General Information

- *expressions* — index of descriptions for Maple expressions
- *expression sequences* — expression sequences
- *functions* — index of Maple functions
- *statements* — index of descriptions for Maple statements
- *procedure* — procedures

Variables

Description of expressions:

!	""""	""	"	$	
**	*	+	-	.	
/	<	<=	<>	=	
>	>=	@	@@	^	
and	arithmetic	boolean	constant	ditto	
factorial	fraction	integer	intersect	minus	
mod	name	neutral	not	nullstring	
operators	or	ratpoly	selection	set	
uneval	union				

An expression is of type integer if it is an (optionally signed) sequence of one or more digits of arbitrary length. The length limit of an integer is system dependent but generally much larger than users will encounter - typically greater than 500,000 decimal digits.

In addition to arithmetic operators, other basic functions of integers are:

abs — sign — min — max — factorial
irem — iquo — modp — mods — mod
isqrt — iroot — isprime — ifactor — ifactors
igcd — ilcm — igcdex — iratrecon — rand

There are also many special functions for integers in the number theory and combinatorial packages including the binomial coefficients, Fibonacci numbers, Stirling numbers, Jacobi symbol, Eulers totient function, etc.

Description of expression sequences:

Expression sequences, (or simply sequences), are created using the comma operator ,.

For example *s := 1,2,3* assigns *s* the sequence *1,2,3.* In Maple, sequences form the basis of many data types. In particular, they appear in function calls, lists, sets and subscripts.

Example:

- f(s) applies the function f to the sequence 1,2,3,
- [s] creates the list containing the elements 1,2,3,
- {s} creates the set containing the elements 1,2,3, and
- a[s] is the 1,2,3 subscript.

These are equivalent to f(1,2,3), [1,2,3], {1,2,3}, and a[1,2,3] respectively.

Two key tools for constructing sequences are the *seq* function and the repetition operator *$*. For example, the call *seq(f(i), i=1..3)* will generate the sequence *f(1), f(2), f(3)*. The call *x$3* will generate the sequence *x, x, x*.

Description of statements:

!	#	ERROR		assignment	break
by	do	done		elif	else
empty		fi	for	from	function
if	in	next		od	proc
quit	read	restart		save	separator
stop	then	while			

Description of Procedures:

Calling sequence:

- proc (argseq) local nseq; global nseq; options nseq; description stringseq; statseq end

A procedure definition is a valid expression that can be assigned to a name.

The parenthesized argseq, which may be null, specifies the formal parameter names. Each parameter is a name followed by an optional type specifier preceded by a double colon. If the type specification is supplied, Maple will generate an error if arguments of the incorrect type are passed to the function. Maple will also generate an error if an argument is missing, but only at the time that such an argument is first needed.

The phrases *local nseq;, global nseq;* and *options nseq;* are optional. If present, they specify, respectively, the names of local and global variables and the options in effect.

If variables are undeclared, the following rules are used to determine whether a variable is local or global: by default, each variable to which an assignment is made, or which appears as the controlling variable in a for loop, is local. All others are global.

Procedures have special evaluation rules (like tables) so that if the name *f* has been assigned a procedure then *f* evaluates to the name *f*; *eval(f)* yields the actual procedure structure; op(eval(f)) yields the sequence of six operands mentioned above (any or all of which may be null).

A procedure assigned to *f* is invoked via: *f(arguments)*. The value of a procedure invocation is the value of the last statement executed, or else the value specified in a *RETURN* call.

Example:

```
> f := proc(x) local y; y := x * 2; g(x) / 4; end;
```

$$f := \mathbf{proc}(x)\ \mathbf{local}\ y;\ y := x * 2;\ \mathrm{g}(x) / 4;\ \mathbf{end}$$

```
> g := proc(x) local z; z := x^2; z * 2; end;
```

$$g := \mathbf{proc}(x)\ \mathbf{local}\ z;\ z := x \wedge 2;\ z * 2;\ \mathbf{end}$$

```
> trace(f,g);
```

$$f, g$$

```
> f(3);
{--> enter f, args = 3
```

$$y := 6$$

```
{--> enter g, args = 3
```

$$z := 9$$

$$18$$

```
<-- exit g (now in f) = 18}
```

$$\frac{9}{2}$$

```
<-- exit f (now at top level) = 9/2}
```

$$\frac{9}{2}$$

```
> f(3):

{--> enter f, args = 3
{--> enter g, args = 3
<-- exit g (now in f) = 18}
<-- exit f (now at top level) = 9/2}
```

Definition of a type in Maple

Description:

By definition, a type in Maple is any expression which is recognized by the type function and causes it to return true or false for some set of expressions. For example, in type(expr, typexpr), expr is of type typexpr if and only if the above expression evaluates to true.

Any particular type belongs to one of the following four categories:

1. A system type: a type which is a name that is defined in the kernel of the Maple system. System types usually correspond to primitive algebraic or data structure concepts, such as integer, float, list, and relation.

2. A procedural type where the type is a name, for example xxx, and there is a procedure \`type/xxx\` which will perform the type analysis of the argument and will return true or false accordingly. This procedure may be available from the global Maple environment or from the library (for the latter case it is automatically loaded into the environment). This is one of the mechanisms to define new types in Maple. monomial, algnum, and odd are presently implemented as external Maple functions.

3. An assigned type where the type is a name, such as *xxx*, and the global name \`*type/xxx*\` is assigned a type expression. The type evaluation proceeds as if the type checking were done with the expression assigned to \`*type/xxx*\`. Thus

```
`type/intargs` := [algebraic, {name, name=algebraic..algebraic}];
. . . .
if not type([args],intargs) then ERROR(...)
```

4. A type expression which is a general Maple expression as described in type[structured].

Example: the type algebraic

Description:

An expression is of type algebraic if it is one of the following types:

Integer	*fraction*	*float*	*string*	*indexed*
\`+\`	\`*\`	\`^\`	\`**\`	*series*
function	\`!\`	\`.\`	*uneval*	

5. Input and Output

Input:

- *fscanf* — formatted printing to a file or pipe
- *read* — the read statement
- *readbytes* — read bytes from a file or pipe as a string or list
- *readdata* — read raw data from files
- *readline* — read a string from the terminal or a file
- *readstat* — read one statement from the input stream
- *sscanf* — scan and parse numbers and strings within a string

Output:

- *interface* — set or query user interface variables
- *fprintf* — formatted printing to a file or pipe
- *lprint* — linear printing of expression
- *print* — pretty-printing of expressions
- *printf* — display objects using a specified format
- *printlevel* — printlevel (display of information; debugging procedures)
- *TEXT* — the TEXT data structure
- *writebytes* — write bytes from a string or list to a file or pipe
- *writedata* — write numerical data to a text file
- *writeline* — writes strings to a file or pipe
- *writestat* — writes a string — or expression(s) to a file or pipe

Translation:

- *C* — generate C code
- *eqn* — produce output suitable for troff/eqn printing
- *fortran* — generate Fortran code
- *latex* — produce output suitable for latex printing
- *optimize* — common subexpression optimization

The *read* statement

Calling sequence:

read filename

Description:

The read statement is used to read Maple language or Maple internal format files into Maple.

If the filename ends with the characters ".m", the file is assumed to be in Maple internal format. The objects stored in the file are read into Maple and become available for use.

If the file is in Maple language format, the statements in the file are read and executed as if they were being typed in. However, the statements are not echoed to the display unless interface(echo) is set to 2 or higher.

If filename contains unusual characters (e.g., "/", ".", etc.) then the name must be enclosed in backquotes.

To load a file from the standard Maple library, use the readlib function.

Examples:

```
> read `lib/f.m`;
> read temp;
> read `temp.m`;
```

Readdata - read numerical data from text files

Calling Sequence:

readdata(fileID, n)
readdata(fileID, format, n)
readdata(fileID, format)

Parameters:

- fileID - the name or descriptor of the data file
- n - positive integer: for how many columns of data
- format - specifies if the data is to be read as integer or floating point values

Description:

The readdata function reads numeric data from an input text file into Maple. The data must consist of integers or floats arranged in columns separated by white space. If only one column of data is read, the output is a list of the data. If more than one column is read, the output is a list of list of rows of data corresponding to the rows of data in the file.

The fileID argument gives the name or file descriptor of the file from which to read.

The first form of the readdata function reads n columns of numerical data from the file specified by fileID in floating point format.

The second and third forms specify whether the data is to be read as integer, floating point, or string data. The parameter format must be one of integer, float, or string, or a list containing one or more of these.

If a file name is given (instead of a file descriptor), and the file is not already open, the file will be opened as a TEXT file in READ mode. Furthermore, if a file name is given, the file will be closed when readdata returns.

Examples:

The file data: 1 1 50 1 2 55 2 1 55 2 2 70 read 3 columns of floats from the file 'data':

```
    > readdata(data,3);
```

read 3 columns as integers

```
    > readdata(data,integer,3);
```

read first column of integers

```
    > readdata(data,integer);
```

read first column as floats

```
    > readdata(data,float);
```

read the first two columns as integers and the third as floats

```
> readdata(data,[integer,integer,float]);
```

Printing files and expressions

Function *fprintf* - prints expressions to file or pipe based on a format string
Function: *sprintf* - Prints expressions to a string based on a format string
Function: *printf* - Prints expressions to default stream based on a format string

Calling Sequence:

fprintf(file, fmt, x1, ..., xn);
sprintf(fmt, x1, ...,xn);
printf(fmt, x1, ...,xn);

Parameters:

- file - a file descriptor or file name
- fmt - the output format specification
- x1, ..., xn - the expressions to be formatted

Description:

The *fprintf* function is based on a C standard library function of the same name. It uses the formatting specifications in the fmt string to format and write the expressions to the specified file.

The *sprintf* function is based on a C standard library function of the same name. It uses the formatting specifications in the fmt string to format and write the expressions into a Maple string, which is returned.

The *printf* function is based on a C standard library function of the same name. It uses the formatting specifications in the fmt string to format and display the expressions.

The *fprintf* function returns a count of the number of characters written.

Flow Control

- *break* the break construct
- *empty statement* the empty statement
- *ERROR* error return from a procedure
- *if* the selection (conditional) statement
- *quit* the quit statement
- *RETURN* explicit return from a procedure

Iteration or Looping:

$	operator for forming an expression sequence
..	expressions of type range
do	the repetition (for / while / do) statement
for	the repetition (for / while / do) statement
next	the next construct
product	definite and indefinite product
seq	create a sequence
sum	definite and indefinite summation
while	the repetition (for / while / do) statement

The repetition (for / while / do) statement:

Description:

SYNTAX:

```
|for <name>| |from <expr>| |by <expr>| |to <expr>| |while <expr>|
                  do <statement sequence> od;
```

OR,

```
|for <name>| |in <expr>| |while <expr>|
                  do <statement sequence> od;
```

(Note: | | indicates an optional phrase.)

The repetition statement provides the ability to execute a statement sequence repeatedly, either for a counted number of times (using the for-to clauses) or until a condition is satisfied (using the while clause). Both forms of clauses may be present simultaneously.

If the from or by clause is omitted then the default value from 1 or by 1, respectively, is used.

The tests to *expr* and *while expr* are tested at the beginning of each iteration. If neither clause is present then the loop will be infinite. Exit from such a loop is possible via the *break* construct, via a *RETURN* from a procedure, or via the *quit* statement.

The *expr* in the *while* clause is a Boolean expression which must evaluate to true or false; otherwise, an error occurs.

Use of the *in expr* clause will cause the index variable to take as values the successive operands of the specified expression *expr* (as would be determined via the op function).

Arguments to the *in* or *to* clauses are evaluated only once at the beginning of the loop and not after every iteration.

Examples:

1) print even numbers from 6 to 100:

```
> for i from 6 by 2 to 100 do  print(i)  od;
```

2) find the sum of all two-digit odd numbers:

```
> sum := 0;
> for i from 11 by 2 while i < 100 do
>    sum := sum + i
> od;
```

3) add together the contents of a list:

```
> sum:=0;
> for z in bob do
>   sum:=sum+z
> od;
```

4) add together the contents of a list:

```
> sum:=0;
> for z in bob do
>   sum:=sum+z
> od;
```

The selection (conditional) statement and operator

Calling Sequence:

```
if conditional expression then statement sequence
| elif conditional expression then statement sequence |
| else statement sequence |
fi
if(conditional, true statement, false statement)
```

Description:

The construct *elif conditional expression then statement sequence* may be repeated any number of times. The keyword *elif* stands for *else if*; the short form avoids the requirement for multiple closing *fi* delimiters.

A conditional expression is any Boolean expression formed using the relational operators (<, <=, >, >=, =, <>), the logical operators (and, or, not), and the logical names (true, false).

When a conditional expression is evaluated in this context, it must evaluate to true or false; otherwise, an error occurs.

The statement sequence following else will be executed if all of the conditional expressions evaluate to false.

Examples:

```
> a := 3; b := 5;
> if (a > b) then a else b fi;
> 5*(Pi + `if`(a > b,a,b));
                         5 Pi + 25
```

The *next* construct

Description:

When the special name *next* is evaluated, the result is to exit from the current statement sequence (i.e., the do-od block) corresponding to the innermost repetition (for/while/do) statement within which it occurs.

To exit from the current statement sequence means to allow execution to proceed with the next iteration of this repetition statement.

Note that to proceed with the next iteration implies incrementing the index of iteration and then applying the tests for termination (specified by the to-clause and while-clause, if present) before proceeding. Thus, an exit from the loop may occur.

It is an error if the name next is evaluated in a context other than within a repetition statement.

Note that next is not a reserved word and therefore it is possible (but unwise) to assign a value to it.

2D

plot	create a 2D plot of functions
function	acceptable plot functions
branches	plot the branches of a multi-valued function
infinity	infinity plots
multiple	multiple plots
parametric	parametric plots
animate	create an animation of 2D plots of functions
conformal	conformal plot of a complex function
densityplot	2D density plotting
display	display a list of plot structures
fieldplot	plot a 2D vector field
gradplot	plot a 2D gradient vector field
implicitplot	2D implicit plotting
logplot	create a 2D log-plot of functions

loglogplot	create a 2D log-log plot of functions
odeplot	2D or 3D plot of output from dsolve(, numeric)
polar	polar coordinate plots
polygonplot	create a plot of one or more polygons
replot	redo a plot
sparsematrixplot	2D plot of nonzero values of a matrix
textplot	plot text strings
structure	plot structure
geometry[draw]	drawing geometric objects
options	options to the plot command

3D

plot3d	3D plotting of functions
addcoords	add a new coordinate system
animate3d	create an animation of 3D plots of functions
contourplot	contour plotting
cylinderplot	plot a 3D surface in cylindrical coordinates
densityplot	2D density plotting
display3d	display a set of 3D plot structures
fieldplot3d	plot a 3D vector field
gradplot3d	plot a 3D gradient vector field
implicitplot3d	3D implicit plotting
matrixplot	3D plot with z values determined by a matrix
odeplot	2D or 3D plot of output from dsolve/numeric
pointplot	create a 3D point plot
polygonplot3d	create a plot of one or more polygons
polyhedraplot	create a 3D point plot with polyhedra
spacecurve	plotting of 3D space curves
sphereplot	plot a 3D surface in spherical coordinates
surfdata	create a 3D surface plot from data
textplot3d	plot text strings
tubeplot	3D tube plotting