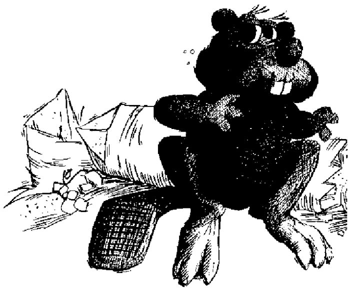

ARRAYS AND TABLES IN MAPLE: Supplement to the Maple User's Manual

Stephen Watt

Research Report
CS-83-10

May, 1983

ARRAYS AND TABLES IN MAPLE:

Supplement to the Maple User's Manual

Stephen M. Watt

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

ABSTRACT

Maple is a language for symbolic mathematical computation under development at the University of Waterloo. The *Maple User's Manual* * describes Maple as of December 1982 (version 2.2). Since then, the language has been extended to have array and table data types. This report is a reference guide for using these new data types.

* K.O. Geddes, G.H. Gonnet and B.W. Char, *Maple User's Manual,* Second Edition, University of Waterloo Research Report CS-82-40.

CONTENTS

PREFACE

Before describing arrays and tables, it is necessary to mention changes concerning two data types that existed previously in Maple:

(i) An expression sequence is now syntactically valid as input wherever an expression may appear.

(ii) Subscripted names are no longer of type 'name', but belong to a new type, 'indexed'.

These features are described below.

Expression Sequences

An expression sequence is a valid expression in its own right. The comma operator (',') is used to concatenate expressions to form an expression sequence. 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
[c];                             yields [A, B, C, D, A, B, C, D]
nops([a]);                       yields 4
f := proc() param(6) end;
f(c);                            yields B
```

Indexed Expressions

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

<name>[<expression sequence>] .

For an indexed expression, the zero*th* 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>] .

Examples:

nops(T[t,x,y,z]);	yields 4
nops(S[ ]);	yields 0
op(2, T[t,x,y,z]);	yields x
op(0, A[sin(x)+ t]);	yields A
op(0, B[1,2][3,4]);	yields B[1,2]

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. 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 *ith* range gives the bounds on the *ith* 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 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.

2. CREATING TABLES

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 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])

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)
```

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 the Appendix.

3. TABLE COMPONENTS

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. 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

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])
```

4. TABLES AS OBJECTS

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 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);
```

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])
```

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.

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. INDEXING FUNCTIONS

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 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.

5.2. 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
```

5.3. 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. LIBRARY FUNCTIONS

6.1. array (indexing_function, init_list, lo1..hi1, lo2..hi2, ...)

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])

6.2. convert (expr, typename)

With the implementation of tables in Maple, an addition has been made to the Maple library for conversion to type 'array'.

When *convert* is called and 'typename' is 'array', an attempt is made to construct an array from the input expression. 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 parameters are given after 'typename' in the call to *convert*, then they are used as the index bound. When 'expr' is a list, then 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'.

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])
table(sparse,[1=x, 100=y]);
convert(", array);                yields  array(sparse,1..100,[(1)=x,(100)=y])
```

6.3. 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])]
```

6.4. 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;
```

6.5. 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;
```

6.6. 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])

6.7. type (expr, typename)

The two names 'table' and 'array' are known to the *type* function to check for tables and arrays. Because arrays are also tables, type(A,table) is true if A is an array.

Indexed expressions are recognized by the type function using the type name 'indexed'.

APPENDIX

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;
```