levels of nesting. When the effective value of printlevel upon entry to a function is 15 or greater, the entry point and exit point for that function are displayed in the printout. Thus if the user assigns

```
printlevel:=20;
```

in the main session then entry and exit points will be displayed for functions only up to two levels deep (because for a function 3 or more levels deep the value of printlevel will have been decremented by 6 or more, making its effective value less than 15). However with this setting statements will be displayed from within functions nested up to 10 levels. It is not uncommon to use a debug setting such as

```
printlevel := 100;
```

in which case all entry and exit points will be displayed for functions nested up to 42 levels deep, and statements will displayed from within functions up to 50 levels deep. For more selective debugging information, the value of printlevel should be assigned within specific functions.

7.2. Echoing and Timing

Another name known to the Maple system is the name echo. Its default value is 0. If the user assigns

```
echo := 1;
```

then the system creates a temporary file named .echo. into which it saves (in 'user format') every input line entered into the Maple session from this point until the name echo is assigned the value 0 (or until the Maple session is ended).

A common use of the echo facility would be to re-run a Maple session, perhaps after some alterations. A standard procedure would be to use a text editor to alter the .echo. file before re-running it. Noting that the Maple system displays only system responses and not user input lines, it is useful to use a text editor to create a comment line for each input line in a file before running it through Maple. For example, if a file contains the statements

```
a:=4;
quit
```

then it might be altered by the qed or fred command

```
g/^/k(t) s/^/#-->/ za(t)
```

yielding

```
#-->a:=4;
a:=4;
#-->quit
quit
```

which could be written into a file, say /example. Then executing the command

```
maple/sys </example >/output
```

would run the file through Maple, putting the output into the file /output, and since Maple echos comments the input preceding each system response will be seen when the /output file is examined. (Note that often a user will create an input file initially using a text editor, applying the above ideas, without requiring the echo facility).

The time command on Honeywell TSS may be used to obtain timing information for a Maple run. To put this command into effect simply precede the maple/sys command by the time command, as in

```
time maple/sys </example >/output
```

The timing information will be displayed at the terminal after execution has completed.

7.3. Facilities for Names

When a user is using Maple for an interactive session, he may wish to know at some point which names he has used and whether they have been assigned values or simply stand for themselves. There are two system functions for this purpose:

```
assigned_names()
unassigned_names() .
```

(Note: As of this writing, these two functions are called variables() and indeterminates(), respectively, but these names are expected to change to the above in a later version of maple/sys). The first function prints out all names in the current Maple environment which have a value other than their own name, and the second function prints out all names in the current Maple environment which stand for themselves.

Another facility relating to names is the keepdot option. If at the beginning of a session the user specifies

```
keepdot;
```

then the '.' operator will not disappear when names are formed using concatenation. For example, the statements

```
n := 4; a.sub.n;
i :=15; j := 3; p.i.j;
```

normally yield the names asub4 and p153 while if the keepdot option has been specified then the above statements will yield the names a.sub.4 and p.15.3 .

7.4. Other Facilities

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. For example, on Honeywell TSS one could list a file without leaving Maple by using the command

```
!list <filename> .
```

As of this writing, the Maple system has no garbage collection facility. It can be useful to effect a manual garbage collection from the interactive level of Maple by using the sequence

```
save temp.m;
quit
```

followed by re-entering maple/sys and then

```
read temp.m;
```

This will restore the Maple environment but with all 'garbage' having disappeared. The user is given an indication of the amount of core being used by the printout

```
core used = xxxxx
```

which is randomly displayed. Note that on Honeywell TSS the effective maximum core which can be used is about 110000 words.

8. THE MAPLE LIBRARY

This section is not yet written.