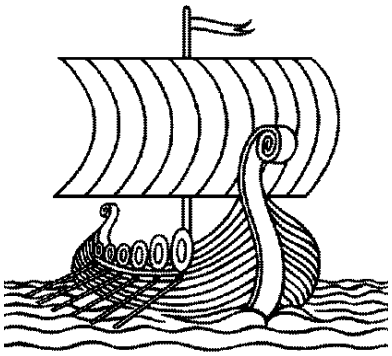


Q'Nial

The Nial Dictionary

Version 6.3

August 2005

Index of Entry Classes

abs

Class: arithmetic operation

Property: unary pervasive

Usage: `abs A`

See Also: opposite

The operation *abs* produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	corresponding integer
integer	absolute value
real	absolute value
character	fault ?A
phrase	fault ?A
fault	argument A

If the argument is the largest negative integer for the computer, the result is the corresponding positive real number because the corresponding positive integer is too large for representation as an integer.

```
     abs  1  -2  3.5  `a  "abc  ??error
1 2 3.5 ?A ?A ?error
```

The operation *abs* can be used to convert a boolean value to an integer. Its primary use is in the numeric domains, where it is used to give a positive number measuring the size of a number.

Equations

```
abs abs A = abs A
abs opposite A = abs A
```

accumulate

Class: reduction transformer

Usage: `ACCUMULATE f A`

See Also: reduce

The transformer *accumulate* is similar to the concept of scan in APL. It computes the partial reductions of the initial lists of the items of *A*, using *reduce f* (a right-to-left reduction) to compute each partial result. (A reductive operation is one which when applied to an array having a number of items returns a single result.)

In the example below, the **initial lists** of *Z* are:

```
[13, 13 39, 13 39 92, 13 39 92 45];
```

and the right-to-left reduction of *f* on *Z* is equivalent to:

```
     [f 13, 13 f 39, 13 f (39 f 92), 13 f (39 f ( 92 f 45))]

     Z := 13 39 92 45 ;
     ACCUMULATE pass Z
+--+-----+----------+---------------+
|13|13 39|+--+-----+|+--+----------+|
|  |     ||13|39 92|||13|+--+-----+||
|  |     |+--+-----+||  ||39|92 45|||
|  |     |          ||  |+--+-----+||
|  |     |          |+--+----------+|
+--+-----+----------+---------------+
```

The first example shows that accumulate builds the result values with a right-to-left reduction.

```
     ACCUMULATE max Z
13 39 92 92

     ACCUMULATE min Z
13 13 13 13

     ACCUMULATE sum (count 10)
1 3 6 10 15 21 28 36 45 55

     ACCUMULATE minus (count 10)
1 -1 2 -2 3 -3 4 -4 5 -5
```

Useful applications of *accumulate* include the *accumulate sum* transform used to compute a running sum, and the *accumulate minus* transform which gives an alternating sum.

```
     ACCUMULATE or oollooll
oolllllll

     S := '  black and white'
  black and white

     B := not (S match ` )
oolllllollloll lll

     ACCUMULATE or B sublist S
black and white
```

The *accumulate or* transform produces a bitstring that can be used with *sublist* to drop leading items that fail to meet a predicate.

Definition

```
ACCUMULATE IS TRANSFORMER f OPERATION A {
        heads IS OPERATION A {
           count tally A EACHLEFT take list A};
        EACH REDUCE f heads A }
```

Equations

```
   shape ACCUMULATE f A = shape A
   ACCUMULATE f solitary A = solitary A
   ACCUMULATE f single A = single A
   ACCUMULATE f Null = Null
   ACCUMULATE f[A,B,C]=[A,A f B,A f(B f C)]
```

action

Class: syntax

See Also: definition, expression sequence, external declaration, comment

An **action** is the construct that is entered in the interactive loop of the Q'Nial interpreter or accepted as an input unit within the operation loaddefs:

```
action ::= definition-sequence
   | expression-sequence
   | external-declaration
   | remark

definition-sequence ::= definition
   { ; definition } [ ; ]

remark ::= # < any text >
```

If an action is a definition-sequence, its definitions are installed in the global environment.

If an action is an expression-sequence, it is executed and a value is returned. The value returned by an expression-sequence is displayed on the screen unless it is the fault ?noexpr.

An external-declaration assigns a role to a name in the global environment so that the name can be used in other definitions before it is completely specified.

A remark is an input to the Q'Nial interpreter that is not processed. It begins with a line that has the symbol # as the first non-blank character in the line. In direct input at the top level loop, a remark ends at the end of the line unless a backslash symbol (\ ) is used to extend the line. In a definition file, a remark ends at the first blank line. A remark cannot appear within a definition or expression-sequence.

across

Class: recursion transformer

Usage: `ACROSS [endf, parta, joinf] A`

See Also: recur, down

ACROSS is a general recursion transformer for traversing the length of an array. It has three operation arguments: *endf* is applied to the end argument before starting to build the result, *parta* computes the left value from the argument, which is stacked, and *joinf* combines the left and right values as the recursion unwinds.

```
     ACROSS [ 0 first, pass, plus ] 3 4 5 6
18
     reshape [ shape, ACROSS  [pass, pass, CONVERSE append ] ] 2 15 12 4
4 12 15 2
```

Definition

```
ACROSS IS TRANSFORMER endf parta joinf OPERATION A {
        A := list A;
        Res := endf void last A;
        FOR I WITH reverse tell tally A DO
                Res := parta A@I joinf Res
        ENDFOR }
```

Equations

```
ACROSS [endf, parta, joinf] A = RECUR [empty, endf, parta first, joinf, rest ] list A
sum A = ACROSS [ 0 first, pass, plus ] A
reverse A = reshape [ shape, ACROSS  [pass, pass, CONVERSE append ] ] A
EACH f A = reshape [ shape, ACROSS [ Null first, f, hitch ] ] A
```

address

Class: concept

See Also: array, pick, indexing

An **address** is an integer or a list of integers describing the location of an item in an array. An address uses 0-origin counting, i.e. the first position in a list is at address 0. All the addresses of an array can be stored in an array of the same rectangular structure as the array itself. Such an array is called the **grid** of an array. Consider the following example:

```
             GRID OF THE ARRAY                 ARRAY
+-----+-----+-----+-----+-----+-----+ +---+---+---+---+---+---+
|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+| |499|434|122|770|733|890|
||0|0|||0|1|||0|2|||0|3|||0|4|||0|5|| +---+---+---+---+---+---+
|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+| |660|160| 32|808|240|584|
+-----+-----+-----+-----+-----+-----+ +---+---+---+---+---+---+
|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+| |473|235|164|496|808|966|
||1|0|||1|1|||1|2|||1|3|||1|4|||1|5|| +---+---+---+---+---+---+
|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+| |156|205| 34|576|236|454|
+-----+-----+-----+-----+-----+-----+ +---+---+---+---+---+---+
|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+|
||2|0|||2|1|||2|2|||2|3|||2|4|||2|5||
|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+|
+-----+-----+-----+-----+-----+-----+
|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+|
||3|0|||3|1|||3|2|||3|3|||3|4|||3|5||
|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+|+-+-+|
+-----+-----+-----+-----+-----+-----+
```

The array on the right is a 4 by 6 table of numbers. It has 24 items arranged along two directions of length 4 and 6 respectively. The address of 496 is [2,3].

The addresses of a *list* are integers: 0, 1, etc. For convenience, in situations where an address of a list is expected, a solitary integer is treated as an integer.

The address of a *single* is the empty list *Null*.

Allbools

Class: measurement operation

Property: predicate

Usage: `allbools A`

See Also: allints, allreals, allchars, allnumeric

The operation *allbools* tests whether or not *A* is a nonempty array of boolean items. It returns *true* if *A* is not empty and all the items of *A* are boolean atoms, *false* otherwise.

```
     allbools lool
l
     allbools 1 0 1
o
     allbools Null
o
```

Definition

allbools IS OPERATION A { and EACH isboolean A and not empty A }

allchars

Class: measurement operation

Property: predicate

Usage: `allchars A`

See Also: allbools, allints, allreals, allnumeric

The operation *allchars* tests whether or not *A* is a nonempty array of character items. It returns *true* if *A* is not empty and all the items of *A* are character atoms, *false* otherwise.

```
     allchars 'apple pie'
l
     allchars "today
o
     allchars Null
o
```

Definition

```
allchars IS OPERATION A { and EACH ischar A and not empty A }
```

allin

Class: set-like operation

Properties: binary, predicate

Usage: `A allin B   allin A B`

See Also: in, like, notin

The operation *allin* tests whether or not all the items of array *A* are also items of array *B*. The result is *true* if the test holds and *false* if it does not.

```
     3 5 7 allin count 10
1
     'ae' allin 'where are you?'
1
     'where' allin 'hear'
0
```

The items of *A* and *B* are compared for exact equality, including type. The items being compared may be atoms or they may be any other array.

Definition

```
     allin IS OPERATION A B { and ( A EACHLEFT in B ) }
```

Equation

```
   A allin B = (list A) allin (list B)
```

Allints

Class: measurement operation

Property: predicate

Usage: `allints A`

See Also: allbools, allreals, allchars, allnumeric

The operation *allints* tests whether or not *A* is a nonempty array of integer items. It returns *true* if *A* is not empty and all the items of *A* are integer atoms, *false* otherwise.

```
     allints lool
0
     allints 1 0 25
1
     allints Null
0
```

Definition

```
allints IS OPERATION A { and EACH isinteger A and not empty A }
```

allnumeric

Class: measurement operation

Property: predicate

Usage: `allnumeric A`

See Also: allints, allreals, allbools, numeric

The operation *allnumeric* tests whether or not *A* is a nonempty array with all atoms of the same numeric type. It returns *true* if *A* is not empty and all items are boolean atoms, or all are integer atoms or all are real atoms, *false* otherwise.

```
     allnumeric 3.5 -2.97
l
     allnumeric l 45 3.78
o
     allnumeric Null
o
```

Definition

```
allnumeric IS OPERATION A { allbools A or allints A or allreals A }
```

allreals

Class: measurement operation

Property: predicate

Usage: `allreals A`

See Also: allbools, allints, allchars, allnumeric

The operation *allreals* tests whether or not *A* is a nonempty array of real items. It returns *true* if *A* is not empty and all the items of *A* are real atoms, *false* otherwise.

```
     allreals 2 2.5 47.9
o
     allreals 2.5 -7.8 27.3
l
     allreals Null
o
```

Definition

allreals IS OPERATION A { and EACH isreal A and not empty A }

and

Class: logic operation

Properties: multi pervasive, reductive

Usage: `and A   A and B`

See Also: or, not

The operation *and* applied to a boolean array *A* does the boolean product of its items. If all items of *A* are *true*, the result is *true*; otherwise it is *false*. In binary form and implements the and-connective of logic.

If *A* is a simple array and has a non-boolean item, the result is the logical fault *?L*. The operation extends to non-simple arrays by the multi pervasive mechanism.

```
     l and l
l

     and lllllloll
o

     lloo and lolo
looo
```

And is a reductive operation in that it reduces an array of booleans to a single boolean. If its argument is a pair of bitstrings or a list of bitstrings, *and* is applied to bits in corresponding positions of each item of the argument producing a bitstring of the same length as one of the items of its argument.

Equations

```
A and B = B and A
not and A =f= or EACH not A
and Null = True
```

append

Class: construction operation

Property: binary

Usage: `A append B   append A B`

See Also: hitch, link

The operation *append* attaches *B* to the end of the list of items of *A*. It returns a list of length one greater than the *tally* of *A*.

```
     (2 3 4) append (5 6 7)
+-+-+-+-----+
|2|3|4|5 6 7|
+-+-+-+-----+

     append '' 'Wow'
+---+
|Wow|
+---+
```

The first example shows list (5 6 7) appended to list (2 3 4). The second example shows that if an array is appended to an empty list, the result is the solitary of the second argument. If *append* is applied to an array that is not a pair, a fault is returned.

Definition

```
     append IS OPERATION A B { A link solitary B }
```

Equations

```
A append B = ( list A ) append B
Null append A = solitary A
A append B = list (A append B)
```

appendfile

Class: file operation

Usage: `appendfile Filename A`

See Also: putfile, getfile

The operation *appendfile* writes a list of character arrays *A* to the end of the file named *Filename*, using one line of the file for each row in the items of *A*. If the file does not exist, it is created by the operation. The file must not be open.

Appendfile is similar to operation *putfile* except that *putfile* overwrites an existing file. *Getfile* reads a file containing text records.

```
     putfile "Mailing ['1','2','3'];
     appendfile "Mailing ['Mr. Jones','123 Main St','New York NY 12345'];
     getfile "Mailing
+-+-+-+---------+----------+----------------+
|1|2|3|Mr. Jones|123 Main St|New York NY 12345|
+-+-+-+---------+----------+----------------+
```

Definition

```
     appendfile IS OPERATION Filename A {
        Fnum := open Filename "a ;
        ITERATE (Fnum writefile) (link EACH rows A) ;
        close Fnum ; }
```

apply

Class: evaluation operation

Usage: `Op apply A   apply Op A`

See Also: eval, execute, value, cast

The operation *apply* carries out the application of the operation represented by the array *Op* to array *A*. *Op* may be a phrase or string that names a predefined or user-defined operation or it may be the *cast* of an operation expression.

If the operation is represented by a phrase or string giving its name, the name is sought in the environment where *apply* is being used.

```
     apply "second (3 4 5)
4
```

The operation to be applied may also be represented by the *parse tree* or *cast* for the operation expression. If the cast is a name, it is sought in the environment where the cast appears. In the following example, *!first* is the cast of the operation *first*.

```
     (!first apply 3 4 5)
3

     apply !(first rest) (3 4 5)
4
```

As seen in the second example, an operation expression does not have to be named in order to be used with *apply*. The entire first example is enclosed in parentheses to avoid conflict with the notation for executing a host command.

arccos

Class: scientific operation

Property: unary pervasive

Usage: `arccos A`

See Also: cos, cosh, arcsin, pi

The operation *arccos* is the inverse of *cos* in the numeric domain. It produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	arccosine of the corresponding real
integer	arccosine of the corresponding real
real	the angle B in radians such that cos B = A, if A is in the range -1 to 1; otherwise the fault ?arccos
character	fault ?arccos
phrase	fault ?arccos
fault	argument A

```
     arccos 1 -2 0.35
0. ?arccos 1.21323

     arccos `a "abc ??error
?arccos ?arccos ?error
```

Definition

```
     arccos IS OPERATION A ( Pi / 2.0 - arcsin A )
```

Equation

```
   cos arccos cos A = cos A (within roundoff error)
```

arcsin

Class: scientific operation

Property: unary pervasive

Usage: `arcsin A`

See Also: sin, sinh, arctan, pi

The operation *arcsin* is the inverse of *sin* in the numeric domain. It produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	arcsine of the corresponding real
integer	arcsine of the corresponding real
real	the angle B in radians such that sin B = A, if A is in the range -1 to 1; otherwise, the fault ?arcsin
character	fault ?arcsin
phrase	fault ?arcsin
fault	argument A

```
     arcsin l -2 0.35
1.5708 ?arcsin 0.357571

     arcsin `a "abc ??error
?arcsin ?arcsin ?error

     arcsin l 1 .5
1.5708 1.5708 0.523599
```

Equation

```
   sin arcsin sin A = sin A   (within roundoff error)
```

arctan

Class: scientific operation

Property: unary pervasive

Usage: `arctan A`

See Also: tan, tanh, arcsin, pi

The operation *arctan* is the inverse of *tan* in the numeric domain. It produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	arctangent of the corresponding real
integer	arctangent of the corresponding real
real	angle B in radians where tan B = A
character	fault ?arctan
phrase	fault ?arctan
fault	argument A

```
     arctan 1 -2 0.35
0.785398 -1.10715 0.336675

     arctan `a "abc ??error
?arctan ?arctan ?error

     arctan 2. 1 1 .5
1.10715 0.785398 0.785398 0.463648
```

Equation

```
   tan arctan tan A = tan A   (within roundoff error)
```

argument

Class: concept

See Also: operation

An **argument** is an array value supplied to an operation. All operations in Nial take a single array as the argument, but binary operations expect that argument to be a pair. When an operation is used in infix syntax, then the two array values on the left and right are combined into a pair to be supplied as the argument to the operation.

The argument to an operation-form is treated specially. If the operation-form has only one formal parameter then the argument is assigned to the parameter. If it has two or more formal parameters, then the argument is checked to see if it has the same number of items as there are parameters. If so, the items are assigned to the parameters. If not, the result is the fault *?op_parameter*.

array

Class: concept

See Also: item, address, dimensions

The data objects of Nial are nested rectangular arrays. Atomic data objects such as numbers and characters are included within this description by virtue of an atom being considered as a self-containing array object with no dimensions.

Atomic Arrays

There are six types of atoms in Nial. They are *boolean*, *integer*, *real*, *character*, *phrase* and *fault*. The first three are **numeric** types and are used for arithmetic operations. The last three are **literal** types and are used for text and symbol manipulation. All six types of atoms are used in comparisons.

Rectangularity Structure

An **array** is a collection of data objects having its **items** held at locations in a rectangular structure. The items are viewed as being at locations that are positioned relative to a set of directions at right angles to each other. The items may be arranged along zero, one, two or more directions. For example, the following

array is a 4 by 6 table of numbers. It has 24 items arranged along two directions of length 4 and 6 respectively.

```
+---+---+---+---+---+---+
|499|434|122|770|733|890|
+---+---+---+---+---+---+
|660| 32|808| 24|  5|473|
+---+---+---+---+---+---+
|499|434|122|770|733|890|
+---+---+---+---+---+---+
|660| 32|808| 24|  5|473|
+---+---+---+---+---+---+
```

The items of an array are themselves arrays. Thus, an array can have an arbitrarily deep nesting structure.

assign

Class: evaluation operation

Usage: `Nm assign A   assign Nm A`

See Also: update, place, value, assign expression

The operation *assign* assigns the value of *A* to the variable named by *Nm*. *Nm* may be a phrase or string (e.g. *"X* or *'X'*), in which case the named variable is sought in the current environment; or it may be the cast of a name (e.g. *!X*), in which case it is sought in the environment in which it was cast. If the variable is found, its value is replaced by the array *A*. Otherwise, a variable with that name is created in the global environment and given *A* as its value.

Any phrase may be used as a variable name with the operation *assign*. Thus, it is possible to assign a value to names which are invalid identifiers. The associated value can be retrieved using the operation *value*. The result of the operation is *A*.

```
     assign "X (2 3 4)
2 3 4

     (!Var assign 2 3 4)
2 3 4
```

In the first example, the variable *X* is assigned the list 2 3 4. If this is done in an environment where *X* is a local variable, the local variable is updated. If *X* does not exist, it is created in the global environment.

In the second example, a variable *Var* must exist in the environment where the cast is done. The cast can be formed in a more global environment and passed into an operation as a parameter (This technique is called **by-variable** parameter passing).

assign expression

Class: syntax

See Also: variable, expression, indexing, assign

```
assign-expression ::=
   { variable }+ := expression
   | indexed-variable := expression
```

An **assign-expression** assigns an array value to one or more variables at the time of evaluation of the assign expression. The semantics of an assign expression is interpreted in two stages: when the expression is analyzed (parsed) and when it is executed.

During the parse of the assign-expression appearing in a block, each name on the variable list is sought in the local environment. If the name exists in the local environment, the assignment affects the local association. If a name does not exist in the local environment and no reference has been made to a nonlocal variable with the same name, a local variable is created in the block. An assign-expression parsed in the global environment creates a global variable if a variable with that name does not already exist.

When an assign expression is executed, the expression on the right of the assignment symbol (:=) is evaluated. If the variable list on the left has only one name, the value of the expression is assigned to that variable. That is, the value is associated with that name.

If the variable list has several names, the items of the value are assigned to the variables in the order in which they appear. If the number of items does not match the number of variables, the fault ?assignment is returned as the value of the assign-expression. Otherwise, the value of the assign-expression is the value of the expression on the right.

When an indexed-variable is used on the left in an assign-expression, the parts of the array associated with the variable at the locations specified by the index are replaced by the values of the expression on the right.

If the index expression for an indexed-variable assignment specifies a number of locations (at-all or slice indexing), there are two cases: if the value on the right is a single, the item of the single is placed in each location; otherwise, the value on the right must have the same number of items as the index expression indicates and the corresponding locations are updated with the items of the array value.

atlas

Class: syntax

See Also: operation

```
atlas ::= [ operation-expression { , operation-expression } ]
```

An **atlas** is an operation made up of a list of component operations. The result of applying an atlas is a list of the same length as the atlas. Each operation in the atlas is applied in turn to the argument resulting in an array value that becomes the item of the result list in the corresponding position. An atlas is used by the transformers *FORK, INNER* and *TEAM*.

atomic

Class: structure testing operation

Property: predicate

Usage: `atomic A`

See Also: simple, leaf

The operation *atomic* tests whether or not its argument is an atom. It returns *true* if it is and *false* if it is not.

```
     atomic 3.5
1
     atomic "hello
1
     atomic 'hello'
0
```

The examples illustrate that a number and a phrase are atoms and that a string is not an atom.

An atom is a primitive concept in array theory and Nial. Atoms are distinguished from other arrays by the property that they are self nesting. The definition of atomic is based on this property.

Definition

```
     atomic IS OPERATION A { first A = A }
```

Equations

```
   atomic A <==> single A equal A
   and EACH atomic A = simple A
```

axes

Class: measurement operation

Usage: `axes A`

See Also: valence, tell

The operation *axes* generates a list of axis numbers for the array *A* counting from zero.

```
     axes tell 3 4 5
0 1 2
```

Definition

```
     axes IS OPERATION A { tell valence A }
```

Equation

```
   tally axes A = tally shape A
```

binary

Class: operation property

Usage: `A f B   f A B`

See Also: binary pervasive

An operation is said to *binary* if it must have exactly two items in its argument. Many of the built-in operations of Nial are binary. They can be used in both an infix and prefix manner. If a binary operation *f* is used in infix syntax then the arguments on each side of *f* are treated as the two items of its argument.

```
     3 reshape 5
```

In prefix usage, *f* can precede a single array with two items, or an explicit pair formed with strand notation or bracket-comma notation.

```
     X := 3 5;
     reshape X
5 5 5

     reshape 3 5
5 5 5

     reshape [3,5]
5 5 5
```

binary pervasive

Class: operation property

Usage: `A f B   f A B`

See Also: binary, unary pervasive, multi pervasive, pervasive, eachboth, pack

Each operation *f* in this class maps a pair of atoms to an atom.

A binary pervasive operation maps two arrays having identical structure to one with the same structure, mapping each pair of corresponding atoms by the function's behaviour on pairs of atoms.

All of the binary operations of arithmetic and logic are binary pervasive.

If a binary pervasive operation is applied to a pair of arrays that do not have the same shape, the effect is to build a conformable pair by replicating an atom or solitary item of the pair to the shape of the other item. If both items are of unequal shape and if both items are made up of more than one item, the fault *?conform* is returned. The replication of an argument with one item provides binary pervasive operations with a scalar extension capability. For example,

```
     3 4 5 6 - 5 = (3 4 5 6 - 5 5 5 5)
1
```

If a binary pervasive operation is applied to an array that is not a pair, a fault is returned.

The following table lists the binary pervasive operations.

Operation	Function
divide	division of numbers
gt	greater than comparison
gte	greater than or equal comparison
lt	less than comparison
lte	less than or equal comparison
match	equality of atoms without type coercion
mate	equality of atoms with type coercion
minus	subtraction of numbers
mod	remainder on division of integers
plus	addition of numbers
quotient	quotient on division of integers
times	multiplication of numbers divide division of numbers

Equations

```
A f B = A EACHBOTH f B
A f B = EACH f (A pack B)
shape (A f B) = shape pack A B
```

blend

Class: nesting restructuring operation

Property: binary

Usage: `A blend B  blend A B`

See Also: split, mix, rank

The operation *blend* combines the top two levels of an array *B* into a single level blending the axes of the second level of *B* into the combined level according to the axis numbers given in *A*. The items of *B* must be of the same shape.

If the array *B* is equivalent to *A split C* for some array *C*, the result of *A blend B* is *C*.

The tally of *A* is the valence of an item of *B*. The items of *A* indicate where the axes of the items of *B* are placed in the shape of the result.

```
     C := (2 4 3 reshape count 24) ;
     B := 2 0 split C
+----+----+----+-----+
|1 13|4 16|7 19|10 22|
|2 14|5 17|8 20|11 23|
|3 15|6 18|9 21|12 24|
+----+----+----+-----+

     C := 2 0 blend B
 1  2  3   13 14 15
 4  5  6   16 17 18
 7  8  9   19 20 21
10 11 12   22 23 24
```

In the example, *B* is a list of four tables of shape 3 2 created by a *2 0 split*. The result of *2 0 blend B* is the array of valence 3 that is the second argument to *split*.

Equations

```
SORT <= list I = axes A and not empty A ==> I blend (I split A) = A
equal EACH shape A and not empty A ==> (valence A + axes first A) blend A = mix A
A blend B =f= [Null,A] PARTITION first B
```

block

Class: syntax

See Also: local environment, scope of a variable, nested definition

```
block ::=
   { [ LOCAL { identifier-sequence }+ ; ]
[ NONLOCAL { identifier-sequence }+ ; ]
[ definition-sequence ; ]
  expression-sequence }
```

A **block** is a scope-creating mechanism that permits an expression-sequence to be created so that it has local definitions and variables which are visible only inside the block. A block may appear as a primary-expression or as the body of an operation-form.

A local environment is a collection of associations that are known within a limited section of program text. These limited sections are formed by blocks, operation-forms and transformer-forms. A name that has a local association in one of these forms is said to have local scope.

If the definition appears within a block, the association is made in the local environment. Otherwise, the association is made in the global environment and assigns a role to the name as representing that kind of expression.

If a block is used as a primary-expression, the local environment created by a block is determined by the block itself. If it is the body of an operation-form, the local environment includes the formal parameter names of the operation-form as variables.

Local and Nonlocal Declaration

The identifiers included in the local and nonlocal declarations are declared to be variables. Both forms of declarations are optional, but if both are given, local declarations must be made first. If the block is the body of a globally defined operation-form or expression, a nonlocal declaration effectively declares its

variables as global ones.

A block delimits a local environment. It allows new uses of names which do not interfere with uses of those names outside the block. For example, within a block, a predefined operation name can be redefined and used for a different purpose. Only the reserved words of Q'Nial cannot be reused in this fashion.

Definitions that appear within the block have local scope. That is, the definitions can be referenced only in the body of the block. Variables assigned within the block may or may not have local scope, depending on the appearance of a local and/or a nonlocal declaration. If there is no declaration, all assigned variables have local scope. Declaring some variables as local does not change the effect on undeclared variables that are used on the left of assignment. They are automatically localized.

If a nonlocal declaration is used, an assigned name that is on the nonlocal list is sought in surrounding scopes. If the name is not found, a variable is created in the global environment.

bracket-comma notation

Class: syntax

See Also: strand notation

A list may be constructed directly by using **bracket-comma** notation. In this notation, the items of a list are separated by commas and the list is bounded by square brackets. If an item is omitted before or after a comma, then the fault value *?noexpr* is used for the value of the missing item. The notation denotes the *Null* if their are no items, and a solitary if there is only one item.

```
     [2,3 4,5]
+-+---+-+
|2|3 4|5|
+-+---+-+

     [,4 5]
+-------+---+
|?noexpr|4 5|
+-------+---+

     [] = Null
1

     ['hello world']
+-----------+
|hello world|
+-----------+

     [3,[4,5,6],7]
+-+-----+-+
|3|4 5 6|7|
+-+-----+-+
```

break

Class: system expression

Usage: `Break`

See Also: breakin, debugging, callstack, step, next, resume

The execution of *Break* causes the interpreter to interrupt normal execution in an expression sequence and

to display the current callstack. It then prompts for input with the prompt --> followed by the default command in brackets. The visible environment is that of the expression in which the break occurs. Thus, it is possible to examine the values of local variables in break mode.

At a break you can type any expression to inspect the value of a variable or to see a portion of its value. The operation *see* can also be used to view any of the definitions in the environment.

The debugging capability allows one to step forward in expression sequences using commands *step*, *stepin*, *next* or *toend* to control whether you step into or over other definitions or to the end of a loop or a definition. The command *resume* ends the break and normal execution is restarted.

In console versions of Q'Nial, break mode can also be entered by typing <Ctrl B> in response to the prompt for a *read* or *readscreen* operation or *Readchar* expression when in windows mode. In this case, after *resume* or *step* is entered, Q'Nial returns to the *read* or *readscreen* operation and awaits the user input. Break mode cannot be entered by typing <Ctrl B> in *editwindow* or when *setinterrupts* has been used to inhibit interrupts.

```
     foo is op A { Break; A }
     foo 3
------------------------------------------------------------
    Break debug loop: enter debug commands, expressions or
      type: resume    to exit debug loop
      <Return> executes the indicated debug command
    current call stack :
foo
------------------------------------------------------------
?.. A
-->[stepv]
```

In the example, the operation *foo* has a *Break* which is executed when *foo* is applied to 3. If *Return* is pressed at the prompt the expression *A* is evaluated and 3 is displayed followed by another prompt.

breakin

Class: system operation

Usage: `breakin Defname [Mode]`

See Also: break, debugging, breaklist

The operation *breakin* installs a break point prior to the first executable expression in a definition named by the string or phrase *Defname*. The named definition must be an expression or an operation form. The effect is that when the definition is executed a break interrupt occurs prior to the execution of the first expression in the expression sequence in the body of the definition.

The optional argument *Mode* is provided to set the internal flag explicitly. If it is omitted, the internal flag value is toggled. All breakin flags are initially *false*. The value of the breakin flag is retained if the definition is replaced by a *loaddefs* so that editing a definition and reloading it does not change its breakin status. However, if definitions are reloaded using a *restart* then the breakin status is set to *false*.

The operation returns the previous setting. If the result of *breakin* is assigned to a variable, the previous setting can be restored later. The names of definitions that have breakin set can be viewed by the expression *Breaklist*.

```
     library "labeltab

     see "labeltable
```

```
labeltable IS OPERATION Corner Rowlabel
   Columnlabel Table {
   % Combine the corner label with the column
      labels for first line;
   Firstrow := Corner hitch Columnlabel;
   % Hitch the row labels to the rows of
      the table;
   Labeledrows := Rowlabel EACHBOTH hitch
      rows Table;
   % Hitch the first row of labels to the labeled
      rows and mix them;
   mix (Firstrow hitch Labeledrows) }

     X gets count 3
1 2 3

# execute labeltable without break;

     labeltable "TIMES X X (X OUTER times X)
TIMES 1 2 3
    1 1 2 3
    2 2 4 6
    3 3 6 9

# set breakin and execute again

     breakin "labeltable
o

     labeltable "TIMES X X (X OUTER times X)

------------------------------------------------------------
    Break debug loop: enter debug commands, expressions or
      type: resume    to exit debug loop
      <Return> executes the indicated debug command
    current call stack :
labeltable
------------------------------------------------------------
?.. Firstrow := Corner hitch Columnlabel
-->[stepv] TIMES 1 2 3
?..   % Hitch the row labels to the rows of the table
-->[stepv] resume
TIMES 1 2 3
    1 1 2 3
    2 2 4 6
    3 3 6 9
```

To clear all breaks use:

```
     EACH breakin Breaklist
```

breaklist

Class: system expression

Usage: `Breaklist`

See Also: break, breakin, debugging

The execution of *Breaklist* prints out a list of the definition names for which the breakin flag has been set.

Its main use is to assist in clearing the breaks as debugging proceeds.

```
     Breaklist
labeltable
```

To clear all breaks use:

```
EACH breakin Breaklist
```

bycols

Class: apply-by-partition transformer

Usage: `BYCOLS f A`

See Also: rank, partition

The transformer *BYCOLS* applies an operation *f* to the columns of a table *A*, where *f* is an operation that maps lists to lists.

```
     setformat '%5.3f';
     BYCOLS (SORT <=)(5 6 reshape random 30)
0.726 0.009 0.210 0.385 0.123 0.119
0.851 0.448 0.384 0.476 0.193 0.384
0.880 0.449 0.536 0.603 0.455 0.498
0.882 0.729 0.631 0.819 0.639 0.521
0.911 0.893 0.810 0.961 0.940 0.672
```

The example generates a random table and then sorts each column.

Definition

```
     BYCOLS IS TRANSFORMER f OPERATION A { transpose (1 RANK f transpose A) }
```

Equation

```
   shape f A = shape A ==> shape BYCOLS f A = shape A
```

bye

Class: system expression

Usage: `Bye`

See Also: continue, save, restart

The expression *Bye* is used to terminate a session of Q'Nial and to return to the host environment. The current workspace is not saved.

bykey

Class: applicative transformer

Usage: `K BYKEY f A   BYKEY f K A`

See Also: each

The transformer *BYKEY* applies an operation *f* to lists gathered from *A*, where each list is made up of items of *A* that have the same value in the corresponding position in *K*. The result is a list with as many items as there are unique items in *K*.

```
     Str := 'a stitch in time saves nine';

     cull Str
a stichnmev

     Str BYKEY tally Str
2 5 3 2 4 1 1 3 1 3 1

     Keys := 3 5 3 7 2 5 4 2 1 3 4;
     Data := 23.1 14.2 13.5 18.9 22. 98. 3.5 28.7 19.3 16.5 43.2;

     Keys BYKEY sum Data
53.1 112.2 18.9 50.7 46.7 19.3

     average IS OP A { sum A / tally A }

     Keys BYKEY average Data
17.7 56.1 18.9 25.35 23.35 19.3
```

In the first example *BYKEY tally* is used to count the frequency of the letters in the string *Str*, giving the counts in the order of *cull Str*. In the remaining examples, the values in *Data* corresponding to equal items in *Keys* are added and averaged in the two uses of *BYKEY*.

Definition

```
     BYKEY IS TRANSFORMER f OPERATION K A {
        Keys gets cull A;
        IF simple Keys THEN
           Patterns := Keys EACHLEFT match A;
        ELSE
           Patterns := Keys EACHLEFT EACHRIGHT = A;
        ENDIF;
        EACH f (Patterns EACHLEFT sublist B) }
```

Equations

```
   tally BYKEY f K A = tally cull K
   valence BYKEY f K A = 1
```

byrows

Class: apply-by-partition transformer

Usage: `BYROWS f A`

See Also: bycols, partition

The transformer *BYROWS* applies an operation *f* to the rows of a table *A*, where *f* is an operation that maps lists to lists.

```
     BYROWS reverse (5 6 reshape count 30)
 6  5  4  3  2  1
12 11 10  9  8  7
18 17 16 15 14 13
24 23 22 21 20 19
30 29 28 27 26 25
```

The example reverses the rows of the generated table.

Definition

```
     BYROWS IS TRANSFORMER f OPERATION A { 1 RANK f A }
```

Equation

```
shape f A = shape A ==> shape BYROWS f A = shape A
```

by-variable

Class: concept

See Also: argument, assign

One use of the operation *assign* is to mimic a **by-variable** form of parameter passing in place of Nial's by-value form. The result depends on what kind of name is provided, a phrase or a cast. If the name is provided as a phrase, the variable that is selected is determined by *assign* when it does the assignment by looking first in the local environment and then in the surrounding ones. If the name is provided as a cast, the variable selected is the one that exists at the point where the cast is formed. Thus, by-variable parameter passing is achieved by using the cast of the variable as an argument in the call. In the body of the operation the formal parameter is assigned using *assign* and evaluated using *value*.

```
     foo is op A Nm {
       B := A + value Nm;
       Nm assign (B + sum count 5); }

     X := 100;
     foo 1000 "X;
     X
1115
```

callc

Class: system operation

Usage: `P Op callc A`

See Also: calldllfun, user primitives

The operation *callc* provides an interface to a C-coded operation *Op* and executes it with arguments *A*. The operation is one of several in a package called *P*.

Q'Nial Version 6.2 for Unix provides the CallC capability. This version of Q'Nial can be extended by adding packages and operations written in C and linked into the interpreter.

calldllfun

Class: system operation

Usage: `calldllfun Nm Arg0 ... ArgN`

See Also: registerdllfun, dlllist, user primitives, callc

A Prototype DLL calling interface has been added to the Windows versions of Q'Nial (Console, GUI and DLL version). This ability allows NIAL code to call external routines in DLL libraries. These libraries can be user written, part of the Operating System, or 3rd party DLLs.

The DLL interface routines manage an internal table that keeps track of the currently installed DLLs and their argument types and result type. This table also tracks if the actual library has been loaded (by a call to *calldllfun*). The table is preserved as part of the workspace, and once DLL functions have been registered in a workspace, they are still available if the workspace is saved and subsequently loaded. The CallDLL facility assures that the DLL files are loaded and unloaded appropriately.

Every time a workspace is saved all DLL libraries are unloaded and the internal table is adjusted to reflect

that. When a workspace is loaded, none of the DLL libraries will be loaded until a specific call to *calldllfun* brings in the DLL. This process assures that resources are not wasted on registered DLL calls that are never used.

The *registerdllfun* routine must always be called first to enter DLL function into the internal mapping table. This table keeps track of the name and DLL file for the routine and all of the argument types and function results. The DLL file is **not** loaded until the routine is called with the *calldllfun* routine. So errors involving paths to the DLL file or other related problems are not reported until the first call to *calldllfun*.

The operation *calldllfun* provides an interface to an external routine implemented as a DLL for 32-bit Windows. The name *Nm* is the name associated with the routine when it was registered using *registerdllfun*. The use of the name with appropriate arguments causes the DLL routine to be executed and its result (if any) returned as a Nial array.

The arguments *Arg0 ... ArgN* must agree in number and be compatible in type with the declarations made while registering the routine.

The details of argument passing and some examples are included in the entry for *registerdllfun*.

callstack

Class: system expression

Usage: `Callstack`

See Also: break, debugging

The expression *Callstack* displays the sequence of active definition calls at the point it is invoked. It is usually used in conjunction with *Break*. *Callstack* can be used to see the execution path by which the computation reached the current state while computation is suspended during a break.

canonical

Class: concept

See Also: descan, execute

There is a **canonical** way of displaying program text in Nial. This is done automatically by the routines *descan* and *deparse* used by *see*. The canonical form sets the case of all identifiers used in Nial program text according to their role in order to assist visual parsing of Nial text. The following table summarizes the rules:

Role	Case rule
Variable	first letter upper case, the rest lower case
Expression	first letter upper case, the rest lower case
Operation	all lower case
Transformer	all upper case
Reserved Word	all upper case

Definition

```
canonical IS link descan deparse parse scan
```

cart

Class: construction operation

Usage: `cart A   A cart B`

See Also: outer, tell

The operation *cart* corresponds to the cartesian product of set theory. Its purpose is to form all possible combinations of the items of its argument and return them in a structured result having as many axes as the sum of the number of axes of the items of argument.

```
     2 3 4 cart 5 6
+---+---+
|2 5|2 6|
+---+---+
|3 5|3 6|
+---+---+
|4 5|4 6|
+---+---+
```

In the example above, the cart of a triple with a pair yields a table of shape 3 2 of pairs.

```
     cart ['abc', 4 2 reshape count 8]
+---+---+  +---+---+  +---+---+
|a 1|a 2|  |b 1|b 2|  |c 1|c 2|
+---+---+  +---+---+  +---+---+
|a 3|a 4|  |b 3|b 4|  |c 3|c 4|
+---+---+  +---+---+  +---+---+
|a 5|a 6|  |b 5|b 6|  |c 5|c 6|
+---+---+  +---+---+  +---+---+
|a 7|a 8|  |b 7|b 8|  |c 7|c 8|
+---+---+  +---+---+  +---+---+
```

In the above example, the shape of the result is 3 4 2, getting an axis of length 3 from the first item and axes of length 4 and 2 from the second.

Equations

```
shape cart A = link EACH shape A
valence cart A = sum EACH valence A
tally cart A = product EACH tally A
equal EACH shape cart A = True
A OUTER pair B = A cart B
cart link A = EACH link cart EACH cart A
cart EACH EACH f A = EACH EACH f cart A
cart single A = EACH single A
simple A ==> cart A = single A
cart Null = single Null
empty A ==> cart A = single A
or EACH empty A ==> empty cart A
not empty A and not empty cart A ==> shape first cart A = shape A
not empty A and not empty cart A ==> first cart A = EACH first A
isshape A ==> tell A = cart EACH tell A
```

case-expr

Class: control structure

Usage: `CASE expression FROM C1: ES1 END ... Cn: ESn END ELSE ESx ENDCASE`

See Also: if-expr, fork

The expression following case is evaluated. If the result matches one of the constants, C1 ... Cn, the corresponding expression sequence is executed. If the result does not match any constant, the expression sequence following else is executed.

Example:

```
     Month := 3 ;
     CASE Month FROM
        1:   Mname := 'January'; END
        2:   Mname := 'February'; END
        3:   Mname := 'March'; END
     ELSE   Mname := 'Year End';
     ENDCASE ;
     Mname
March
```

cast

Class: syntax

A *cast* is an array expression that denotes an internal representation of a valid fragment of Q'Nial program text:

```
     cast ::= ! identifier
        | ! ( expression-sequence)
        | ! ( operation-expression )
        | ! ( transformer-expression )
```

The use of the exclamation symbol *!* before an identifier causes Q'Nial to select the internal representation for the identifier rather than the value of the array associated with the identifier. Its use before a parenthesized program fragment selects the internal representation of the program fragment.

The major use of casts is in conjunction with the operations assign and apply. These operations mimic the Q'Nial constructs for assignment to a variable and application of an operation to an array. Casts permit passing an argument to an operation by variable name rather than by value. They also permit evaluation of a program fragment that has been stored in its internal form using the operation eval rather than requiring the use of the operation execute on the corresponding program text stored as a string.

```
     Salary := 90000.
     A gets 'Salary > 100000.' ;
     Rule1 := execute A
o
     Rule1 := eval !A
o
```

The details of the internal representation is not specified as part of the Nial language.

The cast notation *!Name* is used to denote the parse tree that represents the name. At the top level loop, parentheses must be included around the use of the cast notation, e.g. *(!Name)*, to avoid ambiguity with the use of *!* to indicate a host command.

The cast, because it is analyzed in the context in which it appears, refers to a variable or definition in a static way.

Q'Nial contains operations that mimic the underlying meaning of variables, expressions and operations in Q'Nial. The operations use strings, phrases or casts to represent the name of the object under consideration (except that see and getdef do not take casts).

Operation	**Action**
value A	Return the value of a variable named by string, phrase or cast *A*.
A assign B	Assign the array *B* to the variable named by string, phrase or cast *A*; return *B*.
A apply B	Apply the operation named by string, phrase or cast *A* to array *B*; return the result of the operation.
getdef A	Return the parse tree associated with the definition named by string or phrase *A*.
see A	Display the definition named by the string or phrase *A*.
update P A B	Put array *B* at address *A* in the variable named by the phrase, string or cast *P*; return the new value of *P*.
updateall P A B	Put items of *B* at addresses *A* in the variable named by the phrase, string or cast *P*; return the new value of the variable named by *P*.
deepupdate P A B	Put array *B* at path *A* in variable named by the string, phrase or cast *P*; return new value of the variable named by *P*.

catenate

Class: construction operation

Usage: `I catenate A  catenate I A`

See Also: link, laminate

The operation *catenate* joins the items of *A* along axis *I*. The items of *A* must conform in all other axes.

```
     1 catenate (tell 2 3) (count 2 5)
+---+---+---+---+---+---+---+---+
|0 0|0 1|0 2|1 1|1 2|1 3|1 4|1 5|
+---+---+---+---+---+---+---+---+
|1 0|1 1|1 2|2 1|2 2|2 3|2 4|2 5|
+---+---+---+---+---+---+---+---+
```

The example joins two tables along the rows. Each table has two rows.

Definition

```
catenate IS OPERATION I A {
   % "push down" I axis of items of A;
   B := EACH ( I split ) A ;
   IF equal EACH shape B THEN
      I blend EACH link pack B
   ELSE
      fault '?conform error in catenate'
   ENDIF }
```

ceiling

Class: arithmetic operation

Property: unary pervasive

Usage: `ceiling A`

See Also: floor, mod, quotient

The operation ceiling produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	corresponding integer
integer	argument
real	next higher integer or the fault ?A, if the result is outside the range of integers
character	fault ?A
phrase	fault ?A
fault	argument A

Examples

```
     ceiling  1  -2  3.5
1 -2 4
     ceiling  `a  "abc  ??error
 ?A ?A ?error
     ceiling  3.5  -4.6  7.0  25.3e20
4 -4 7 ?A
```

Definition

```
ceiling IS OPERATION A {
   opposite floor opposite A }
```

char

Class: conversion operation

Property: unary pervasive

Usage: `char N`

See Also: charrep, ischar, isinteger, isstring

The operation *char* is used to convert an integer in the range 0 to 255 to the character that has the integer as its internal representation. The result is system dependent.

```
     char 66
B
     char 66 67 68 69
BCDE
```

The major purpose of *char* is to create special characters for a specific system. For example, the characters that control cursor motion differ from one terminal to another. Programs using *char* may not be portable.

Equations

```
N in tell 256 ==> charrep char N = N
ischar C ==> char charrep C = C
```

charrep

Class: conversion operation

Property: unary pervasive

Usage: `charrep C`

See Also: char, isinteger, ischar, isstring

The operation *charrep* is used to convert a character to its internal representation as an integer. This operation is system dependent.

```
     charrep `A
65
     charrep 'hello'
104 101 108 108 111
```

The major purpose of *charrep* is to permit determination of special characters for a specific system. For example, the characters that control cursor motion differ from one terminal to another.

Programs using charrep may not be portable.

Equations

```
N in tell 256 ==> charrep char N = N
ischar C ==> char charrep C = C
```

checkpoint

Class: user defined expression

Usage: `Checkpoint`

See Also: save, load, latent

Checkpoint is a user defined expression executed after a *save*.

When a *save* is executed, the current computation is ended and the *Checkpoint* is done in the top level environment.

If a workspace being saved contains an expression named *Checkpoint*, the expression is executed following the saving of the workspace and prior to restarting the top level loop. It can be used to restart a computational loop after an intermediate dump of the workspace.

choose

Class: selection operation

Property: binary

Usage: `I choose A  choose I A`

See Also: pick, take, takeright, reach

The operation *choose* is used to select a subarray from array *A* specified by the array of addresses *I*. The result is an array of the same shape as *I* with items chosen from *A*. If an item of *I* is not an address of *A*, the corresponding position in the result is the fault *?address*.

The operation *choose* is related to the *at all* notation *V#I*, which selects items from the array associated with the variable *V*. The differences are that *choose* may select from an array that has not been assigned to a variable and that it handles out-of-range in a different manner.

```
     3 1 0 1 3 4 choose 'range'
garage
```

The following example shows that the *valence* of the array of addresses can be higher than that of the array from which the selection is made.

```
     I gets (2 4 reshape tell 8)
0 1 2 3
4 5 6 7
     I choose 'some words as a string'
some
 wor
```

Definition

```
     choose IS OPERATION I A { I EACHLEFT pick A }
```

Equations

```
shape (I choose A) = shape I
J choose (I choose A) = (J choose I) choose A
tell shape A choose A = A
(list I) choose A = list (I choose A)
(EACH list I) choose A = I choose A
I allin grid A ==> EACH f (I choose A) = I choose EACH f A
```

clear workspace

Class: concept

See Also: continue workspace, standard definitions, save, load, symbols, clearws, restart

Q'Nial organizes the data and code objects available for use into a logical structure called a **workspace**. It consists of a symbol table to hold associations between names and the predefined and user defined objects, a heap to store data and parse trees, a stack used to hold values temporarily during execution, and an atom table for uniquely storing phrases and faults.

When Q'Nial is invoked it starts the session with a clear workspace unless a specific saved workspace is requested or there is a *continue.nws* file in the local directory. The clear workspace is either found in the local directory, in the directory pointed at by *Nialroot*. The name of the clear workspace is *clearws.nhs*. If it is not found, then Q'Nial creates an internal clear workspace using its initialization process. A message indicates whether a clear workspace was loaded or created.

A user can tailor the initial state of Nial for their own purposes by saving a workspace with definitions in place as *clearws.nws* using *save*.

clearprofile

Class: profiling expression

Usage: `Clearprofile`

See Also: profile, setprofile, profiletable, profiletree, profiling

The expression *Clearprofile* is used to clear the internal data structures that are used in the gathering of profiling statistics. It should be called when one profiling session has been completed and *profile* has been called, before starting another one.

A detailed explanation of the profiling mechanism is given in the help entry on profiling.

Example

```
Clearprofile
```

clearws

Class: system expression

Usage: `Clearws`

See Also: restart, load, save

The expression *Clearws* clears the current workspace, erasing all variables and user definitions. It can cause loss of valuable information. The operation *save* should be used prior to using *Clearws* if the information in the workspace will be needed again.

The expression *Restart* is similar to *Clearws* except *Restart* re-initializes the interpreter according to the initial options given on the command line in a console version or set as defaults in a GUI version.

close

Class: file operation

Usage: `close Fd`

See Also: open, filestatus

The operation *close* is used to close a file previously opened with the *open* operation. The argument is the file designator, an integer returned previously by *open* for that file.

The result of *close* is the fault *?noexpr* or a fault message indicating an error.

```
     Fd := open "foo "w
3
     close Fd
     close Fd
?file is not open
```

In the examples above, the first use of *close* was successful. The second attempt to close the file resulted in the fault message.

cols

Class: nesting restructuring operation

Usage: `cols A`

See Also: rows, split, raise, rank

The operation *cols* rearranges the axes of a table of shape *M* by *N* to form a list of length *N* of the columns of length *M* of the table. *Cols* is generalized to arrays of all valences as follows: if *A* is a list or a single, the result is *single A*; if *A* has valence three or higher, the result is an array of valence one less than the valence of *A* with the second last axis pushed down.

```
     A := 3 4 reshape count 12
1  2  3  4
5  6  7  8
9 10 11 12

     cols A
+-----+------+------+------+
|1 5 9|2 6 10|3 7 11|4 8 12|
+-----+------+------+------+
```

If only column *I* of a table is needed, it can be selected by *I pick cols A* or by *A|[,I]* using the *at slice* indexing notation. The second way is more efficient for large arrays because it avoids restructuring the array. If a table *A* has no columns, *cols A* results in the empty list *Null*.

Definition

```
cols IS OPERATION A {
   IF valence A = 0 THEN single A
   ELSE
      valence A - 2 max 0 split A
   ENDIF }
```

Equations

```
valence A = 2 and not empty A ==> cols A = pack rows A
valence A = 2 and not empty A ==> cols A = rows transpose A
valence A = 2 and not empty A ==> mix cols A = transpose A
```

comment

Class: syntax

See Also: action

```
comment ::=
   % <any text excluding a semicolon> ;

remark ::= # < any text >
```

A **comment** is a brief section of text included in a program fragment to assist readability. Comments may be placed anywhere in a block before or after declarations, definitions or expressions. Their purpose is to provide an explanation of the program fragment for the programmer who may be required to modify the program at a later date. The value of a comment as an expression is the *?noexpr* fault. Comments are retained when a definition is translated into internal form and they appear in its creation in the canonical form used by the operations see and defedit.

A **remark** is an input to the Q'Nial interpreter that is not processed. It begins with a line that has the symbol # as the first non-blank character in the line. In direct input at the top level loop, a remark ends at the end of the line unless a backslash symbol (\ ) is used to extend the line. In a definition file, a remark ends at the first blank line. A remark cannot appear within a definition or expression-sequence.

conform

Class: concept

See Also: pack, binary pervasive, multi pervasive

The operation pack is used in evaluating binary pervasive and multi pervasive operations. Its task is to interchange the top two levels of the argument to such operations if the items of the argument **conform**. For the binary case there are two items; they conform if the items have the same shape, or if one or both items have only one item. In the latter case, the item with only one item is replicated to the shape of the other item.

For the multi pervasive case, all the items that do not have only one item must be of the same shape and all the items with one item are replicated to that shape.

```
     pack [2 3 4,10 11 12]
+----+----+----+
|2 10|3 11|4 12|
+----+----+----+

     pack [2 3,10 11 12]
?conform

     pack [3 4 5,3,10 11 12,[4]]
+--------+--------+--------+
|3 3 10 4|4 3 11 4|5 3 12 4|
+--------+--------+--------+
```

```
    pack [tell 4 5,count 4 5]
+---------+---------+---------+---------+---------+
|+---+---+|+---+---+|+---+---+|+---+---+|+---+---+|
||0 0|1 1|||0 1|1 2|||0 2|1 3|||0 3|1 4|||0 4|1 5||
|+---+---+|+---+---+|+---+---+|+---+---+|+---+---+|
+---------+---------+---------+---------+---------+
|+---+---+|+---+---+|+---+---+|+---+---+|+---+---+|
||1 0|2 1|||1 1|2 2|||1 2|2 3|||1 3|2 4|||1 4|2 5||
|+---+---+|+---+---+|+---+---+|+---+---+|+---+---+|
+---------+---------+---------+---------+---------+
|+---+---+|+---+---+|+---+---+|+---+---+|+---+---+|
||2 0|3 1|||2 1|3 2|||2 2|3 3|||2 3|3 4|||2 4|3 5||
|+---+---+|+---+---+|+---+---+|+---+---+|+---+---+|
+---------+---------+---------+---------+---------+
|+---+---+|+---+---+|+---+---+|+---+---+|+---+---+|
||3 0|4 1|||3 1|4 2|||3 2|4 3|||3 3|4 4|||3 4|4 5||
|+---+---+|+---+---+|+---+---+|+---+---+|+---+---+|
+---------+---------+---------+---------+---------+
```

Definition

```
conform IS OP A {
    equal EACH shape
      (not (EACH tally A match 1) sublist A) }
```

content

Class: nesting restructuring operation

Usage: `content A`

See Also: link, list, in, solitary

The operation *content* returns the list of atoms of *A*. The effect of content is to remove all structure from an array, returning a list of the atoms in a depth first, row-major order. The effect on an atom is to produce the solitary of the atom.

```
     A := (2 3) (4 5 (6 7))
+---+---------+
|2 3|+-+-+---+|
|   ||4|5|6 7||
|   |+-+-+---+|
+---+---------+

     content A
2 3 4 5 6 7
```

Definition

```
content IS OPERATION A {
   IF atomic A THEN
      list A
   ELSE
      link EACH content A
   ENDIF }
```

Equations

```
content A = list content A
content A = content list A
content A = link EACH content A
```

continue

Class: system expression

Usage: `Continue`

See Also: bye, load, save

The expression *Continue* terminates a session of Q'Nial and returns control to the host environment. The workspace is saved in file *continue.nws* and is reloaded when Q'Nial is invoked at a later session using the same current directory.

If a *Restart* is executed in a session that began with a continue workspace, the current workspace will be reset to the continue workspace. If a session that began with a continue is terminated with *Continue*, the file continue.nws is updated with the current workspace. If it is terminated with *Bye*, the file *continue.nws* is deleted.

continue workspace

Class: concept

See Also: continue, clear workspace

The file *continue.nws* is a workspace file that is created when a session is ended with the expression *Continue*. When Q'Nial is invoked from the same directory, it will use *continue.nws* rather than *clearws.nws* as the initial workspace. If the Q'Nial invocation names a workspace, that workspace will take precedence over the continue workspace.

control tuples

Class: display parameter

Usage: `[foreground, background, blink status]`

See Also: putstrings, setwindow, window

Many of the screen related operations require a display attribute, termed a control tuple, as one of the arguments. The display attribute specifies the color or style of the characters displayed by the operation. It is display screen dependent.

Various displays are available for IBMPC systems and they can run in different modes. For a monochrome display, Q'Nial can specify normal, bold, underlined, reverse video and blinking characters. If the terminal has a color display, Q'Nial can specify foreground and background colors. Each color combination can have blinking characters.

For a Unix system, the display screen is driven by the terminfo capability table which provides the following styles: normal, standout, underlined, bold and blinking, where either standout or bold is mapped to reverse video. Not all terminals support all these capabilities. No specific allowance for color is made in the Unix version of Q'Nial.

To accommodate all these varieties, the display attribute is encoded as a triple of integers on IBMPC versions and as a single integer on Unix with the following interpretations:

Display	first	second	third
IBMPC Mono	style	reversed or not	blinking
IBMPC Color	foregrd	backgrd color	blinking
Unix terminal	style	unused	unused

The following tables fill in the detailed codes for the three forms of display attributes.

IBMPC Monochrome Display Attributes

Code	Ch Style	Rev. Video	Blink
0	Normal	No	No
1	Underlined	Yes	Yes
2	Bold		
3	Underlined and bold		

IBMPC CGA Color Display Attributes

Code	Foregrd	Backgrd	Blink
0	black	black	No
1	blue	blue	Yes
2	red	red	
3	yellow	yellow	
4	white	white	
5	light blue	light blue	
6	purple	purple	
7	green	green	

Color codes 8 through 15 result in high intensity display of the color.

Unix Terminal Display Attributes

Code	Ch Style
0	Normal
1	Standout
2	Bold
3	Underlined
4	Blinking

With this variety of attributes, it is impossible to make programs work identically when they are ported from one system to another. To retain portability attributes should be set up in terms of variables such as

Normalattribute, Highattribute, Borderattribute etc. and given values according to the system in use.

converse

Class: applicative transformer

Usage: `A CONVERSE f B  CONVERSE f A B`

See Also: fold

The transformer *CONVERSE* is used with a binary operation *f* and applies *f* to the pair formed by reversing the arguments of *CONVERSE f*.

```
     at IS CONVERSE pick
     'abcde' at 2
c

     holds IS CONVERSE in
     count 20 holds 5
1
```

In the above examples, *at* and *holds* are defined in terms of *pick* and *in*, respectively. The first example shows that *'abcde' at 2* gives the result expected for *2 pick 'abcde'*. The second example shows that *count 20 holds 5* gives the same result as *5 in (count 20)*. Thus, *at* and *holds* do the same work as *pick* and *in*; the former operations simply use their arguments in the reverse order.

Definition

```
     CONVERSE IS TRANSFORMER f OPERATION A B {
        B f A }
```

Equations

```
   A CONVERSE CONVERSE f B = A f B
   A EACHLEFT f B = B EACHRIGHT CONVERSE f A
```

copyright

Class: constant expression

Usage: `Copyright`

See Also: system, version

The expression *Copyright* returns a message specifying ownership of the rights to Q'Nial by Queen's University at Kingston, Canada (K7L 3N6).

```
     Copyright
Copyright (c) Queen's University 1983-97
```

cos

Class: scientific operation

Property: unary pervasive

Usage: `cos A`

See Also: sin, cosh, pi

The operation *cos* implements the cosine function of mathematics. It produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	cosine of the corresponding real
integer	cosine of the corresponding real
real	cosine of angle A given in radians
character	fault ?cos
phrase	fault ?cos
fault	argument A

```
    cos  1  -1  0.5  `a  "abc  ??error
0.54030 0.54030 0.87758 ?cos ?cos ?error
```

Equation

```
cos opposite A = cos A
```

cosh

Class: scientific operation

Property: unary pervasive

Usage: `cosh A`

See Also: sinh, cos, pi

The operation *cosh* implements the hyperbolic cosine function of mathematics. It produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	hyperbolic cosine of the corresponding real
integer	hyperbolic cosine of the corresponding real
real	hyperbolic cosine of angle A given in radians
character	fault ?cosh
phrase	fault ?cosh
fault	argument A

```
     cosh  1  -1  0.5
1.54308 1.54308 1.12763

     cosh `a  "abc  ??error
?cosh ?cosh ?error
```

Equation

```
   cosh opposite A = cosh A
```

count

Class: array generation operation

Usage: `count N`

See Also: tell

The operation *count* generates a list of integers starting at 1 and going up to and including *N*. It differs from *tell* by counting from 1 instead of from 0.

```
     count 5
1 2 3 4 5

     .1 times count 5
0.1 0.2 0.3 0.4 0.5
```

The first example shows *count* generating the sequence of integers from 1 to 5. The second example shows how *count* can be used to generate a sequence of five real numbers with an interval of .1 between each number.

```
     count 2 3
+---+---+---+
|1 1|1 2|1 3|
+---+---+---+
|2 1|2 2|2 3|
+---+---+---+
```

The third example shows how *count* generalizes to other arguments in the same manner as the operation *tell*.

Definition

```
     count IS OPERATION A { 1 + tell A }
```

Equations

```
   isshape A ==> shape count A = A
   isshape A ==> count A = cart EACH count A
```

cull

Class: selection operation

Usage: `cull A`

See Also: except, diverse, bykey

The operation *cull* returns a list whose items are those of *A* with duplicates removed. The order of the items is maintained.

```
     cull 3 5 4 3 5 2 4
3 5 4 2

     cull 'a few letters with duplicates'
a fewltrsihdupc
```

Definition

```
     cull IS OPERATION A {
        grid A EACHLEFT in (A EACHLEFT find A) sublist A }
```

Equations

```
   cull A = list cull A
   diverse A <==> cull A = list A
   sortup cull A = cull sortup A
```

Pragmatics

The operation *cull* executes faster when a large array has been sorted, hence if the ordering of the result is unimportant it is better to use *sortup* to order the array prior to applying *cull*.

curried operation

Class: concept

See Also: apply, operation composition

A *curried-operation* is an operation in which the left item of a two-item argument is combined with a given operation to form an operation expression. Examples are: *1+* and *3 reshape*.

A curried operation can be named or grouped in parentheses as an argument to a transformer.

```
     incr IS 1+

     EACH (5 take) Lines
```

In general, the syntax of a curried operation is:

```
       curried-operation ::= simple-expression simple-operation
```

The result of applying a curried-operation is determined by applying the simple-operation to the pair formed from the simple-expression and the argument to the curried-operation. Thus,

```
     (1+) 5
6
```

is interpreted as

```
     + (1 5)
6
```

cut

Class: selection operation

Property: binary

Usage: `B cut A   cut B A`

See Also: cutall

The operation *cut* converts an array *A* into a list of items formed from the items of *A* according to the bitstring *B*. The list of items is divided where true values occur in the corresponding positions in the bitstring *B*. The items of *A* where the divisions occur are not included in the items of the result and any empty segments are not included.

```
     A := 'The boy stood   on the burning deck' ;
     `  match A cut A
+---+---+-----+--+---+-------+----+
|The|boy|stood|on|the|burning|deck|
+---+---+-----+--+---+-------+----+

     B := 'Formula 1: 3,5,7,,,9';

     `, match B cut B
+------------+-+-+-+
|Formula 1: 3|5|7|9|
+------------+-+-+-+

     EACH equal tell 4 4 cut tell 4 4
+-----------------+-----------------+-----------------+
|+---+---+---+---+|+---+---+---+---+|+---+---+---+---+|
||0 1|0 2|0 3|1 0|||1 2|1 3|2 0|2 1|||2 3|3 0|3 1|3 2||
|+---+---+---+---+|+---+---+---+---+|+---+---+---+---+|
+-----------------+-----------------+-----------------+
```

Cut can be used to cut a string into a list of substrings. In the first example, the string is cut at blank characters. Several adjacent blanks are treated as one blank character. In the second example, the cut is done where commas occur. The third example shows that the items of *A* do not have to be atoms.

Definition

```
cut IS OPERATION B A {
   C := EACH rest (B cutall A);
   not EACH empty C sublist C }
```

Equations

```
B cut A = (shape A reshape B) cut A
list (B cut A) = B cut A
B cut list A = B cut A
```

cutall

Class: selection operation

Property: binary

Usage: `B cutall A   cutall B A`

See Also: cut

The operation *cutall* converts an array *A* into a list of items formed from the items of *A* according to the

bitstring *B*. The list of items is divided where true values occur in the corresponding positions in the bitstring *B*. The items where the divisions occur are kept as the first item of each group.

```
     A := 'The boy stood on the deck' ;
     `  match A cutall A
+---+---+------+---+----+-----+
|The| boy| stood| on| the| deck|
+---+---+------+---+----+-----+

     B := 'Formula 1: 3,5,7,,,9';
     `, match B cutall B
+------------+--+--+-+-+--+
|Formula 1: 3|,5|,7|,|,|,9|
+------------+--+--+-+-+--+

     EACH equal tell 4 4 cutall tell 4 4
+---------------------+--------------------+--------------------+-----+
|+---+---+---+---+---+|+---+---+---+---+---+|+---+---+---+---+---+|+---+|
||0 0|0 1|0 2|0 3|1 0|||1 1|1 2|1 3|2 0|2 1|||2 2|2 3|3 0|3 1|3 2|||3 3||
|+---+---+---+---+---+|+---+---+---+---+---+|+---+---+---+---+---+|+---+|
+---------------------+--------------------+--------------------+-----+
```

Equations

```
tally (B cutall A) = sum (shape A reshape B)
B cutall A = (shape A reshape B) cutall A
list (B cutall A) = B cutall A
B cutall list A = B cutall A
```

debugging

Class: concept

See Also: breakin, watch

Debugging Definitions

The Q'Nial system provides an optional debugging facility that aids interactive debugging of definitions. It is active by default, but can be turned off for running production applications. See the detailed documentation for the various versions on how to turn off debugging.

The debugging system is based on the idea of placing breaks in the code and stepping through the program code in a number of different ways. Due to constraints in the way Q'Nial is implemented, debugging is always done in the context of an expression sequence. A break point occurs either before the execution of the expression sequence in a definition, or at an explicit break expression within an expression sequence. There is also a watch mechanism that executes a defined action whenever the value of a variable changes, and an ability monitor all use of user defined objects and of the predefined operations.

Defining a Break Point

There are two ways to cause a break in a Nial definition: by using the expression *Break* in an expression sequence, or by using the operation *breakin* to set a break on entry to the operation. The following table summarizes the break related primitives:

Expression	Action
Break	Suspend evaluation of the expression and pass control to an evaluation loop in the environment at the point of the break. Variables accessible at that point can be displayed. This loop recognizes a number of

	commands described below.
breakin Nm [M]	Set or rest an internal break flag for the definition of *Nm*. If the boolean value *M* is omitted, the flag is toggled. If set, a break occurs before the execution of the expression sequence of the definition. The *Nm* must be the name of a defined expression or a defined operation using the operation form style of operation expression.
Breaklist	Display the list of names of definitions with break flag set.

deepplace

Class: insertion operation

Usage: `B P deepplace A  deepplace (B P) A`

See Also: place, placeall, deepupdate

The operation *deepplace* returns an array the same as *A* except that the array at path *P* is replaced by *B*. It is the insertion operation corresponding to the selection operation *reach* and generalizes the operation *place* from addresses to paths.

```
     A1 := 1 2 (3 4 (5 6 (7 (8 9))))
+-+-+-------------------+
|1|2|+-+-+-------------+|
| | ||3|4|+-+-+-------+||
| | || | ||5|6|+-+---+|||
| | || | || | ||7|8 9||||
| | || | || | |+-+---+|||
| | || | |+-+-+-------+||
| | |+-+-+-------------+|
+-+-+-------------------+
     A1 := "M (2 2 2 0) deepplace A1
+-+-+-------------------+
|1|2|+-+-+-------------+|
| | ||3|4|+-+-+-------+||
| | || | ||5|6|+-+---+|||
| | || | || | ||M|8 9||||
| | || | || | |+-+---+|||
| | || | |+-+-+-------+||
| | |+-+-+-------------+|
+-+-+-------------------+
```

The array 7 is replaced with

Definition

```
deepplace IS OPERATION C A {
   B Path := C ;
   IF empty Path THEN
      B
   ELSE
      I := first Path ;
      ((B (rest Path)) deepplace (I pick A)) I place A
   ENDIF }
```

Equations

```
Path a valid path into A ==> ( Path reach A ) Path deepplace A = A
B Null deepplace A = B, I in grid A ==> B
deepplace A = B I place A
```

deepupdate

Class: evaluation operation

Usage: `deepupdate Nm P A`

See Also: update, updateall, deepplace

The operation *deepupdate* provides the semantics of the *Nm@@P := A* form of assignment expression. *Nm* must be an existing variable represented by a string, phrase or a cast; *P* is the path of addresses to the location to be updated; and *A* is the array to be placed in the variable.

```
     Array1 := (1 2(3 4 (5 6 (7 (8 9)))))
+-+-+--------------------+
|1|2|+-+-+--------------+|
| | ||3|4|+-+-+-------+||
| | || | ||5|6|+-+---+|||
| | || | || | ||7|8 9||||
| | || | || | |+-+---+|||
| | || | |+-+-+-------+||
| | |+-+-+--------------+|
+-+-+--------------------+


     deepupdate "Array1 (2 2 2 1 0) "Tom
+-+-+----------------------+
|1|2|+-+-+----------------+|
| | ||3|4|+-+-+---------+||
| | || | ||5|6|+-+-----+|||
| | || | || | ||7|Tom 9||||
| | || | || | |+-+-----+|||
| | || | |+-+-+---------+||
| | |+-+-+----------------+|
+-+-+----------------------+
```

The major purpose of *deepupdate* is to allow a selective update with a path on a global variable without forcing a copy. By passing the name of the variable to the operation that is doing the update, rather than its value, no sharing of the internal data is made and hence the update can be made "in place".

definition

Class: concept

See Also: expression, operation, transformer

A **definition** in Nial is a syntactic construct that names a program fragment. The syntax is one of the three forms:

```
<name>  IS  <array expression>
<name>  IS  <operation expression>
<name>  IS  <transformer expression>
```

A definition is used to associate a name (identifier) with a program fragment that is an array expression, an operation expression or a transformer expression. If the definition appears within a block, the association is made in the local environment. Otherwise, the association is made in the global environment and assigns a role to the name as representing that kind of expression.

If the program fragment is syntactically correct, the name is associated with the program fragment in the environment and no result is given. If a syntax error is detected in the analysis of the program fragment, an

explanatory fault message is returned and the name association is not made.

If the name being associated in a definition is already in use, the new definition must be for a construct of the same role and the earlier definition is replaced. The use of a defined name always refers to its most recent definition.

deparse

Class: evaluation operation

Usage: `deparse Pt`

See Also: descan, parse, scan, defedit, see

The operation *deparse* is used to convert a parse tree representation of a Nial definition or program fragment into a token stream which can be converted into text corresponding to the Nial definition. The argument to *deparse* must be either a cast or the result of *parse* or *getdef*.

```
     deparse !(Pi * cos 0.5)
99 1 ( 2 Pi 2 * 2 cos 18 0.5 1 )
```

The principal use of *deparse* is in displaying Nial definitions. It is used in the definition of *defedit* and *see*.

The token stream returned by *deparse* also includes indicators to identify where new lines are to begin and to control indentation. The tokens corresponding to identifiers are given in the canonical form, indicating what role each token plays.

The result of *deparse* is implementation dependent and should be viewed as an internal representation provided as an interface to editing. It is subject to change as Q'Nial evolves.

Equations

```
Pt a parse tree ==> parse deparse Pt = Pt
Ts a token stream ==> parse deparse parse Ts = parse Ts
```

Depth

Class: measurement operation

Usage: `tally A`

See Also: tally, down

The operation *depth* returns an integer indicating the number of levels of nesting of the array. This is called the depth of the array. The depth of an atom is 0. The depth of a simple nonempty is 1. In general, the depth of an array is 1 plus the maximum of the depths of the items.

```
     depth "abc
0
     depth 3 4 5
1
     depth 1 (2 3) (4 5 6)
2
     depth [2, [3, 4, [5],8],24]
3
```

Definition

```
     depth IS OPERATION A
     { IF atomic A THEN
          0
       ELSE
          1 + (max EACH depth A)
       ENDIF
     }
```

Equations

```
   depth Null = max Null
   tally depth A = 1
   depth A = FORK [atomic, 0 first, 1 plus max EACH depth] A
```

descan

Class: evaluation operation

Usage: `descan Ts`

See Also: scan, deparse, defedit, see, canonical

The operation *descan* converts a token stream to a list of strings that represents the program fragment given by the token stream. The input to *descan* must be a token stream produced by *scan* or *deparse* or an equivalent list of tokens.

```
     descan scan 'A + 32'
+-------+
|A + 32 |
+-------+

     descan deparse !(Pi * cos 0.5)
+-----------------+
|( Pi * cos 0.5 ) |
+-----------------+
```

The principal use of *descan* is in displaying Nial definitions. It is used to define *defedit* and *see*. The argument to *descan* may also include tokens indicating the beginning of new lines and controlling indentation used to display a structured definition.

Equations

```
   S a string of Nial text ==> scan link descan scan S = scan S
   S a string of Nial text ==> canonical canonical S = canonical S
   S a string of Nial text ==> execute canonical S = execute S
```

diagram

Class: picture operation

Usage: `diagram A`

See Also: display, paste, picture, positions, sketch, set

The operation *diagram* computes a character table that gives the fully boxed picture of *A* with the decoration of the atoms determined by the current setting of the decor switch. An array is displayed as a frame with cells for each item. It is arranged in two dimensions, using groupings of table frames to picture arrays of higher dimension. Each cell is large enough to hold the diagram of the corresponding item of *A*.

The diagram of an atom is a picture that indicates its value. The diagram of a non-atomic single has an "o" in its upper left corner.

Diagram returns the display given by the operation *picture* when in diagram mode of display.

The decor or nodecor mode switch controls the display of atoms. With decor set, it gives a picture that distinguishes all atoms.

```
     set "decor;
     A := diagram (2 3 2 reshape 3 'abc' (2 1 reshape count 3) "apple 8.5
          (3 4));
     set "nodecor; A
+---+----------+  +---+----------+
|  3|+--+--+--+|  |  3|+--+--+--+|
|   ||`a|`b|`c||  |   ||`a|`b|`c||
|   |+--+--+--+|  |   |+--+--+--+|
+---+----------+  +---+----------+
|+-+|"apple    |  |+-+|"apple    |
||1||          |  ||1||          |
|+-+|          |  |+-+|          |
||2||          |  ||2||          |
|+-+|          |  |+-+|          |
+---+----------+  +---+----------+
|8.5|+-+-+     |  |8.5|+-+-+     |
|   ||3|4|     |  |   ||3|4|     |
|   |+-+-+     |  |   |+-+-+     |
+---+----------+  +---+----------+
```

The display of the result of a *picture* operation, such as *diagram*, makes sense when it itself is pictured in sketch-nodecor mode. In other modes, the characters making up the table would be boxed and/or decorated.

Definition

```
     diagram IS OPERATION A {
        Old_setting := set "diagram ;
        Result := picture A ;
        set Old_setting ;
        Result }
```

dimensions

Class: concept

See Also: valence, axes

The number of axes of an array is referred to as its dimensionality. In array theory terminology the

dimensionality is called the **valence** of the array. The following terms describe arrays by their valence:

Valence	Description
0	single
1	list, vector
2	table, matrix
2 or more	multivalent

display

Class: picture operation

Usage: `display A`

See Also: picture, execute

The operation *display* returns a string which, when executed, returns the value *A*.

```
     set "decor; display 23.5
'23.5'
```

Display inserts the operation to construct values that cannot be described directly by constants. In the example below, since the phrase containing a blank character cannot be specified simply using the phrase mark, display inserts the operation *phrase* and the string that will create the desired phrase *An answer*.

```
     display "Nm (phrase 'An answer') 'Nm'
'["Nm,(phrase 'An answer'),'Nm']'

     X := "Name (2 3 (4 5) 'Queen''s' (3 2 reshape count 6))
+-----+-----------------------+
|"Name|+-+-+---+---------+---+|
|     ||2|3|4 5|'Queen's'|1 2||
|     || | |   |         |3 4||
|     || | |   |         |5 6||
|     |+-+-+---+---------+---+|
+-----+-----------------------+

     display X
'["Name,[2,3,4 5,'Queen''s',3 2 reshape 1 2 3 4 5 6]]'

     execute display X = X
1
```

In the definition of *X*, strand notation was used. In the display of *X*, brackets notation is created to represent *X*. The display is seen to be correct by the last example that shows that the *execute* of the *display* of *X* is *X*.

Equation

```
execute display A = A
```

diverse

Class: logic operation

Property: predicate

Usage: `diverse A`

See Also: equal, in, cull

The operation *diverse* tests whether or not the items of *A* are all different. It returns *true* if they are and *false* if they are not.

```
     diverse 2 3 5
1

     diverse 'hello world'
0

     diverse 2 3 (2 3)
1
```

In the last example, the items are all different because the last item is a pair of items (2 3) which is not the same as 2 or 3.

Definition

```
     diverse IS OPERATION A { cull A = list A }
```

Pragmatics

The operation *diverse* executes faster when a large array has been sorted, hence it is better to use *sortup* to order a large array prior to applying *diverse*.

divide

Class: arithmetic operation

Property: binary pervasive

Usage: `A divide B  A / B  A div B  divide A B`

See Also: product, quotient, mod, reciprocal

The operation *divide* returns the result of dividing two numeric atoms. It coerces the type of the atoms to be real and gives a real number result. If *B* is a numeric zero, the result is the fault *?div*.

If one argument is numeric and the other is a fault or if both arguments are the same fault, the answer is the fault. In all other cases when one or more of the arguments is not numeric, the result is the arithmetic fault *?A*.

```
     R gets 1 2 2.5 `a "abc ??error
1 2 2.5 `a "abc ??error

     R outer divide R
    1.    0.5    0.4 ?A ?A ?error
    2.     1.    0.8 ?A ?A ?error
   2.5   1.25     1. ?A ?A ?error
?A     ?A     ?A     ?A ?A ?A
?A     ?A     ?A     ?A ?A ?A
?error ?error ?error ?A ?A ?error
```

The above example illustrates all combinations of atom types for the two arguments to *divide*.

Equations

```
1.0 divide A = reciprocal A
A divide B = A times reciprocal B (within roundoff error)
```

dlllist

Class: system expression

Usage: `Dlllist`

See Also: calldllfun, registerdllfun

The expression *Dlllist* provides a list of the current routines that have been registered.

Down

Class: recursion transformer

Usage: `DOWN [test, endf, structf, joinf] A`

See Also: recur, across

DOWN is a general transformer that recurs over the depth of an array to some arbitrary level. *DOWN* has four operation arguments: *test* tests when the recursion has gone as deep as necessary, *endf* is applied to the argument that satisfies *test*, *structf* rearranges the argument before recurring on each item, and *joinf* combines the results of the recursion on the items.

```
     DOWN [ atomic, 0 minus, pass, pass]  3 -4 (5 6)
+--+-+-----+
|-3|4|-5 -6|
+--+-+-----+
     DOWN [ simple, dosum, pack, pass ] 3 4 5 6
18
     DOWN [ atomic, 0 first, pass, 1 plus max ] [2, [3, 4, [5],8],24]
3
```

Definition

```
DOWN IS TRANSFORMER test endf structf joinf OPERATION A {
   Candidates := [A];
   Results := Null;
   WHILE not empty Candidates DO
     Candidates B := [front, last] Candidates;
     IF B = "Start THEN
        Candidates Shp := [front, last] Candidates;
        N := prod Shp;
        IF N = 0 THEN N := N + 1; ENDIF;
        Results Items := opposite N [drop, take] Results;
        Results := Results append joinf (Shp reshape Items);
     ELSEIF test B THEN
        Results := results append endf B;
     ELSE
        B := structf B;
        Candidates := Candidates link [shape B, "Start] link reverse list B;
       IF empty B THEN Candidates := Candidates append first B; ENDIF;
ENDIF;
   ENDWHILE;
   first Results }
```

Equations

```
   DOWN [test, endf, structf, joinf ] A = FORK [test, endf, joinf EACH DOWN [test, endf,
structf, joinf] structf ] A
   opposite A = DOWN [ atomic, 0 minus, pass, pass] A
   sum A = DOWN [ simple, dosum, pack, pass ] A
   depth A = DOWN [ atomic, 0 minus, pass, pass] A
```

drop

Class: selection operation

Property: binary

Usage: `A drop B   drop A B`

See Also: take, rest

The operation *drop* selects the items of an array after a specified number of items have been dropped. If *B* is a list and *A* is a non-negative integer, the result is the list formed from dropping *A* items from the front of *B*. If *A* is a negative integer, the result is formed by dropping *abs A* items from the right end of *B*.

If *B* is a table and *A* is a pair of non-negative integers, the result is lower right corner of *B* that remains after dropping the number of rows and columns indicated by *A* from the upper left corner of *B*. If one or both items of *A* are negative, the dropping occurs from the other end of the extent of the corresponding axis.

For a higher dimensional array *B*, *tally A* must equal *valence B*; and the result is obtained by dropping from the front or back of the extents along each axis. If *B* is a single and *product A* is zero, the result is *B*; otherwise the result is an empty array with valence of tally *A*.

```
     2 drop "Able "Baker "Charlie "Dog
Charlie Dog

     T1 := tell 2 4
+---+---+---+---+
|0 0|0 1|0 2|0 3|
+---+---+---+---+
|1 0|1 1|1 2|1 3|
+---+---+---+---+

     1 2 drop T1
+---+---+
|1 2|1 3|
+---+---+

     1 -2 drop T1
+---+---+
|1 0|1 1|
+---+---+
```

Definition

```
     drop IS OPERATION A B {
        IF not and EACH isinteger A THEN
           fault '?left arg of drop must be integers'
        ELSEIF valence B = 0 THEN
           IF product A > 0 THEN
              (tally A reshape 0) reshape B
           ELSE
              B
           ENDIF
        ELSEIF tally A ~= valence B THEN
           fault '?valence error in drop'
        ELSE
           ((A < 0) + (-1 * (A >= 0) )) * (shape B - abs A max 0) take B
        ENDIF }
```

Equations

```
   0 times shape B drop B = B
   tally B drop list B = Null
```

dropright

Class: selection operation

Usage: `A dropright B   dropright A B`

See Also: drop, takeright, front

The operation *dropright* is an obsolete operation that drops items from the ends of extents. It is provided to retain compatibility with earlier versions of Q'Nial.

Definition

```
dropright IS OPERATION A B { opposite A drop B }
```

each

Class: distributive transformer

Usage: `EACH f A`

See Also: eachall, eachboth, eachleft, eachright, iterate, unary pervasive

The transformer *EACH* forms an operation, called the *EACH* transform of *f*, that applies *f* to every item of the argument *A*. The shape of the result of the application of *EACH f* to *A* is the same shape as *A*.

```
     EACH reverse ('abc' 'def' 'ghi')
+---+---+---+
|cba|fed|ihg|
+---+---+---+

     EACH first tell 3 4
0 0 0 0
1 1 1 1
2 2 2 2
```

EACH is used to distribute an operation *f* across an entire array *A*. Its use avoids the need to explicitly allocate space for the result and to write an explicit loop to apply *f* to each item of *A* individually.

Equations

```
shape EACH f A = shape A
I in grid A ==> I pick (EACH f A) = f (I pick A)
(EACH f) (EACH g) A = EACH (f g) A
EACH f A = shape A reshape (f first A hitch EACH f rest A)
EACH f single A = single f A
EACH f solitary A = solitary f A
EACH f list A = list EACH f A
isshape A ==> EACH f (A reshape B) = A reshape EACH f B
EACH f link A = link EACH EACH f A
EACH EACH f cart A = cart EACH EACH f A
```

eachall

Class: distributive transformer

Usage: `EACHALL f A`

See Also: each, eachboth, eachleft, eachright, pack, multi pervasive

The transformer *EACHALL* forms an operation which applies *f* to arrays formed by selecting the items of the items of *A* in corresponding positions, assuming that the items of *A* are all of the same shape.

When the items of *A* are not all the same shape, *A* is examined to see if all the items with tally greater than one have the same shape. If so, all the items of tally 1 are replicated to that shape. If not, the fault *?conform* is returned. *EACHALL* is implicitly applied in multi pervasive operations.

```
     A := 9 [2] (4 4 reshape (5+tell 16))
+-+-+-----------+
|9|2| 5  6  7  8|
| | | 9 10 11 12|
| | |13 14 15 16|
| | |17 18 19 20|
+-+-+-----------+

     pack A
+------+------+------+------+
|9 2 5 |9 2 6 |9 2 7 |9 2 8 |
+------+------+------+------+
|9 2 9 |9 2 10|9 2 11|9 2 12|
+------+------+------+------+
|9 2 13|9 2 14|9 2 15|9 2 16|
+------+------+------+------+
|9 2 17|9 2 18|9 2 19|9 2 20|
+------+------+------+------+

     (EACH sum) pack A
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31

     EACHALL sum A
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31

     sum A
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31
```

The example shows that *EACHALL* is defined in terms of *pack* and is implicitly used in the multi pervasive operation *sum*.

Definition

```
     EACHALL IS TRANSFORMER f OPERATION A { EACH f pack A }
```

Equations

```
   shape EACHALL f A = shape pack A
   I in grid pack A ==> I pick (EACHALL f A) = f (I pick pack A)
```

eachboth

Class: distributive transformer

Usage: `A EACHBOTH f B   EACHBOTH f A B`

See Also: each, eachall, eachleft, eachright, pack, binary pervasive

The transformer *EACHBOTH* forms an operation which applies *f* to pairs formed by selecting the items of *A* and *B* in corresponding positions, assuming that *A* and *B* have the same shape.

When *A* and *B* do not have the same shape, if either *A* or *B* has only one item, that item is replicated to the shape of the other argument; otherwise, the fault *?conform* is returned.

EACHBOTH is used implicitly in all binary pervasive operations.

```
     7 2 1 EACHBOTH post 4 5 3
+-+-+-+
|7|2|1|
|4|5|3|
+-+-+-+
```

Definition

```
     EACHBOTH IS TRANSFORMER f OPERATION A B { EACHALL f A B }
```

Equations

```
   shape (A EACHBOTH f B) = shape (A pack B)
   I in grid pack A B ==> I pick (A EACHBOTH f B) = f (I pick (A pack B))
```

eachleft

Class: distributive transformer

Usage: `A EACHLEFT f B   EACHLEFT f A B`

See Also: each, eachall, eachboth, eachright

The transformer *EACHLEFT* forms an operation which applies *f* to pairs formed by pairing the items of *A* with *B*. The shape of the result is the shape of *A*.

```
     2 3 4 EACHLEFT reshape 5
+---+-----+-------+
|5 5|5 5 5|5 5 5 5|
+---+-----+-------+
```

```
     (2 3 reshape 'abcdef') EACHLEFT hitch '123'
+----+----+----+
|a123|b123|c123|
+----+----+----+
|d123|e123|f123|
+----+----+----+
```

Definition

```
     EACHLEFT IS TRANSFORMER f OPERATION A B { EACH (B CONVERSE f) A }
```

Equations

```
   shape ( A EACHLEFT f B ) = shape A
   I in grid A ==> I pick (A EACHLEFT f B) = f (I pick A) B
   A EACHLEFT f B = A EACHBOTH f single B
   single A EACHLEFT f B = single (A f B)
   solitary A EACHLEFT f B = solitary (A f B)
   list A EACHLEFT B = list (A EACHLEFT f B)
   isshape S ==> (S reshape A) EACHLEFT f B = S reshape (A EACHLEFT f B)
```

eachright

Class: distributive transformer

Usage: `A EACHRIGHT f B   EACHRIGHT f A B`

See Also: each, eachleft, eachall, eachboth

The transformer *EACHRIGHT* forms an operation which applies *f* to pairs formed by pairing *A* with the items of *B*. The shape of the result is the shape of *B*.

```
     2 EACHRIGHT reshape 3 4 5
+---+---+---+
|3 3|4 4|5 5|
+---+---+---+

     `X EACHRIGHT hitch (2 3 reshape 'ab' 'cd')
+---+---+---+
|Xab|Xcd|Xab|
+---+---+---+
|Xcd|Xab|Xcd|
+---+---+---+
```

Definition

```
EACHRIGHT IS TRANSFORMER f OPERATION A B { EACH ( A f ) B }
```

Equations

```
I in grid B ==> I pick (A EACHRIGHT f B) = f A (I pick B)
shape ( A EACHRIGHT f B ) = shape B
A EACHRIGHT f B = single A EACHBOTH f B
A EACHRIGHT f single B = single (A f B)
A EACHRIGHT f solitary B = solitary (A f B)
A EACHRIGHT f list B = list(A EACHRIGHT fB)
isshape S ==> A EACHRIGHT f (S reshape B) = S reshape (A EACHRIGHT f B)
```

edit

Class: system operation

Usage: `edit Filename`

See Also: defedit, host, loaddefs

The operation *edit* passes control to the standard editor, requesting it to edit the file named by the string or phrase *Filename*. The editor used is determined by the environment variable *EDITOR* or by a default chosen for each version. When the editing task is completed, the editor returns control to Q'Nial. The result is the fault *?noexpr*.

```
edit "test.ndf
edit 'data_records'
```

If *edit* is used to edit a file of Nial definitions, the operation *loaddefs* must be executed on return from the editor to load the definitions.

empty

Class: structure testing operation

Property: predicate

Usage: `empty A`

See Also: null, in, simple, single

The operation *empty* tests whether or not an array *A* has any items. If *A* has no items, it returns *true*; otherwise, *false*. The predefined expression *Null* is empty as is any array with a zero in its shape.

Since there are empty arrays with more than one dimension, it is better to test for an empty array by using *empty* rather than by direct comparison with *Null*.

```
     empty Null
1

     empty (0 3 2 reshape 5)
1

     empty solitary 5
0

     empty solitary Null
0
```

The last two examples show that a solitary is not empty even if the item it contains is empty.

Definition

```
empty IS OPERATION A { tally A = 0 }
```

Equation

```
empty A = 0 in shape A
```

equal

Class: logic operation

Property: predicate

Usage: `A equal B   A = B   equal A`

See Also: unequal, match, mate, gte, lte

The operation *equal* is normally used to compare two arrays *A* and *B* to see whether or not they are identical. It returns *true* if they are equal; *false* otherwise. (Two nonempty arrays are identical if they have identical shapes and hold identical items at each location. Two empty arrays are identical if they have identical shapes.) *Empty* is extended to arbitrary arrays by returning *true* if all items of an array are identical.

The symbol = is a synonym for equal and can be used in both infix and prefix application. In the *Equations* sections of the dictionary entries, the symbol = is used to separate two sides of an equation. To test such an equation, it may be necessary to replace the expression following = with the same expression in parentheses in order to force correct parsing of the equation as a Nial expression.

```
     2 3 4 = [2,3,4]
1
     Null = ''
1
     2 (3 4) = (2 3) 4
0
     equal EACH shape tell 2 3 4
1
```

The first example shows that a triple formed using strand notation is identical to that formed using brackets notation. The second shows that *Null* is equal to the empty string. The third shows that grouping items of a list in different ways creates different arrays. The last shows the convenience of the more general form of *equal*.

Equations

```
A equal B <==> shape A = shape B
A equal B and (I in grid A) ==> I pick A = I pick B
```

equations

Class: concept

The equations in the Nial Dictionary and in the on-line help provide an abbreviated way of stating properties of the term or object being described. They could be expanded into an explanation in English but that would lengthen the manual considerably.

The equations use variables such as *A*, *B* and *C* that take on array values; and variables such as *f*, *g* and *h* that denote operations that map arrays to arrays. An equation holds for all arrays and for all computable operations without side effects, unless a specific qualification is made. Thus, the equation:

```
(EACH f) (EACH g) A = EACH (f g) A
```

says that for all operations *f* and *g* and all arrays *A*, the use of the *EACH* transform of *f* on the result of the use of the *EACH* transform of *g* on *A* has the same value as the use of the *EACH* transform of the composition of *f* and *g* on *A*. In mathematical terms, the *EACH* transformer distributes over operation composition.

The symbol = is used in its mathematical sense in equations and separates two Nial expressions. The symbol = has higher precedence than the two expressions it is separating. Where the Nial equivalent to = is needed to state the equation, the term *equal* is used.

To test an equation using Q'Nial, in order to force the correct parsing of the equation as a Nial expression, it may be necessary to replace

```
Expr1 = Expr2
```

with

```
Expr1 = ( Expr2 )
```

The equality used in equations assumes that both sides of the equations compute without triggering a fault and produce equal arrays; or both produce the same fault value if fault triggering is off. In some cases, the equality is inexact due to roundoff errors.

The symbol =*f*= is used to denote an equality where both sides produce the same non-fault value; but in some cases one or both of the sides may fault and the equality is no longer valid.

Some of the equations are qualified by a constraint on the variables. The constraint is written in English or as a conditional expression in Nial that must hold for the equation to be true. In a statement of the form

```
Expr ==> Eqn
```

Expr is the qualification written as a Nial boolean expression, the symbol ==> is used for "implies", and Eqn is the equation that holds under the qualification.

The symbol <==> denotes an if and only if implication. Thus a statement of the form

```
Expr1 <==> Expr2
```

states that both expressions have the same truth value; either both are true or both are false.

Reading the Equations

The equations related to *abs*, the operation that finds the absolute value of a number, are:

```
abs A = EACH abs A
shape abs A = shape A
abs abs A = abs A
```

The first two come from the property that *abs* is *unary pervasive*. The first one says that applying *abs* to an array *A* is the same as applying *abs* to the items of array *A*. It also implies that the shape of the result of *abs A* is the same as shape of *A* since *EACH* transforms always preserve shape. The second equation says that the shape of the result of *abs A* is the same as the shape of *A*. The third equation indicates that subsequent applications of *abs* after the first do not change the result.

The equations related to *minus*, denoted by -, are:

```
A - B = EACH - pack A B
shape (A - B) = shape pack A B
A - B = A + opposite B
```

The first equation says that pair of arrays *A* and *B* are packed at each level of nesting to bring corresponding items together. Then, the items are subtracted.

The second equation states that the shape of the result of subtracting *A* from *B* is the shape formed by packing *A* and *B*. The third equation states that subtracting *A* from *B* is equivalent to adding the opposite of *B* to *A*.

The equations related to *hitch* are as follows:

```
A hitch B = A hitch list B
list (A hitch B) = A hitch B
```

The first says that *hitch* treats its right argument as though it were a list, and the second states that the result of *hitch* is a list.

The following equations illustrate the use of qualifications:

```
atomic A ==> single A = A
not empty A ==> mix rows A = A
diverse A <==> cull A = list A
```

The first equation says that if *A* is atomic, the single of *A* is equal to *A*. That is, the single of an atom is the atom. The second equation says that if *A* is not empty, forming the rows of *A* and applying *mix* to recombine them, results in the original array. The third equation says that if *A* is diverse, the cull of *A* is equal to the list of *A*; and that if *A* is not diverse, the cull of *A* is not equal to the list of *A*.

There are many identities that hold for all arrays in Nial. The equations are shown in the on-line help for each topic. Three general equations are the following:

A unary pervasive operation f satisfies the equation:

```
f A = EACH f A
```

A binary pervasive operation f satisfies the equation:

```
A f B = A EACHBOTH f B
```

A multi pervasive operation f satisfies the equations:

```
f A = EACHALL f A
f A = REDUCE f A
```

erase

Class: system operation

Usage: `erase Nm`

See Also: symbols

The operation *erase* is used to remove unwanted global definitions and variables from the workspace. The argument *Nm* must be a phrase or string giving the object to be removed. Predefined names and names in local scopes cannot be removed.

When a name has been erased, it remains in the symbol table with its original role. It can be redefined in the same role or, if a variable, reassigned a value by a subsequent action. A name cannot change its role once established.

```
     average IS divide[sum,tally];
     calc IS EACH average;
     Var3 := calc (2 3 4) (5 6 7)
3. 6.

     EACH erase "average 'Var3' "sum
?noexpr ?noexpr ?system name
```

In the example above, the operation *sum* cannot be erased as it is a predefined operation. In the next three examples below, faults are returned because of missing variables or operations that were erased. The effect of erasing an object is not propagated to other definitions referring to the erased name.

```
     Var3
?no_value

     average count 10
?missing_op

     calc (2 3 4) (5 6 7)
?missing_op ?missing_op

     average := 5 6 7
?invalid assignment: AVERAGE :=  <***> 5 6
```

In the last example, an attempt is made to redefine the role of *average* and Q'Nial indicated that *average* is not a suitable name for such an assignment.

eraserecord

Class: nial direct access file operation

Usage: `eraserecord Filenum N`

See Also: readrecord, writerecord, readarray, writearray, open, close, filetally

The operation *eraserecord* is used to erase component *N* of direct access file designated by the number *Filenum*. In a file with record components, *eraserecord* replaces the erased component with an empty string. In a file with array components, it replaces the erased component with the fault *?missing*.

N may be a single record number or a list of record numbers.

If the last component in a file is erased, any immediately preceding components that are empty in a record component file or contain the fault *?missing* in an array component file are also removed. The file tally, which records the number of records in the file, is adjusted accordingly.

The operation returns the fault *?noexpr* or a fault indicating improper use of the operation.

```
File_num := open "Names "d ;
eraserecord File_num 23 ;
```

eval

Class: evaluation operation

Usage: `eval Pt`

See Also: execute, scan, parse, value

The operation *eval* provides access to the underlying expression evaluator of Q'Nial. The argument to *eval* must be a parse tree returned by *parse* or *getdef*, a named expression or a cast which represents a Nial expression. The effect is to evaluate the expression and return the resulting value. As seen below, the parse tree can be one formed by using the cast mechanism or one built explicitly using *parse*.

```
     eval parse scan '23 + 45'
68

     eval !(23 + 45)
68
```

The next example shows that *eval* can be used to evaluate a named expression using either the cast or the phrase of its name.

```
     A := 100 ;
     Addone IS (A := A + 1);
     (eval !Addone)(eval "Addone)
101 102

     (eval !A)(eval "A)(value "A)
102 ?not an expression 102
```

The last example shows that *eval* can be used to evaluate the cast of a variable but not a phrase naming the variable. The operation *value* should be used instead.

Equations

```
S a string holding a Nial expression ==> eval parse scan S = execute S
Pt a parse tree ==> eval parse deparse Pt = eval P
```

except

Class: selection operation

Property: binary

Usage: `A except B   except A B`

See Also: cull, sublist, notin

The operation *except* returns the list of items of *A* that are not in *B*. It corresponds to set difference if *A* and *B* are viewed as set representations. The result is a list with items in the same order as in *A*.

```
     count 10 except 3 5 7
1 2 4 6 8 9 10

     'a list of words' except 'aeiou'
 lst f wrds

     tell 2 3 except [0 0]
+---+---+---+---+---+
|0 1|0 2|1 0|1 1|1 2|
+---+---+---+---+---+

     tell 2 3 except (0 0)
+---+---+---+---+---+---+
|0 0|0 1|0 2|1 0|1 1|1 2|
+---+---+---+---+---+---+
```

The last two examples show that if the left argument has nonatomic items, to remove a single non-atomic array, the right argument must be made into a solitary list. Otherwise the items of the right argument are used.

Definition

```
     except IS OPERATION A B { A EACHLEFT notin B sublist A }
```

Equations

```
   list (A except B) = A except B
   list A except list B = A except B
   (A except B) except C = A except (B link C)
   A except cull B = A except B
```

Pragmatics

The operation *except* uses an internal sort on its arguments to reduce the algorithmic complexity of the above definition. It executes considerably faster if its arguments have already been sorted.

execute

Class: evaluation operation

Usage: `execute S`

See Also: eval, parse, scan, tonumber, apply, value

The operation *execute* evaluates a Nial expression given as a string *S*. The result of the evaluation is returned. The execution of the string takes place in the environment where execute is applied.

If the string contains a name, the local meaning of that name will be used if a local meaning exists. If execution of the string results in the creation of a new variable or defined name, the new object is placed in the global environment.

If a string being executed contains an integer representation that would be converted to an integer outside the range for the computer, the result is converted to the corresponding real number.

```
     execute '23 + 40045'
40068.

     A := count 5; opname := 'product';
     execute link opname ' A'
120
```

The last example shows that Nial program text can be constructed under program control and then executed. This feature is useful in applications where responses are generated by combining text elements based on the user input and the current state of the computation. The technique is useful for applications involving computer assisted instruction and knowledge based systems.

It is more efficient to use *eval* to evaluate the cast of an expression than to use *execute* to evaluate the corresponding string. In the former approach the *scan* and *parse* is done once, whereas in the latter it is repeated on each execution.

Definition

```
     execute IS OPERATION A { eval parse scan A }
```

Equations

```
   execute readscreen Prompt = read Prompt
   Txt, a Nial expression ==> eval !(Txt) = execute 'Txt'
   S a string holding a Nial expression ==> execute canonical S = execute S
```

where

```
  canonical IS link descan deparse parse scan.
```

exp

Class: scientific operation

Property: unary pervasive

Usage: `exp A`

See Also: ln, power, log

The operation *exp* implements the exponential function of mathematics. It produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	exponential of the corresponding real
integer	exponential of the corresponding real
real	exponential of A
character	fault ?exp
phrase	fault ?exp
fault	argument A

```
     exp  o  1  0.5  `a  "abc  ??error
1. 2.71828 1.64872 ?exp ?exp ?error
```

Equations

```
exp ln A = A   (within roundoff error)
exp A = exp 1.0 power A   (within roundoff error)
```

expression

Class: concept

See Also: operation, transformer

The term **expression** is used in its most general sense to describe a program fragment that denotes one of the three primary objects of Nial: an *array*, an *operation*, or a *transformer*. However, in most contexts we use the term as an abbreviation of an *array expression*.

An array expression denotes an array value. That is, it is a program fragment that when evaluated in the proper context will produce an array. The predefined expressions of Q'Nial either produce a constant value, or they carry out some system action and return the fault value *?noexpr*.

The control constructs of Nial are array expressions made up of keywords, simple-expressions and expression-sequences.

A named expression is either a predefined expression or an expression that has been given an explicit name using the *IS* definition mechanism.

expression sequence

Class: syntax

See Also: expression

An **expression-sequence** is the main construct used for program text that produces a value. It consists of one or more expressions separated by semi-colons and possible followed by a semi-colon.

```
expression-sequence ::= expression
   { ; expression } [ ; ]
```

The expressions in an expression-sequence are evaluated in left-to-right order. If the sequence does not terminate with a semicolon, the array returned is the result of the last expression. If the sequence does end with a semicolon, the array returned is the fault *?noexpr*. At the top level loop, if the array returned is the fault *?noexpr*, it is not displayed.

exprs

Class: system expression

Usage: `Exprs`

See Also: symbols

The expression *Exprs* returns a list of phrases giving the names of all user defined expressions in the workspace.

Definition

```
Exprs IS {
   Names Roles := pack symbols 0;
   "expr match Roles sublist Names }
```

extent

Class: concept

See Also: shape

The term **extent** is used to describe the length of an axis of an array in a particular dimension. Thus a 4 by 6 table is said to have extent 4 in the first dimension and extent 6 in the second dimension.

external declaration

Class: syntax

See Also: action, role

```
external-declaration ::= identifier IS
   EXTERNAL ( EXPRESSION
                    | OPERATION
                    | TRANSFORMER
                    | VARIABLE )
```

An external declaration assigns a role to a name, allowing it to be used in a definition before its own definition is given. This mechanism is useful for creating mutually recursive definitions. An external declaration is made only in the global environment.

If the name is already defined with the same role, the declaration has no effect. If the name has another role, a fault is reported. If the name is not currently defined, a default object is associated with it.

false

Class: constant expression

Usage: `False`

See Also: true, isboolean

The constant expression *False* denotes the boolean atom for *false*, which Nial also denotes by *o*. It is the result of comparing two arrays that are not identical for equality.

```
     False (not False)
ol
```

Equations

```
   tally False = 1
   shape False = Null
   single False = False
   not False = True
```

fault

Class: conversion operation

Usage: `fault S`

See Also: isfault, phrase, quiet_fault, settrigger, string

The operation *fault* converts a string or phrase into the fault value with the string as its message. If the argument is not a string, phrase or a fault, the result is the fault *?type error*.

Fault is used to construct faults that contain blanks or other characters than cannot appear in a literal fault or ones constructed from a message provided from the host system.

By default, fault triggering is on when the interpreter is initialized. This causes the creation of a fault to interrupt the flow of execution. The operation *quiet_fault* can be used to create a fault value without triggering an interrupt.

```
     settrigger o;
     fault '?missing data'
?missing data

     fault "Notastring
Notastring

     fault 3.4
?type error
```

The convention is that all faults generated by Q'Nial internally begin with the character *?*. The second example shows that this is not a requirement.

Equations

```
   isfault F ==> fault string F = F
   isstring S ==> string fault S = S
```

fault triggering

Class: concept

See Also: fault values, settrigger, quiet_fault

Nial assumes that every computation that terminates results in an array value. However, there are many cases where a computation does not have a sensible answer. If division by zero occurs, for example, there is no suitable number to return. Nial uses special atomic arrays called faults to indicate such results. For division by zero it is ?div.

Q'Nial has two ways of handling a fault: either a trigger mechanism is executed that causes an interruption when a fault is created, or during execution of a defined operation, expression or transformer the fault is treated as a normal atomic array.

When Q'Nial is invoked, the fault triggering mechanism is turned on by default. (This effect can be suppressed by using an option in the setup for each version). During execution, the state of the triggering mechanism can be turned on or off using the operation *settrigger*. The operation *quiet_fault* can be used to create a fault without causing fault triggering.

If fault triggering is set and a fault is generated during execution of a defined operation, execution is interrupted. On an interruption caused by a fault, a display message appears giving the call stack of definitions currently executing and the line of text that caused the fault. For example, the definition:

```
     foo is op A B { A / B + 1 }
```

followed by the evaluation of the expression

```
     foo 3 0
```

results in the output:

```
------------------------------------------------------------
    Fault interruption loop:  enter expressions or
      type: <Return>   to jump to top level
    current call stack :
foo
      ?div triggered in : ... A / B
------------------------------------------------------------
>>>
```

where the string '>>>' is a special prompt indicating that a fault has occurred and execution has been interrupted. The prompt permits you to query the value of variables in the expression and its surrounding computation or to view the operation that has triggered the fault. The above session might continue as:

```
>>> see "foo
foo IS OPERATION A B {
    A / B + 1 }
>>> A
3
>>> B
0
>>>
```

A variable in a definition that called the current one can be referenced by preceding the variables name by the definition name and a colon, e.g. *G:X* denotes variable *X* in definition *G*. You can execute any expressions you want at the prompt. A useful thing to do is to see the definition that has interrupted. When you are ready to resume, reply to the prompt with a Return and control returns to the interactive loop.

fault values

Class: concept

The Role of Faults

A **fault** is a special kind of atomic value used by Q'Nial to signal special values or to indicate that an operation has been given an argument it cannot handle in a normal way. The special value faults are:

Fault	Meaning
?noexpr	indicates that no answer is expected
?eof	end of file indication
?I	Zenith which is greater than all atoms
?O	Nadir which is less than all atoms

filelength

Class: host direct access file operation

Usage: `filelength Filename`

See Also: readfield, writefield

The operation *filelength* returns the length in bytes of the host file named by string or phrase *Filename*. It is used in conjunction with *readfield* and *writefield* in processing host files.

```
Len := filelength "Myfile;
Data := readfield "Myfile 0 Len;
```

In the example, *filelength* is used to determine the size of the file and *readfield* is used to read it in as raw byte data. If the file corresponds to a text file, the data will include end of line indications appropriate for the host system.

filepath

Class: file operation

Usage: `filepath Nms`

See Also: separator

The operation *filepath* converts a list of phrases or strings giving the names of directories and a filename into the system dependent string that access the specific file. This operation is useful in writing system independent programs that will work with either UNIX or DOS (and WINDOWS) path name conventions.

Example:

```
filepath ".. "aitools "newfnfl "frames
```
returns the string
```
../aitools/newnfl/frames
```
on a UNIX version of Q'Nial, and returns
```
..\aitools\newnfl\frames
```
on a PCDOS version.

Definition

```
filepath is op Nms {
  front link (EACH string Nms EACHLEFT append Separator ) }
```

filestatus

Class: file expression

Usage: `Filestatus`

See Also: status, open, close, filetally

The expression *Filestatus* gives information on the files currently open in a Q'Nial session. It returns a list of triples, one for each open file, giving the file number as an integer, the filename as a phrase and the mode as a character.

The modes are *r*, *w*, *a*, *d*, *pr*, *pw* and *c*, standing for *read*, *write*, *append*, *direct*, *pipe_read*, *pipe_write* and *communications* respectively.

The files for standard input, standard output and standard error are opened with modes of r, w and w respectively using file numbers 0, 1 and 2.

```
Fnum := open "F "a;
Filestatus
+---------+----------+----------+-----+
|0 stdin r|1 stdout w|2 stderr w|3 F a|
+---------+----------+----------+-----+
```

filetally

Class: nial direct access file operation

Usage: `filetally Filenum`

See Also: open, readarray, writearray, filelength

The operation *filetally* returns the number of records in the file designated by *Filenum*, an integer returned by an earlier call to *open*. The filetally is one higher than the highest component number of a record or array that is written but not erased.

```
     Fn := open "newfile `d;
     writearray Fn (9 99) ('The' 'End');
     filetally Fn
100
```

filter

Class: selection transformer

Usage: `FILTER f A`

See Also: each, sublist

The transformer *FILTER* is given a predicate operation *f* that is used to select items from *A*. The result of applying *FILTER f* is to produce a list of the items of *A* that satisfy the predicate *f*.

```
     FILTER (0<) (4 -2 3 -5)
4 3

     FILTER (not isphrase) (37 4.5 "cat)
37 4.5
```

Definition

```
     FILTER IS TRANSFORMER f OPERATION A { EACH f A sublist A }
```

Equations

```
   tally FILTER f A = sum EACH f A
```

find

Class: search operation

Property: binary

Usage: `A find B   find A B`

See Also: findall, seek, in, gage, sortup

The operation *find* returns the address of the first occurrence of *A* as an item of *B*, searching *B* in row major order. If *A* does not occur in *B*, the result is the *gage* of *shape B*. The result of *find* is an integer if *B* is a list; and a list of integers of tally equal to *valence B* otherwise.

```
     set "diagram ;
     3 find 56 34 3 23 3 57 3
2
```

The result of find is 2.

```
     `a find 'hello world'
11

     2 3 find count 3 4
+-+-+
|1|2|
+-+-+
```

In the second example, the character a is not in the string 'hello world' and the result is the tally of 'hello world'. In the last example, the result is a list of two integers because the valence of count 3 4 is 2.

Definition

```
     find IS OPERATION A B { gage first ( A findall B append shape B ) }
```

Equations

```
   A seek B = A [in,find] B
   A in B ==> A find B pick B = A
   I in grid A and diverse A ==> I pick A find A = I
```

Pragmatics

The operation *find* uses a linear search on the items of *B* if the array has not been sorted, or uses a binary search algorithm if it has. The latter fact suggests that an array that is searched frequently should be kept in lexicographical order by applying *sortup* to it when it is created or changed.

findall

Class: search operation

Property: binary

Usage: `A findall B   findall A B`

See Also: find, choose, seek, in

The operation *findall* returns a list of the addresses of all occurrences of *A* as an item of *B*, searching *B* in row major order. If *A* does not occur in *B*, the result is the empty list *Null*. The result of *findall* is a list of integers if *B* is a list and a list of pairs if *B* is a table.

```
     3 findall 56 34 3 23 3 57 3
2 4 6

     X := 3 4 reshape 1 7 3 2 3 4 3 2 6 3
1 7 3 2
3 4 3 2
6 3 1 7

     3 findall X
+---+---+---+---+
|0 2|1 0|1 2|2 1|
+---+---+---+---+
```

```
     `a findall 'hello world'

     1 1 findall tell 3 4
+---+
|1 1|
+---+
```

In the first example *findall* returns the list of integers where 3 is found in the list; in the second a list of pairs is returned. The last example shows that if only one occurrence is found the result is a solitary list.

Definition

```
     findall IS OPERATION A B { A EACHRIGHT equal B sublist grid B }
```

Equations

```
   (list A) findall B = A findall B
   list (A findall B) = A findall B
   A in B ==> A find B = first (A findall B)
```

Pragmatics

The operation *findall* uses a linear search on the items of *B* if the array has not been sorted, or uses a binary search algorithm if it has. The latter fact suggests that an array that is searched frequently should be kept in lexicographical order by applying *sortup* to it when it is created or changed.

first

Class: selection operation

Usage: `first A`

See Also: last, second, third, pick, take

The operation *first* returns the first of the items of *A*. If *A* is empty, it returns the fault *?address*.

```
     first 4 5 6
4

     first count 3 4 5
1 1 1

     first Null
?address
```

Definition

```
     first IS OPERATION A { 0 pick list A }
```

Equations

```
   first single A = A
   first solitary A = A
   first list A = first A
   first (S reshape A) = first A
   (valence A reshape 0) pick A = first A
```

flip

Class: data rearrangement operation

Usage: `flip A`

See Also: pack

The operation *flip* is a synonym for *pack*. It has been retained for compatibility with earlier versions.

floor

Class: arithmetic operation

Property: unary pervasive

Usage: `floor A`

See Also: ceiling

The operation *floor* produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	corresponding integer
integer	argument A
real	next lower integer, or the fault ?A if the result is outside the range of integers
char	fault ?A
phrase	fault ?A
fault	argument A

```
     floor  1  -2  3.5  `a  "abc  ??error
1 -2 3 ?A ?A ?error

     floor  3.5  -4.6  7.0  25.3e20
3 -5 7 ?A
```

Equation

```
   floor A = opposite ceiling opposite A
```

fold

Class: applicative transformer

Usage: `N FOLD f A   FOLD f N A`

See Also: iterate, each

The transformer *FOLD* modifies an operation *f* to one that takes a pair of arguments consisting of an

integer *N* and an arbitrary array *A*. The result of applying *N FOLD A* is to apply *f*, *N* times, applying it first to *A* and subsequently to the result of the previous application.

```
     3 FOLD rest 4 5 6 7 8 9
7 8 9
     2 FOLD sum (2 3 4) (5 6 7)
27
     sum sum (2 3 4) (5 6 7)
27
```

Definition

```
     FOLD IS TRANSFORMER f OPERATION N A {
        IF N > 0 THEN
           N - 1 FOLD f ( f A )
        ELSE
           A
        ENDIF }
```

Equations

```
   0 FOLD f A = A
   1 FOLD f A = f A
   2 FOLD f A = f f A
```

fork

Class: control structure transformer

Usage: `FORK [f,g,h] A   FORK [f,g,h,i,j,...z] A`

See Also: iterate, if-expr

The transformer *FORK* implements a conditional functional mechanism corresponding to the if-expression. The argument to *FORK* must be an atlas of length two or more. The operation in the first position is a predicate. If there are exactly three items in the atlas, the first is a predicate; the second is applied to *A* if the predicate returns *true*; and the third is applied to *A* if the predicate returns *false*. That is, if *f A* is *true*, the result is *g A*; otherwise it is *h A*.

If there are more than three operations in the atlas, they are taken in pairs from left to right. The first of each pair must be a predicate and is applied to *A* in turn until a result is *true*. The result of the transform is the result of applying the second of that pair to *A*. If no application results in *true* and the number of operations in the atlas is an odd number, the last operation is evaluated. Otherwise the result is the fault *?noexpr*.

```
     FORK [atomic, opposite, 1 +] (3 -4 (5 6 7))
+-+--+-----+
|4|-3|6 7 8|
+-+--+-----+

     FORK [isreal,floor,simple,sum,5+] 2 4 6
12
```

Equation

```
   FORK [A first, B first, C first, D first, E first] Null = IF A THEN B ELSEIF C THEN D
ELSE E ENDIF
```

for-loop

Class: control structure

Usage: `FOR Var WITH Exp DO ExpSeq ENDFOR`

See Also: repeat-loop, while-loop, iterate, each

The *FOR-loop* control structure is used to execute the expression sequence *ExpSeq* repeatedly while variable *Var* takes on the values specified as items in the simple expression *Exp*.

```
     FOR X WITH 1. 2. 3. DO
        write (X) (X*X) (X power X) ;
     ENDFOR
1. 1. 1.
2. 4. 4.
3. 9. 27.
```

In the example, X takes on the values 1., 2. and 3. in successive loops and the values of X, X squared and X to the power X are displayed.

fromraw

Class: conversion operation

Usage: `fromraw A B`

See Also: toraw

The operation *fromraw* converts the boolean array *A* to a simple array of the same type as atom *B*.

```
     set "decor;
     fromraw oooooooooollololoolloloo `a
'345'

     fromraw ooooooooooooooooooooooolololloool 0
345

fromraw ooooooooooooooooooooooooooooooooooooloooooooloooooloooooooooooooooo 0.0
34.5

     fromraw lollo 1
lollo
```

Equations

```
   simple A and and (type A match type first A) => A = fromraw (toraw A) (first A)
```

Pragmatics

Fromraw is used to convert raw bit data into atomic data that can be manipulated by Nial. If the data was created from Nial data using *toraw* then it will work in a system independent way. However, if raw byte data is obtained using *readfile* and then converted using *toraw* the host system byte ordering may need to be allowed for.

front

Class: selection operation

Usage: `front A`

See Also: last, rest, first

The operation *front* returns a list of all items but the last of *A*. If *A* is a solitary or is empty, the result is the empty list *Null*.

```
front 3 4 5 6
3 4 5
```

```
     front tell 2 3
+---+---+---+---+---+
|0 0|0 1|0 2|1 0|1 1|
+---+---+---+---+---+
```

Definition

```
     front IS OPERATION A { tally A - 1 max 0 reshape list A }
```

Equations

```
   front A = front list A
   list front A = front A
   front Null = Null
   not empty A ==> front A append last A = list A
   shape A reshape (front A append last A) = A
```

functions in nial

Class: concept

See Also: operation, transformer

A function is a mathematical name for an object that maps an argument in a given domain to a result in a given range. In Nial, an **operation** is an object in the set of functions from the domain of Nial arrays to the range of Nial arrays. Thus, an operation always applies to an array and returns an array.

A **transformer** in Nial is also a function. It domain is Nial operations and its range is also Nial operations. Since its argument is itself a function, a transformer is said to be a *second order* function. A transformer always applies to an operation and results in an operation.

Definitions in which the associated object is a simple-expression are used to name program fragments that return an array value but which do not need parameters. The resulting named-expression behaves like a function having no parameters.

Nial is considered to be a functional language, but it is not purely functional in that it has assignments, loops and other non-functional concepts.

fuse

Class: data rearrangement operation

Property: binary

Usage: `I fuse A   fuse I A`

See Also: transpose, blend, split

The operation *fuse* is used for two distinct purposes. If *I* is simple and contains all the axes of *A* without repetition, the result is an array formed by a permutation of the axes of *A* by *I*. The shape of the result is *I choose shape A*. If *I* is not simple but link *I* is simple and contains all of the axes of *A* without repetition, the result is obtained by diagonalizing along axes that are grouped together, ordering them according to the ordering in *I*.

If link *I* does not contain all the axes or if there are repetitions of the axes in link *I*, the fault *?invalid fuse* is returned.

```
     A := 2 3 4 reshape count 24
1  2  3  4   13 14 15 16
5  6  7  8   17 18 19 20
9 10 11 12   21 22 23 24

     1 2 0 fuse A
1 13   5 17    9 21
2 14   6 18   10 22
3 15   7 19   11 23
4 16   8 20   12 24

     0 (1 2) fuse A
 1  6 11
13 18 23

     [0 1 2] fuse A
1 18
```

The first example is an axis permutation; the last two are diagonalizations.

Equations

```
axes A fuse A = A
transpose A = ( reverse axes A fuse A )
sortup I = axes A and not empty A ==> shape ( I fuse A ) = I choose shape A
```

gage

Class: conversion operation

Usage: `gage A`

See Also: pick, tell, grid

The operation *gage* is used to convert an array of integers into an atomic integer if there is only one item or a list if there is more than one. The integers must be non-negative otherwise the fault *?gage* is produced.

```
     gage 4 5 6
4 5 6

     gage find 5 (tell 10)
6

     gage 3 -4 5
?gage
```

The main purpose of *gage* is to express an list of non-negative integers in the form an address takes, i.e. it converts a solitary integer to the integer itself and leaves all other lists alone.

Definition

```
     gage is OPERATION A {
      IF and EACH isinteger A THEN
         IF tally A = 1 THEN
            IF first A >= 0 THEN first A ELSE ??gage ENDIF
         ELSE
            IF and EACH (0<=) A THEN list A ELSE ??gage ENDIF
         ENDIF
      ELSE
        ??gage ENDIF }
```

getdef

Class: evaluation operation

Usage: `getdef Nm`

See Also: parse, eval, see, defedit

The operation *getdef* retrieves the parse tree associated with the global definition named *Nm*. *Nm* may be a phrase or a string. Only the parse trees associated with global user definitions can be retrieved.

```
     foo IS first ; getdef "foo
+---+------------------------------+
|100|+--+---------------+------+-+|
|   ||11|+--+----------+|3 47 3|1||
|   ||  ||22|2 160 7459||      | ||
|   ||  |+--+----------+|      | ||
|   |+--+---------------+------+-+|
+---+------------------------------+
```

The example shows a parse tree for a simple definition. For definitions of any reasonable size the diagram of the parse tree becomes too large to examine easily.

The primary use of *getdef* is to provide an interface between internal representations and the editing facilities. It is used in *see* and *defedit*. The detailed form of the parse tree is implementation specific and may change in future releases.

getfile

Class: file operation

Usage: `getfile Filename`

See Also: putfile, appendfile, open, readfile

The operation *getfile* returns the records of the text file named by the string or phrase *Filename* as a list of strings. The file must not be open.

```
Recs := getfile "Myfile;
```

Definition

```
getfile IS OPERATION Filename {
   Fnum := open Filename `r;
   IF isfault Fnum THEN
      Fnum
   ELSE
      Lines := '';
      Line := readfile Fnum;
      WHILE Line ~= ??eof DO
         Lines := Lines append Line;
         Line := readfile Fnum;
      ENDWHILE;
      Lines
   ENDIF }
```

getname

Class: evaluation operation

Usage: `getname Triple`

See Also: parse, cast, getdef

The operation *getname* retrieves the variable or definition name associated with a name reference *triple* within a parse tree. The argument must be a triple with first item 2 that has been produced by the Q'Nial parser. The operation is useful when analyzing the structure of a parse tree representation. This is a specialized task and *getname* is used only with considerable knowledge of the internal workings of Q'Nial.

getsyms

Class: evaluation operation

Usage: `getsyms Nm`

See Also: parse, getdef

The operation *getsyms* retrieves the parameters and local variables of a defined operation or expression named by *Nm*. The operation is useful when analyzing the name usage of a definition in the context of analyzing the name interaction among a set of definitions. This is a specialized task and *getsyms* is used only with considerable knowledge of the semantics of Q'Nial.

global environment

Class: concept

See Also: local environment

The **global environment** is the set of associations between names and objects formed in the workspace that are either predefined in Nial or have been created by actions that have taken place during a session.

grade

Class: sorting transformer

Usage: `GRADE f A`

See Also: sort, gradeup, lte, gte

The transformer *GRADE* modifies a comparator *f* to produce an operation that, when applied to *A*, returns an array of addresses of the same shape as *A* that orders *A* according to the comparator. The addresses can be used to select the items of *A* (using *choose*) so that the items are in order according to *f*.

A **comparator** is an operation that compares two arrays and returns *true* if they are in the desired ordering or *false* otherwise. Operations *lte* (<=) and *gte* (>=) are the most commonly used comparators.

```
     A gets 3 2 reshape 65 77 4 19 22 11
65 77
 4 19
22 11

     Addrs := GRADE >= A
+---+---+
|0 1|0 0|
+---+---+
|2 0|1 1|
+---+---+
|2 1|1 0|
+---+---+

     Addrs choose A
77 65
22 19
11  4

     GRADE <= ("some "not "in "order)
3 1 0 2

     GRADE <= ['xyz','abc','mno','cat']
?invalid comparison in GRADE

     GRADE up ['xyz','abc','mno','cat']
1 3 2 0
```

The first three expressions illustrate that *GRADE* returns the addresses that re-order the items of table *A* in descending order. The next expression shows that the *GRADE* transform of <= can be applied to a list of phrases. The second last expression shows that *GRADE* <= cannot be applied to a list of strings. This is because <= is used itemwise on the characters of the strings and hence the comparator yields a bitstring rather than an atomic boolean result. The last expression shows that the comparator *up* can be used with strings.

Equations

```
SORT f A = GRADE f A choose A
f a comparator ==> shape GRADE f A = shape A
```

gradeup

Class: sorting operation

Usage: `gradeup A`

See Also: sortup, grade, up

The operation

```
     gradeup 3 7 5 4 9 8 2 1 6 10
7 6 0 3 2 8 1 5 4 9

     gradeup ("some "words "not "in "order)
3 2 4 0 1
```

Definition

```
     gradeup IS GRADE up
```

Equation

```
gradeup A choose A = sortup A
```

grid

Class: array generation operation

Usage: `grid A`

See Also: tell, address, shape

The operation *grid* returns the array of addresses of the array *A*. The result has the same shape as *A*. Each item of the result is the address of the corresponding item in *A*. The grid of a list is a list of integers. The grid of a table is a table of pairs of integers.

```
     grid 3 6 4 7 4
0 1 2 3 4

     A :=  2 3 reshape count 6; A (grid A)

+-----+-------------+
|1 2 3|+---+---+---+|
|4 5 6||0 0|0 1|0 2||
|     |+---+---+---+|
|     ||1 0|1 1|1 2||
|     |+---+---+---+|
+-----+-------------+
```

Definition

```
     grid IS OPERATION A { tell gage shape A }
```

Equations

```
   grid A choose A = A
   shape grid A = shape A
```

gt

Class: comparison operation

Property: binary pervasive

Usage: `A gt B   A > B   gt A B   > A B`

See Also: gte, lt, match, mate, max, sort

The operation *gt* compares two atoms *A* and *B* with the *greater than* relation, returning *true* if *A* is greater than *B* and *false* otherwise. The symbol > is a synonym for *gt*.

The atoms in Nial are organized as a lattice using <= for the ordering. The numeric atoms are comparable across types but numeric and literal atoms are incomparable. The literal types are not comparable across types. A comparison between incomparable objects results in *false*.

```
     R := 1 2 2.5 `a "abc ??error
1 2 2.5 a abc ?error

     R OUTER > R
oooooo
looooo
lloooo
oooooo
oooooo
oooooo
     'apple' > 'above'
olloo

     "apple > "above
1
```

The use of *OUTER* > shows the comparisons between various atom types. The last two examples show the difference between comparing two strings, where the operation is distributed by its pervasive property; and comparing the corresponding phrases.

Definition

```
gt IS OPERATION A B { (A gte B) and not (A mate B) }
```

gte

Class: comparison operation

Property: binary pervasive

Usage: `A gte B   A >= B   gte A B   >= A B`

See Also: gt, lte, match, sort

The operation *gte* compares two atoms *A* and *B* with the *greater than or equal* relation, returning *true* if *A* is greater than or equal to *B* and *false* otherwise. The symbol >= is a synonym for *gte*.

The atoms in Nial are organized as a lattice using <= for the ordering. The numeric atoms are comparable across types but numeric and literal atoms are incomparable. The literal types are not comparable across types. A comparison between incomparable objects results in *false*.

```
     R := 1 2 2.5 `a "abc ??error
1 2 2.5 a abc ?error

     R OUTER >= R
looooo
lloooo
lllooo
oooloo
oooolo
oooool

     'apple' >= 'above'
lllol

     "apple >= "above
1
```

The use of *OUTER* >= shows the comparisons between various atom types. The last two examples show the difference between comparing two strings, where the operation is distributed by its pervasive property; and comparing the corresponding phrases.

Definition

```
gte IS OPERATION A B { B lte A }
```

help

Class: system expression

Usage: `Help`

See Also: topic, symbols

The Q'Nial Help Facility is integrated with the Windows Help mechanism in the Q'Nial for Windows version. If the cursor is over a word for which there is a help entry, then pressing F1 will bring up the help screen for that topic.

The expression *Help* starts up the Q'Nial Help Facility in console versions. It reads text from the help file stored in directory *nialhelp* and displays the information in an edit window. The explanation on using the help facility is given in the initial screen.

hitch

Class: construction operation

Property: binary

Usage: `A hitch B  hitch A B`

See Also: append, link, list, solitary

The operation *hitch* attaches *A* to the front of the list of items of *B*. It returns a list of length one greater than the tally of *B*.

```
     (2 3 4) hitch (5 6 7)
+-----+-+-+-+
|2 3 4|5|6|7|
+-----+-+-+-+

     7 hitch 3
7 3

     hitch 'Wow' ''
+---+
|Wow|
+---+
```

The first example shows that the list 2 3 4 becomes an item on the front of the list 5 6 7. In the next example, the right argument of *hitch* is treated as a list. The last example shows that if the right argument is an empty list, the result is the left argument as a solitary.

Definition

```
hitch IS OPERATION A B { solitary A link B }
```

Equations

```
A hitch B = A hitch (list B)
A hitch Null = solitary A
list (A hitch B) = A hitch B
not empty A ==> first A hitch rest A = list A
shape A reshape (first A hitch rest A) = A
```

host

Class: system operation

Usage: `host S`

See Also: edit, refresh

The operation *host* executes *S* as a host command language instruction. The argument *S* is a string or a phrase. If the action carried out by the host system produces output, the output is displayed on the screen. In window mode, Q'Nial is unable to capture this output and hence it may scramble the screen output. The screen is restored by executing *Refresh*.

The result of *host* is the fault *?noexpr* if the command has returned normally; or a fault generated from a system dependent error message supplied by the host operating system. At the top level loop, a line beginning with *!*, the exclamation mark, is interpreted as a host command.

if-expr

Class: control structure

Usage: `IF C1 THEN Es1 ELSEIF C2 THEN Es2 ... ELSEIF Cn THEN Esn ELSE Esx ENDIF`

See Also: case-expr, fork

The *IF-expr* construct is a notation for executing one of a number of possible expression sequences *Es1*, *Es2*, ... *Esn*. The sequence selected depends on the result of the conditional expressions *C1*, *C2*, ... *Cn*. In the general case, whichever condition is first found to return *true* specifies the expression sequence to be performed. If all the conditional expressions return *false*, expression sequence *Esx* is selected. The *ELSEIF* and *ELSE* clauses are optional.

In the following example, the result is one of phrase *Adult*, *Minor* or *Juvenile*, depending on the value of *Age*.

```
     Age := 17;
     IF Age > 18 THEN
        "Adult
     ELSEIF Age < 16 THEN
        "Minor
     ELSE
        "Juvenile
     ENDIF
Juvenile
```

in

Class: search operation

Property: binary

Usage: `A in B  in A B`

See Also: allin, notin, seek, find, equal, sortup

The operation *in* returns *true* if *A* is an item of *B* and returns *false* if it is not.

```
     3 in 56 34 23 3 57 3
1
     1 1 in tell 3 4
1
     `a in 'hello world'
0
     `a in "apple
0
     `a in string "apple
1
```

The fourth and fifth examples show that a letter is not an item of a phrase but is an item of the corresponding string.

Definition

```
     in IS OPERATION A B { or (A EACHRIGHT equal B }
```

Equations

```
   A in B = A in (list B)
   A in Null = False
   A in solitary A = True
   A in (A hitch B) = True
   A in (B append A) = True
   A in (A pair B) = True
```

Pragmatics

The operation *in* uses a linear search on the items of *B* if the array has not been sorted, or uses a binary search algorithm if it has. The latter fact suggests that an array that is searched frequently should be kept in lexicographical order by applying *sortup* to it when it is created.

indexing

Class: concept

See Also: address, pick, choose, reach

The term **indexing** is used to describe notations that can be used to *select from* or *insert into* a variable.

There are four indexing methods in Nial: *at*, *at all*, *at path* and *slice* represented by @, #, @@ and | respectively. The different indexing methods return different subsets of the array. The following is a summary of the indexing methods:

Method	Name	Description
@	at	indexes one item
#	at all	indexes several items
@@	at path	indexes a part at depth
|	slice	indexes cross-section of items

```
Cities := "London "Washington "Ottawa "Moscow "Paris ;
     Cities@0
London


     Cities#[2,3]
Ottawa Moscow


     Alpha := 5 5 reshape 'ABCDEFGHIJKLMNOPQRSTUVWXY'
ABCDE
FGHIJ
KLMNO
PQRST
UVWXY


     Alpha@[0,2]
C

     Alpha|[1,]
FGHIJ
     Alpha|[,1]
BGLQV

     Alpha|[,[1,3]]
BD
GI
LN
QS
VX
```

The following example shows the slightly different structure which occurs when a comma is either present or missing before the last item. The library operation *findpaths* is used to indicate the path to the integer 10 in each case.

```
    set "diagram ; Nest1 := [1, 2, [3, 4, 5, [6, 7], 8, 9], 10]
+-+-+-----------------+--+
|1|2|+-+-+-+-----+-+-+|10|
| | ||3|4|5|+-+-+|8|9||  |
| | || | | ||6|7|| | ||  |
| | || | | |+-+-+| | ||  |
| | |+-+-+-+-----+-+-+|  |
+-+-+-----------------+--+

     set "diagram ; Nest2 := [1, 2, [3, 4, 5, [6, 7], 8, 9] 10]
+-+-+----------------------+
|1|2|+-----------------+--+|
| | ||+-+-+-+-----+-+-+|10||
| | |||3|4|5|+-+-+|8|9||  ||
| | ||| | | ||6|7|| | ||  ||
| | ||| | | |+-+-+| | ||  ||
| | ||+-+-+-+-----+-+-+|  ||
| | |+-----------------+--+|
+-+-+----------------------+

     findpaths 10 Nest1
+---+
|+-+|
||3||
|+-+|
+---+

     findpaths 10 Nest2
+-----+
|+-+-+|
||2|1||
|+-+-+|
```

```
+------+
      (Nest2@@[2,0,3,0])
6

      (Nest2@@[2,0,4])
8
```

Address Validity

The index used in selecting a part of an array must be an expression that evaluates to a valid address. An invalid index returns a fault as follows:

Indexing Method	Fault
A@I	?address
A#I	?addresses
A@@P	?path
A|I	?slice

The operation *pick* works the same as the *at* method of indexing. If the index is invalid, *pick* returns the fault *?address*. Similarly, *choose* works the same as *at all* indexing.

```
      3 pick Cities
Moscow

      2 3 choose Cities
Ottawa Moscow

      10 pick Cities
?address

      4 5 6 choose Cities
Paris ?address ?address
```

infix notation

Class: syntax

See Also: prefix notation

In Nial an operation-expression may be placed between two array-expressions. This is called an **infix** use of the operation-expression.

```
      7 + 5
12

      2 3 reshape 1 2 3 4 5 6
1 2 3
4 5 6
```

In using the infix notation, one must understand that if a sequence of operations are placed between two array arguments, all but the first operation are applied to the second argument.

```
     2 + reverse tell 3
4 3 2

     2 (+ reverse tell) 3
3 6

     2 (+ reverse) tell 3
2 3 4
```

initial subdirectory

Class: concept

See Also: clear workspace

The *initial* subdirectory is within the directory identified as the Nialroot directory. It holds a number of files that may be used for initializing Q'Nial including the *defs.ndf* file used for creating the clearws workspace.

When a Q'Nial session is initiated, if none of the -defs, Filename or Wsname options is used as a parameter to the nial command, the *clearws.nws* workspace is loaded. If the *clearws.nws* file does not exist in the current directory, it is sought in the Nialroot directory. If it is not present in either place, it is created automatically by the initialization process and kept internal to Q'Nial.

A user can customize their version of Q'Nial by setting up an *initial* directory and modifying the *defs.ndf* file to include definitions they wish to be present in every Q'Nial session.

inner

Class: applicative transformer

Usage: `A INNER [f,g] B   INNER [f,g] A B`

See Also: outer, innerproduct, solve, inverse

The transformer *INNER* generalizes the inner product operation of linear algebra. For a pair of lists *A* and *B*, the result is the application of the reductive operation *f* to the result of applying the binary pervasive operation *g* to *A* and *B*. It is assumed that *f* is one of the seven reductive operations of Nial: *sum*, *product*, *or*, *and*, *max*, *min*, or *link* and that *g* distributes pairwise. For tables, *OUTER g* is applied to the rows of *A* and the columns of *B* and *f* is applied to each item of the outer product.

Thus, *INNER [+,*]* is equivalent to matrix multiplication in linear algebra and *INNER [or,and]* is boolean matrix product. For higher dimensional arrays, the lists are formed by "pushing down" the last axis of *A* and the first axis of *B*.

```
     2 3 4 INNER [+,*] 4 5 6
47
     loll INNER [or,and] olol
l
     A := 2 4 reshape count 8;
     B := 4 3 reshape tell 8;
     A B
+-------+-----+
|1 2 3 4|0 1 2|
|5 6 7 8|3 4 5|
|       |6 7 0|
|       |1 2 3|
+-------+-----+
     A INNER [sum,times] B
28 38 24
68 94 64
```

Definition

```
     INNER IS TRANSFORMER f g (OPERATION A B) {
        rows A OUTER (f g) (0 split B) }
```

Equation

```
   shape (A INNER [f,g] B) = (front shape A) link (rest shape B)
```

innerproduct

Class: linear algebra operation

Usage: `A innerproduct B` `A ip B` `innerproduct A B` `ip A B`

See Also: inner, outer, solve, inverse

The operation *innerproduct* computes the mathematical inner product of real vectors and matrices using special code for efficiency. For real matrices, it produces the same result as *A INNER [+,*] B*, but computes the result more rapidly for large arguments. It coerces boolean and integer arrays to reals. The name *ip* is provided as an abbreviation.

```
     A := 2 4 reshape count 8;
     B := 4 3 reshape tell 8;
     A B
+-------+-----+
|1 2 3 4|0 1 2|
|5 6 7 8|3 4 5|
|       |6 7 0|
|       |1 2 3|
+-------+-----+

     A innerproduct B
28. 38. 24.
68. 94. 64.
```

The number of columns of *A* must match the number of rows in *B*.

```
     B ip A
?conform in ip
```

interrupt

Class: concept

See Also: fault triggering, setinterrupts, toplevel

An **interrupt** is an event that causes the operating system to suspend its operation and address a requirement of higher priority. Typically, interrupts occur to handle input/output. However, an interrupt also occurs when a fault is detected.

In the default mode of operation of Q'Nial, most fault values are not created. Rather, an interrupt is triggered. A description of the fault triggering mechanism is given under *fault triggering*.

The user can interrupt execution by pressing <*Ctrl-c*> <*Return*> at the keyboard in console versions, or clicking on the *STOP* button in the GUI version. This capability can be turned off using *setinterrrupts*. An interrupt can also be triggered under program control using the expression *Toplevel*.

inverse

Class: linear algebra operation

Usage: `inverse A  inverse A`

See Also: solve, inner, innerproduct, outer

The operation *inverse* computes the mathematical inverse of a square matrix *A* returning a square matrix of the same shape. If *A* is **singular** within numerical limits, the result is the fault *?singular*. The name *inv* is provided as an abbreviation.

```
     seed 0.5;
     A := ceiling (100. * (3 3 reshape random 9))
50 44 13
78 74 90
98 67 17

     inverse A
-0.0736442  0.00189821  0.0462669
  0.115652 -0.00654341  -0.053798
-0.0312664   0.0148461 0.00413593

     A innerproduct inverse A
         1. -6.93889e-18 -7.11237e-17
 9.4369e-16           1. -1.97758e-16
3.60822e-16 -6.93889e-18           1.
```

The final computation shows that the result is not always an exact inverse due to roundoff errors introduced by using floating point arithmetic.

Equations

```
   A a square matrix ==> A innerproduct (inverse A) innerproduct A = A   (within roundoff
error)
   inverse A innerproduct B = A solve B   (within roundoff error)
```

isboolean

Class: measurement operation

Property: predicate

Usage: `isboolean A`

See Also: type, isinteger, isreal

The operation *isboolean* tests whether or not *A* is a boolean atom. It returns *true* if *A* is a boolean, *false* otherwise.

```
     isboolean false
l
     isboolean llollool
o
     isboolean 7
o
```

Definition

isboolean IS OPERATION A { type A = o }

ischar

Class: measurement operation

Property: predicate

Usage: `ischar A`

See Also: type, isfault, isphrase, isstring

The operation *ischar* tests whether or not atom *A* is a character atom. It returns *true* if *A* is a character, *false* otherwise.

```
     EACH ischar (`a) ('a') 7
loo
```

In the example *'a'* is not an atom; it is a solitary holding *`a*.

Definition

```
     ischar IS OPERATION A { type A = ` }
```

isfault

Class: measurement operation

Property: predicate

Usage: `isfault A`

See Also: type, isphrase, ischar, isstring

The operation *isfault* tests whether or not *A* is a fault. It returns *true* if *A* is a fault, *false* otherwise.

```
     EACH isfault (??error) ("?error)
lo
```

Definition

```
     isfault IS OPERATION A { type A = ?? }
```

isinteger

Class: measurement operation

Property: predicate

Usage: `isinteger A`

See Also: type, isboolean, isreal

The operation *isinteger* tests whether or not *A* is an integer. It returns *true* if *A* is an integer, *false* otherwise.

```
     EACH isinteger (7)(2 3 4)(`3)(3.0)
looo

     isinteger (`a find 'whale')
l
```

The last example returns *true* because the result of *`a find 'whale'* is an integer, since the second argument of *find* is a list.

Definition

```
     isinteger IS OPERATION A { type A = 0 }
```

isphrase

Class: measurement operation

Property: predicate

Usage: `isphrase A`

See Also: type, isstring, isfault, ischar

The operation *isphrase* tests whether or not *A* is a phrase. It returns *true* if *A* is a phrase, *false* otherwise.

```
     isphrase "Mike
l
     isphrase 'abc'
o
     isphrase  `a
o
```

Definition

```
     isphrase IS OPERATION A { type A = "" }
```

isreal

Class: measurement operation

Property: predicate

Usage: `isreal A`

See Also: type, isboolean, isinteger

The operation *isreal* tests whether or not *A* is a real number. It returns *true* if *A* is a real number, *false* otherwise.

```
     isreal 3.5
l

     isreal 'abc'
o
```

Definition

```
     isreal IS OPERATION A { type A = 0.  }
```

isstring

Class: measurement operation

Property: predicate

Usage: `isstring A`

See Also: ischar, string

The operation *isstring* tests whether or not the array *A* is a string. It returns *true* if *A* is a string, *false* otherwise.

```
     isstring 'A string'
l
     isstring `A
o
     isstring `2 `A `?
l
     isstring ''
l
```

Definition

```
     isstring IS OP A { valence A = 1 and EACH ischar A }
```

Equations

```
   atomic A ==> isstring string A = True
```

item

Class: concept

See Also: array, address

An array *A* is said to be an **item** of array *B* if *B* holds *A* at one or more locations. The term is a relative one; we cannot speak of item except in reference to the array that holds it. The items of an array A are the objects at the locations at the top level.

The number of items in an array is called the tally of the array. Because an array is rectangular, the tally is the product of the shape.

The following names are give to common array structures:

# of items	# of axes	Name
0	1	empty list
1	0	single
1	1	solitary
1	2	1 by 1 table
2	1	pair
3	1	triple
4	1	quadruple

The arrays of Nial are a recursive data type. That is, the items of an array are also arrays. Since an array has arrays as items, it may contain data at lower levels than the top one. A **path** is a list of addresses that describes a data object at some depth within the array.

An array is said to be **simple** if all its items are atomic.

A **part** of an array is a data object that is contained at some level within the array. The atomic parts of an array are called the **leaves** of the array. The simple parts are called **twigs**. The term **level** is used informally to describe the relative position of a part within the nesting structure of an array. An item is at the first or top level, an item of an item is at the second level, etc.

An **empty** array is one that has no items.

iterate

Class: control structure transformer

Usage: `ITERATE f A`

See Also: each, for-loop, fork

The transformer *ITERATE* is used to apply the operation *f* sequentially to the items of *A* in row major order. (Row major order means across the rows moving left to right, starting at the top row and then going down the rows. An example is given under list.) The result is the result of applying *f* to the final item. If *A* has no items, the result is the fault *?noexpr*.

```
     ITERATE write "Hello "out "there.
Hello
out
there.

     A IS EXTERNAL VARIABLE;
     accum IS OPERATION B { NONLOCAL A; A := A + B };
     A := 20;
     ITERATE accum 3 4 5 ; A
32
```

In the second example, the operation *accum* increments variable *A* by the argument value. The example shows *ITERATE* being used to apply *accum* to the list 3 4 5, which results in *A* having the values 23, 27 and 32.

With *ITERATE*, an operation that has a side effect can be applied to an array of arguments sequentially in a specific sequence. On the other hand, the order of application is undefined for *EACH*.

A second major difference between *EACH* and *ITERATE* is that *ITERATE* returns only the result of the last application whereas *EACH* returns the array of all the results.

Definition

```
     ITERATE IS TR f OPERATION A { FOR X WITH A DO f A ENDFOR }
```

juxtaposition

Class: syntax

The syntax rules for simple-expressions show three uses of the side-by-side or juxtapositional notation of Nial: strand formation, prefix operation application and infix operation application. There are no syntactic restrictions as to whether or not a particular operation may be applied in infix or prefix form. A fault is returned at run time if an operation is used inappropriately.

Summary of Juxtapositional Syntax

The following table illustrates the uses of juxtaposition in Nial, where *A* and *B* are array-expressions, *f* and *g* are operation-expressions, and *T* is a transformer:

Form	Name	Object

A B	strand	array
A f	currying	operation
f A	prefix use	array
f g	composition	operation
T f	transform	operation
A f B	infix use	array
T f A	transform use	array

laminate

Class: construction operation

Usage: `I laminate A   laminate I A`

See Also: link, catenate

The operation *laminate* merges the items of *A* adding a new axis before axis *I* of the items. The items of *A* must be of the same shape.

The following example creates a three dimensional array from two tables, placing the new axes at the front:

```
     0 laminate (tell 2 3) (count 2 3)
+---+---+---+  +---+---+---+
|0 0|0 1|0 2|  |1 1|1 2|1 3|
+---+---+---+  +---+---+---+
|1 0|1 1|1 2|  |2 1|2 2|2 3|
+---+---+---+  +---+---+---+
```

Definition

```
     laminate IS OPERATION I A {
        IF equal EACH shape A THEN
           Axesofitems := axes first A;
           link (I take Axesofitems) (I drop Axesofitems + 1) blend A
        ELSE
           fault '?conform error in laminate'
        ENDIF }
```

last

Class: selection operation

Usage: `last A`

See Also: first, pick, take

The operation *last* returns the last of the items of *A*. If *A* is empty, it returns the fault *?address*. The operation *last* is a special case of *pick* (because *first* is defined in terms of *pick*) and its behaviour is determined by that of *pick*. Every nonempty array has a last item.

```
     last 4 5 6
6

     last tell 3 4
2 3
```

Definition

```
     last IS OPERATION A { first reverse A }
```

Equations

```
   last single A = A
   last solitary A = A
   last list A = last A
```

latent

Class: user defined expression

Usage: `Latent`

See Also: recover, checkpoint, load, save

The expression *Latent* is used to name an expression to be executed without user intervention when the workspace is loaded. *Latent* is used in closed applications so that an application can be started when the workspace is loaded. *Latent* can establish any default or initial conditions desired.

```
     Latent IS {
        settrigger o;
        set "log;
        StartApp;
        Bye; }
```

In the example, *Latent* is defined to turn of triggering of faults, to turn on session logging, to start the application and then to terminate the session.

leaf

Class: distributive transformer

Usage: `LEAF f A`

See Also: twig, each, content

The transformer *LEAF* modifies an operation *f* into an operation that applies f to every atom of the argument *A*. The result of applying *LEAF f* to *A* has the same shape as *A*. If *f* maps atoms to atoms, the result has the same structure as *A*.

```
     A := tell 2 3
+---+---+---+
|0 0|0 1|0 2|
+---+---+---+
|1 0|1 1|1 2|
+---+---+---+

     LEAF tally A
+---+---+---+
|1 1|1 1|1 1|
+---+---+---+
|1 1|1 1|1 1|
+---+---+---+

     B := (2 3) (1 2)
+---+---+
|2 3|1 2|
+---+---+

     LEAF tell B
+-----------+-------+
|+---+-----+|+-+---+|
||0 1|0 1 2|||0|0 1||
|+---+-----+|+-+---+|
+-----------+-------+
```

The first example shows that the result of applying a *LEAF* transform to a table is a table of the same shape. The atoms have been mapped to 1 since the tally of an atom is 1. The second example shows that applying a *LEAF* transform to a list gives a list of the same length. However, the structure of the result is not preserved in this example because *tell* maps an integer to a list of integers.

Definition

```
     LEAF IS TRANSFORMER f OPERATION A {
        IF atomic A THEN
           f A
        ELSE
           EACH (LEAF f) A
        ENDIF }
```

Equations

```
   shape LEAF f A = shape A
   f unary pervasive ==> LEAF f A = f A
   (LEAF f) (LEAF g) A = LEAF (f g) A
   LEAF f list A = list LEAF f A
```

level

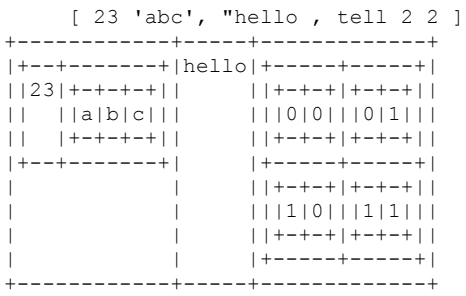
Class: concept

See Also: item

The term **level** is used informally to describe the relative position of a part within the nesting structure of an array. An item is at the first or top level, an item of an item is at the second level, etc.

An atom is viewed in two ways. As an indivisible data object it is viewed as having no levels and cannot be broken into subarrays. As an array data structure it is viewed as a single holding itself and therefore has an infinity of levels. This view is necessary for atomic arrays to fit the theory of nested array mathematics.

The number of levels to reach an atom along each path need not be the same. For example, in the following array, the phrase "hello is at the first level, the integer 23 is at the second level and the character `b is at the third level.



Some of the operations of Nial that operate on simple arrays are extended to arbitrarily nested arrays by being applied to the atoms at the deepest level. These are called *pervasive* operations.

libpath

Class: system variable

Usage: `Libpath := Paths`

See Also: library

The variable *Libpath* is used by the operation *library* as a list of paths to directories that are to be checked for the definition file named in the *library* argument. The directories defined by *Libpath* are searched before the system dependent library directories.

```
Libpath := ['mydefs','mylib\newdefs'];
```

library

Class: system operation

Usage: `library Nm   library Nm Sw`

See Also: loaddefs, libpath, host, standard definitions

The operation *library* loads a definition from the Q'Nial library of definition files. *Nm* is the name of a library file as a phrase or string. The name is augmented with the path information for the library and loaded into the workspace using *loaddefs*.

Sw, an optional argument, is either 0 (the default) or 1. If *Sw* is 0, there is no display of the file as it is loaded. If *Sw* is 1, the file is displayed as the file is read.

Some of the programs in the library are grouped in definition files by function. When such a file is specified, all the definitions in the file are loaded. If it is desired to load only one operation from a definition file which contains several operations, it is necessary to edit the library file in order to isolate the desired operation in one file.

```
library "labeltable
```

Library searches for files using paths that are provided in its definition. Before searching for system defined directories, it searches in the directories named in the global variable *Libpath*, which is empty by default.

To modify Libpath, either edit the standard definitions file *defs.ndf*, or assign it a value dynamically.

like

Class: set-like operation

Properties: binary, predicate

Usage: `A like B  like A B`

See Also: allin, diverse, in, notin

The operation *like* compares two arrays *A* and *B* and returns *true* if all the items of *A* are items of *B* and vice versa. Otherwise it returns *false*. The operation *like* corresponds to set equality at the first level of nesting.

```
     2 3 like 3 2 2 3 2
1
     2 3 like 3 2 4
0
     'hello world' like 'whole door'
1
```

Definition

```
     like IS OPERATION A B { (A allin B) and (B allin A)}
```

link

Class: construction operation

Property: reductive

Usage: `link A  A link B`

See Also: catenate, laminate, hitch, list, append, content, cull

The operation *link* returns the list of the items of the items of its argument. If it is applied to a pair of arrays *A* and *B*, the result is the items of *A* followed by the items of *B*.

If link is applied to an arbitrary array *A*, the items of the first item of *A* are followed by the items of the second item of *A*, etc.

If *A* is empty, the result is *Null*. If *A* is not empty but there are no items in all the items of *A*, the result is also *Null*.

```
     2 3 5 4 7 2 link 2 4 6 8
2 3 5 4 7 2 2 4 6 8

     link 'hen' 'hello' 'eh'
henhelloeh
```

The first example shows that the link of two lists of length 6 and 4 is a list of length 10. The second example illustrates the linking of three strings.

Link is similar to a set-union operation, although it does not remove duplicates in the representation. The composition of *cull* with *link* may be more appropriate as a union operation.

Equations

```
link A = link list A
link A = list link A
link A = EACH list link A
tally link A = sum EACH tally A
link solitary A = list A
link Null = Null
link EACH link A = link link A
cart link A = EACH link cart EACH cart A
EACH f link A = link EACH EACH f A
simple A ==> link A = list A
and EACH simple A ==> content A = link A
```

list

Class: reshaping operation

Usage: `list A`

See Also: reshape, link

The operation *list* returns the list of the items of *A* in row major order.

```
     A := 2 3 4 reshape count 24
1  2  3  4   13 14 15 16
5  6  7  8   17 18 19 20
9 10 11 12   21 22 23 24

     list A
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24

     set "diagram; list "abc
+---+
|abc|
+---+
```

The first example shows row major order for a table. The second example shows that the list of an atom is the solitary holding the atom.

Definition

```
     list IS OPERATION A { tally A reshape A }
```

Equations

```
shape list A = tally A
list list A = list A
list single A = solitary A
EACH f list A = list EACH f A
empty A ==> list A = Null
```

ln

Class: scientific operation

Property: unary pervasive

Usage: `ln A`

See Also: exp, log

The operation *ln* implements the natural logarithm function of mathematics. It produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	natural logarithm of the corresponding real
integer	natural logarithm of the corresponding real
real	real natural logarithm if A > 0.0; ?ln if A <= 0.0
character	fault ?ln
phrase	fault ?ln
fault	argument A

```
    ln  1  2  3.5   `a  "abc  ??error
0. 0.693147 1.25276 ?ln ?ln ?error
```

Equation

```
A > 0 ==> exp ln A = A   (within roundoff error)
```

load

Class: system operation

Usage: `load Wsname`

See Also: save, loaddefs, library, restart, latent, continue

The operation *load* is used to retrieve the saved workspace named by the phrase or string *Wsname*. The convention in Q'Nial is to save workspaces with a file name extension of the form *.nws* to make it easy to identify workspace files in the file system. When saving or loading a workspace, the extension may be omitted.

The effect of loading a workspace is to replace the current workspace with the saved one. To keep the contents of the current workspace, *save* should be used prior to doing a *load*.

The operation *load* may be used within a defined expression or operation. However, in such a use, it interrupts the execution of the operation or expression and does the load as though the operation were entered at top level.

If a workspace contains an expression with the name *Latent*, *load* executes *Latent* when the load is complete. This mechanism can be used to have the load of a workspace automatically begin an application or to chain execution of workspaces.

There is no mechanism to obtain individual objects saved in a workspace.

```
load "mywork
```

loaddefs

Class: system operation

Usage: `loaddefs Deffilename` `loaddefs Deffilename Mode`

See Also: library, defedit, edit

The operation *loaddefs* is used to load Nial actions (definitions or expressions) from the file named by the phrase or string *Deffilename*. The convention in Q'Nial is to name Nial definition files with a file name extension of the form *.ndf* to make it easy to identify such files in the file system.

When loading a file with *loaddefs*, the file extension may be omitted. The operation may have an optional second argument to specify the mode of loading. If mode is 0 or if it is omitted, the actions are not displayed as they are processed. If it is 1, the actions are displayed.

The file of actions consists of groups of lines that are treated as a unit. These lines are separated by a blank line. If the first character in the first line of the group is #, the group is a *remark* and is not processed.

Several lines, up to a blank line, are treated as one long line. No blank spaces are inserted between the end of one line and the beginning of another. Thus, care should be taken to ensure that identifiers are not joined together.

If a syntax error is detected when a definition is being loaded, the definition is not installed. The absence of the erroneous definition may cause subsequent definitions in the file to fail also. If any errors are detected, the number of errors is displayed when *loaddefs* ends. Only the first syntax error found in each action is reported.

The backslash character, \, which is used to continue a line of a definition in immediate mode, must not be inserted at the end of lines in a definition file.

The tab characters, if they exist in a definition file, are treated as though they are space characters. Since some editor programs insert tabs automatically, the display of the file on input may be different from the display using the editor.

A definition file may contain uses of *loaddefs*. For a large application with many definition files, it is a good idea to have one file which can construct the application by using *loaddefs* to bring in the other files in an appropriate order. *Loaddefs* can also be used to execute a script of Nial expressions. This feature is convenient when simulating a Q'Nial session and capturing its output in a log file.

local environment

Class: concept

See Also: global environment, scope of a variable

A **local environment** is a collection of associations that are known within a limited section of program text. These limited sections are formed by blocks, operation-forms and transformer-forms as discussed in the relevant sections below. A name that has a local association in one of these forms is said to have local scope.

Program fragments in which local variables are being assigned can be nested, so that one local scope surrounds another. A local association is not visible outside the construct in which it is defined; and a name with local scope can hide associations that the name has in surrounding scopes.

At any point in a program fragment, there is a current environment consisting of all names whose associations are visible. It includes the names having local scope in the program fragment being executed, names that are visible in the surrounding scopes and names that have global scope.

In program text, the scope of all names is determined by the static structure of the program text. The one exception is text that has the operation execute applied to it under program control.

In a local environment, a variable identifier can be chosen the same as a predefined or user-defined global definition name. Such a choice makes the global use of the name unavailable in the local context.

In any context, an identifier can name only one of: a variable, an array- expression, an operation-expression, or a transformer-expression. During one session, the role of a name, i.e. the class of syntactic object it names, cannot be changed.

If a block is used as a primary-expression, the local environment created by a block is determined by the block itself. If it is the body of an operation-form, the local environment includes the formal parameter names of the operation-form as variables.

A block delimits a local environment. It allows new uses of names which do not interfere with uses of those names outside the block. For example, within a block, a predefined operation name can be redefined and used for a different purpose. Only the reserved words of Q'Nial cannot be reused in this fashion. Definitions that appear within the block have local scope. That is, the definitions can be referenced only in the body of the block. Variables assigned within the block may or may not have local scope, depending on the appearance of a local and/or a nonlocal declaration. If there is no declaration, all assigned variables have local scope. Declaring some variables as local does not change the effect on undeclared variables that are used on the left of assignment. They are automatically localized.

If a nonlocal declaration is used, an assigned name that is on the nonlocal list is sought in surrounding scopes. If the name is not found, a variable is created in the global environment.

During the parse of the assign-expression appearing in a block, each name on the variable list is sought in the local environment. If the name exists in the local environment, the assignment affects the local association. If a name does not exist in the local environment and no reference has been made to a nonlocal variable with the same name, a local variable is created in the block. An assign-expression parsed in the global environment creates a global variable if a variable with that name does not already exist.

An operation-form defines a local environment. The formal parameter names are names of local variables. If the body of the operation form is a block, the local environment of the block is extended to include the formal parameters. When the operation is applied, the formal parameter names are assigned from the value of the actual argument. If there is only one formal parameter, the actual argument is assigned to it as a whole; otherwise, the items of the actual argument are assigned to the formal parameters in corresponding order. If there is a length mismatch between the list of formal parameter names and the values of the actual argument, the fault *?op_parameter* is returned.

The value of the application of the operation is the value of the body of the operation-form, which is evaluated with the local variables in the parameter list assigned as described above. In determining the association for a name that appears in the body of an operation form, Q'Nial looks for the name in the local environment. If the name is not found locally, the name is sought in surrounding environments until it is found or until the global environment is searched. If it is not found, a fault *?unknown identifier:* is given when the operation-form is analyzed (parsed).

Operation-forms are most frequently used in definitions where they are given an associated name. However, an operation-form can appear directly in an expression provided it is enclosed in parentheses. In this usage, it can be an argument to a transformer name or can be applied to an array argument.

The operation *execute* can be used within the execution of a block to make an assignment to variables or to invoke the definition mechanism. If *execute* is used to make a new definition or to create a new variable, the resulting variable or definition is placed in the global environment. However, if the block has local variables or local definitions, execute can be used to change a local version dynamically. A similar situation occurs with dynamic alteration of variables using *assign*.

log

Class: scientific operation

Property: unary pervasive

Usage: `log A`

See Also: power, ln

The operation *log* implements the base 10 logarithm function of mathematics. It produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	base 10 logarithm of the corresponding real
integer	base 10 logarithm of the corresponding real
real	real base 10 logarithm if A > 0.0; ?log if A <= 0.0
character	fault ?log
phrase	fault ?log

fault argument A

```
   log  1  2  3.5  `a  "abc  ??error
0. 0.30103 0.544068 ?log ?log ?error
```

Equation

```
   A > 0 ==> 10. power log A = A (within roundoff error)
```

log file

Class: feature

See Also: setlogname

Q'Nial provides a facility to record the actions in a session in a text file. The default name for the log file is *auto.nlg*. Logging is initiated by:

```
     set "log
nolog
```

Logging is ended by:

```
     set "nolog
log
```

The log file name can be changed using:

```
     setlogname "newname
auto.nlg
```

A log file is opened and closed on each usage by the internal logging routine. As a result, the log file is always available if the session is terminated unexpectedly. If a file with the name of the log file exists when *set "log* is executed, the logging information is appended at the end.

lower

Class: nesting restructuring operation

Usage: `N lower A  lower N A`

See Also: raise, mix, blend, rank

The operation *lower* is used to partition an array *A* along its axes by indicating that the last *N* axes are to become axes of the items of the result. The remaining axes become the axes of the result. N must be an integer in the range from 0 to *valence A*. The result is an array of shape given by dropping the last *N* items of *shape A*. The items of the result have the shape given by taking the *last N* items of *shape A*. Thus, the *1 lower* of an array of shape 3 4 2 is a 3 by 4 table of pairs. The *2 lower* of the same array is a triple of 4 by 2 tables.

```
     A := 3 4 2 reshape 'ABCDEFGHIJKLMNOPQRSTUVWX'
AB  IJ  QR
CD  KL  ST
EF  MN  UV
GH  OP  WX
```

```
     1 lower A
+--+--+--+--+
|AB|CD|EF|GH|
+--+--+--+--+
|IJ|KL|MN|OP|
+--+--+--+--+
|QR|ST|UV|WX|
+--+--+--+--+

     2 lower A
+--+--+--+
|AB|IJ|QR|
|CD|KL|ST|
|EF|MN|UV|
|GH|OP|WX|
+--+--+--+
```

Definition

```
lower IS OPERATION A { valence A - N raise A }
```

Equations

```
N a nonnegative integer <= valence A and not empty A ==>
                       shape (N lower A) = opp N drop shape A
shape first (N drop A) = opp N take shape A ==> mix (N lower A) = A
```

lt

Class: comparison operation

Property: binary pervasive

Usage: `A lt B   A < B   lt A B   < A B`

See Also: lte, gt, gte, match, mate, max, sort

The operation *lt* compares two atoms *A* and *B* with the *less than* relation, returning *true* if *A* is less than *B* and *false* otherwise. Symbol < is a synonym for *lt*.

The atoms in Nial are organized as a lattice using <= for the ordering. The numeric atoms are comparable across types but literal atoms are not comparable with numeric atoms. The literal types are not comparable across types. A comparison between incomparable objects results in *false*.

```
     R := 1 2 2.5 `a "abc ??error
1 2 2.5 a abc ?error
```

```
     R OUTER < R
ollooo
oolooo
oooooo
oooooo
oooooo
oooooo

     'apple' < 'above'
ooolo

     "apple < "above
o
```

The use of *OUTER* < shows the comparisons between various atom types. The last two examples show the difference between comparing two strings, where the operation is distributed by its pervasive property; and comparing the corresponding phrases.

Definition

```
     lt IS OPERATION A B ( ( A lte B ) and not ( A mate B ) )
```

lte

Class: comparison operation

Property: binary pervasive

Usage: `A lte B   A <= B   lte A B   <= A B`

See Also: lt, gt, gte, match, mate, max, sort

The operation *lte* compares two atoms *A* and *B* with the *less than or equal* relation, returning *true* if *A* is less than *B* and *false* otherwise. The symbol <= is a synonym for *lte*.

The atoms in Nial are organized as a lattice using <= for the ordering. The numeric atoms are comparable across types but numeric and literal atoms are incomparable. The literal types are not comparable across types. A comparison between incomparable objects results in *false*.

```
     R := 1 2 2.5 `a "abc ??error
1 2 2.5 a abc ?error

     R OUTER <= R
lllooo
ollooo
oolooo
oooloo
oooolo
oooool

     'apple' <= 'above'
loolo

     "apple < "above
o
```

The use of OUTER <= shows the comparisons between various atom types. The last two examples show the difference between comparing two strings, where the operation is distributed by its pervasive property; and comparing the corresponding phrases.

match

Class: comparison operation

Property: binary pervasive

Usage: `A match B   match A B`

See Also: mate, gt, gte, lt, lte

The operation *match* compares two atoms *A* and *B* for exact equality, returning *true* if *A* is equal to *B* and *false* otherwise. *Match* cannot be used to determine the equality of atoms of different numeric or literal type, as when comparing the real number 3.0 with the integer 3.

```
     3 match 3
l

     3.0 match 3
o

     "o match `o
o

     1.0 match l 1 1.0
ool

     `  match 'a list of letters'
olooooloolooooooo

     1 match tell 2 3
+--+--+--+
|oo|ol|oo|
+--+--+--+
|lo|ll|lo|
+--+--+--+
```

The second last example shows that the binary pervasive extension of *match* allows a lists of atoms to be compared for a value, returning a bitstring of results. The last example shows where in the structure of tell 2 3 a 1 is held.

mate

Class: comparison operation

Property: binary pervasive

Usage: `A mate B   mate A B`

See Also: match, gt, gte, lt, lte

The operation *mate* compares two atoms *A* and *B* for equality with type coercion, returning *true* if *A* is equal to or can be coerced to *B* as a number; and *false* otherwise. *Mate* is used to determine the equality of atoms of different numeric type, as when comparing the real number 3.0 with the integer 3. It cannot be used to determine equality of literal types.

```
     3 mate 3
l

     3.0 mate 3
l

     "o mate `o
o

     1.0 mate l 1 1.0
lll
```

The last example shows that the binary pervasive extension of *mate* allows a lists of atoms to be compared for a value, returning a bitstring of results.

Definition

```
     mate IS OPERATION A B { (A lte B) and (B lte A) }
```

max

Class: comparison operation

Properties: multi pervasive, reductive

Usage: `max A   A max B`

See Also: min, gte, sort

The operation *max*, when applied to a simple array *A*, finds the least atom that is greater than or equal to all the items. If the atoms of *A* are all positive numbers, for example, *max A* returns the biggest. If the items are not comparable, the result is the fault *?I*, the atom in the lattice of Nial atoms that is greater than or equal to all other atoms (the zenith). If A is empty, the result is the fault *?O*, the atom that is less than or equal to all other atoms (the nadir).

Max is extended to arbitrary arrays by its multi pervasive behaviour. It is a reductive operation in that it reduces a simple array to a single atom. Applied to a pair of simple arrays, it produces a simple array with the corresponding items compared.

If the items of *A* are all numeric type, they are comparable. The result is the highest numeric type represented in the array, where boolean is the lowest numeric type and real number is the highest.

```
     max 3 45 23 18 3.5
45.
     max 3 "abc
?I
     max "abc "def "c
def
     max 'apples' 'orange'
orpngs
```

The first example shows that *max* of a list of numbers of different type is the maximum number, coerced to the highest type. The second shows that the integer 3 and the phrase "abc" are incomparable and the result is the fault ?I. The third example shows that phrases are directly comparable. The last shows that strings are compared on a character by character basis.

Equations

```
A max B = B max A
max EACH max A =f= max link A
atomic A ==> max B lte A = and (B EACHLEFT lte A)
max Null = ??O
```

min

Class: comparison operation

Properties: multi pervasive, reductive

Usage: `min A   A min B`

See Also: max, lte, sort

The operation *min*, when applied to a simple array *A*, finds the greatest atom that is less than or equal to all the items. If the atoms of *A* are all positive numbers, for example, *min A* returns the smallest number. If the items are not comparable, the result is the fault *?O*, the atom in the lattice of Nial atoms that is less than or equal to all other atoms (the nadir). If *A* is empty, the result is the fault *?I*, the atom that is greater than or equal to all other atoms (the zenith).

Min is extended to arbitrary arrays by its multi pervasive behaviour. It is a reductive operation in that it reduces a simple array to a single atom. Applied to a pair of simple arrays, it produces a simple array with the corresponding items compared.

If the items of *A* are all numeric type, they are comparable. The result is the highest numeric type represented in the array, where boolean is the lowest numeric type and real number is the highest.

```
     min 3 45 23 18 3.5
3.
     min 3 "abc
?O
     min "abc "def "c
abc
     min 'apples' 'orange'
apalee
```

The first example shows that *min* of a list of numbers of different type is the minimum number, coerced to the highest type. The second shows that the integer 3 and the phrase "abc" are incomparable and the result is the fault *?I*. The third example shows that phrases are directly comparable. The last shows that strings are compared on a character by character basis.

Equations

```
A min B = B min A
min EACH min A = min link A
atomic A ==> A lte min B = and (A EACHRIGHT lte B)
min Null = ??I
```

minus

Class: arithmetic operation

Property: binary pervasive

Usage: `A minus B   A - B   minus A B`

See Also: plus, sum, opposite

The operation *minus* returns the result of subtracting two numeric atoms. It coerces the type of the atoms to be the higher type, if they differ; or to integer, if both are boolean. The result is of the type of the coerced arguments.

The symbol - is a synonym for *minus*. Care must be taken when using this symbol because it also is used to form negative constants and the latter use has higher priority. Thus,

```
     A := 10;
     A - 1
9
```

does a subtraction, but

```
     A -1
10 -1
```

forms a pair.

If one argument is numeric and the other is a fault or if both arguments are the same fault, the answer is the fault. In all other cases when one or more of the arguments is not numeric, the result is the arithmetic fault *?A*.

```
     R gets 1 2 2.5 `a "abc ??error
1 2 2.5 a abc ?error

     R OUTER minus R
     0      -1    -1.5   ?A  ?A  ?error
     1       0    -0.5   ?A  ?A  ?error
   1.5     0.5      0.   ?A  ?A  ?error
?A       ?A       ?A       ?A  ?A  ?A
?A       ?A       ?A       ?A  ?A  ?A
?error   ?error   ?error   ?A  ?A  ?error
```

The above example illustrates all combinations of atom types for the two arguments to *minus*.

Equations

```
   0 minus A = opposite A
   A minus B = A plus opposite B
```

mix

Class: nesting restructuring operation

Usage: `mix A`

See Also: blend, link, rows, rank

The operation *mix* is intended to be used on an array *A* with items which are all the same shape. In this case, the shape of the result is formed by the shape of *A* linked with the shape of the items. The list of items of the result is the link of the items of *A*. If *A* does not have equally shaped items, the fault *?conform* is returned.

```
     mix (2 3) (4 5) (6 7)
2 3
4 5
6 7

     A := (tell 2 3)(count 2 3)
+-------------+-------------+
|+---+---+---+|+---+---+---+|
||0 0|0 1|0 2|||1 1|1 2|1 3||
|+---+---+---+|+---+---+---+|
||1 0|1 1|1 2|||2 1|2 2|2 3||
|+---+---+---+|+---+---+---+|
+-------------+-------------+

     mix A
+---+---+---+  +---+---+---+
|0 0|0 1|0 2|  |1 1|1 2|1 3|
+---+---+---+  +---+---+---+
|1 0|1 1|1 2|  |2 1|2 2|2 3|
+---+---+---+  +---+---+---+
```

The first example shows that the result of mixing a triple of pairs is a 3 by 2 table. The second example shows that mixing a pair of 2 by 3 tables is a 2 by 2 by 3 array, with the tables as the planes of the array.

The operation *mix* is the left inverse of the operations *rows* and *raise* for nonempty arrays. It is often used to transform from a *list of lists* view of data to a table view of data. In practice, transformations between the two representations on large arrays should be avoided in Q'Nial since considerable work is involved in doing the restructuring.

Definition

```
     mix IS OPERATION A {
        IF empty A THEN
           shape A append 0 reshape A
        ELSEIF not equal EACH shape A THEN
           ??conform
        ELSE
           shape A link shape first A reshape link A
        ENDIF }
```

Equations

```
   not empty A and equal EACH shape A ==> shape mix A = link (shape A)(shape first A)
   equal EACH shape A ==> list mix A = link A
   not empty A ==> mix rows A = A
   I in (axes A + 1) and not empty A ==> mix (I raise A) = A
```

mod

Class: arithmetic operation

Property: binary pervasive

Usage: `A mod B   mod A B`

See Also: quotient, ceiling, floor

The operation *mod* returns the remainder of dividing integer *A* by integer *B*. If the divisor *B* is zero, the result is *A*. If it is negative, the result is the fault *?negative divisor*.

The operation *mod* coerces a boolean argument to an integer. It produces a fault if either argument is not an integer or a boolean.

```
     R gets 1 2 2.5 `a "abc ??error
1 2 2.5 a abc ?error

     R OUTER mod R
     0       1  ?A      ?A  ?A  ?error
     0       0  ?A      ?A  ?A  ?error
?A      ?A      ?A      ?A  ?A  ?error
?A      ?A      ?A      ?A  ?A  ?A
?A      ?A      ?A      ?A  ?A  ?A
?error  ?error  ?error  ?A  ?A  ?error

     (5 mod 3)(-5 mod 3)(5 mod -3)(5 mod 0)
2 1 ?negative divisor 5
```

The first example illustrates all combinations of atom types for the two arguments to *mod*.

The remainder on division by a positive integer *B* is always a number between 0 and *B - 1*. If *A* is negative, the integer returned obeys the rule of modular arithmetic as described in the definition. The result of *A mod 0* is *A* rather than a fault, since that is the preferred result in mathematics.

Definition

```
mod IS OPERATION A B { floor (A - (B * (A quotient B)) ) }
```

Equation

```
isinteger A ==> A mod 0 = A
```

mold

Class: reshaping operation

Usage: `A mold B   mold A B`

See Also: take

The operation *mold* is provided for backward compatibility with earlier versions of Nial. The current definition is a synonym for *take*. The previous definition was a reshaping operation that was almost always used on lists and *take* has the same effect as the previous definition on lists.

Definition

```
mold is take
```

multi pervasive

Class: operation property

Usage: `f A   f A B ... C`

See Also: unary pervasive, binary pervasive, pervasive, reductive, eachall, reduce, pack

Each operation *f* in this class reduces a simple array (an array of atoms) to an atom.

A multi pervasive operation maps an array having items of identical structure to one with the same structure, applying the operation to the simple arrays formed from all the atoms in corresponding positions in the atoms.

There are six operations in this class. They are the **reductive** operations of arithmetic and logic.

If a multi pervasive operation is applied to an array that does not have items of the same shape, the effect is to build a conformable pair by replicating an atom or solitary item of the array to the common shape of the other items. If two or more of the items have tally different from one and are of unequal shape, the fault *?conform* is returned. On a pair, a multi pervasive operation behaves the same way as a binary pervasive operation does.

```
     sum [3 4 5 6 , 5 , 7 8 9 10] = 15 17 19 21
1
```

The following table describes the multi pervasive operations.

Operation	Function
and	logical "and" of a boolean array
max	highest item in the array
min	lowest item in the array
or	logical "or" of a boolean array
product	arithmetic product of an array of numbers
sum	arithmetic sum of an array of numbers

Equations

```
f A = EACHALL f A
f A = EACH f pack A
shape f A = shape pack A
f A = REDUCE f A
```

nested definition

Class: concept

See Also: block, definition

A **nested definition** is one that appears in a definition sequence within a block. Its name is a local definition name. If the name is also used outside the block, the external meaning is not known in the block.

Nested definitions can be used to encapsulate support definitions within a larger definition that is to be made available to other users. This avoids cluttering up the name space with names that might interfere with the user's other work. It is often easier to develop the large definition without encapsulation and package it in encapsulated form once the design is completed.

next

Class: debugging command

Usage: `next`

See Also: debugging, break, resume, step, toend, stepin

The command *next* is used in debugging a definition that has been suspended using Break or <Ctrl B>. The effect of *next* is to execute the next expression in an expression sequence and to suspend execution again. If the current expression involves a call on a defined operation or expression, execution of the entire definition is completed and execution is suspended on the expression after the current one. If the current expression is the last one in the expression sequence where the break began, the effect is the same as using *resume*.

The related command *nextv* displays the result of the expression executed on a step before displaying the next expression.

nialroot

Class: system expression

Usage: `Nialroot`

See Also: host, link, library

The expression *Nialroot* returns a string giving the path to the directory holding the help directory, the initial directory and the library directories. Nialroot is defined before Q'Nial is run. In Windows and Unix versions this is set up as part of the installation procedure. Under EXTDOS, it is defined using a set command such as:

```
set nialroot=C:\qnial62
```

no_expr

Class: system expression

Usage: `No_expr`

See Also: no_value, no_op, no_tr

The expression No_expr returns the fault *?missing_expr*. It is the default value for an expression name declared to be *external* or that has been erased.

no_op

Class: system operation

Usage: `no_op A`

See Also: no_value, no_expr, no_tr

The operation *no_op* returns the fault *?missing_op* when applied to any array *A*. It is the default value for an operation name declared to be external or that has been erased.

no_tr

Class: system transformer

Usage: `no_tr f A`

See Also: no_value, no_expr, no_op

The transformer *NO_TR* transforms any operation *f* to an operation that returns the fault *?missing_tr* when applied to any array *A*. It is the default value for a transformer name declared to be external or that has been erased.

no_value

Class: system constant

Usage: `?no_value`

See Also: no_expr, no_op, no_tr

The fault *?no_value* is a special fault value used as the default value of variables that have been declared external or that have been erased.

not

Class: logic operation

Property: unary pervasive

Usage: `not A`

See Also: and, or, true, false, isboolean, notin

The operation *not* reverses the value of a boolean. It returns *true* if *A* is *false* and *false* if *A* is *true*. If *A* is any other atom, the result is the logical fault *?L*.

```
     not l
o
     not lollo
olool
     not 2 "abc o
?L ?L l
```

Equations

```
   not not not A = not A
   not and A =f= or EACH not A
   not or A =f= and EACH not A
```

notin

Class: search operation

Properties: binary, predicate

Usage: `A notin B   notin A B`

See Also: in, not, allin, unequal

The operation *notin* returns *true* if *A* is not an item of *B* and returns *false* if it is.

```
     3 notin 56 34 23 3 57 3
o

     4 1 notin tell 3 4
l

     `a notin 'hello world'
l

     `a notin "apple
l
```

Definition

```
     notin IS OPERATION A B { not (A in B) }
```

Equation

```
   A notin B = A notin list B
```

null

Class: constant expression

Usage: `Null`

See Also: empty, atomic

The expression *Null* denotes the empty list. It is the array returned as the shape of a single and is equal to the empty string. The empty list notation *[ ]* also denotes the empty list *Null*.

```
     Null = shape 5
1
     Null = []
1
     Null = ''
1
```

Since there is an empty array for each shape containing a zero, unless you are certain *A* is a list, the test: *A = Null* should not be used in place of *empty A*.

```
     set "sketch; Null

     set "diagram; Null
+
|
+
```

Null is not displayed when sketch display mode is set. In diagram mode it is displayed as the left border of a list diagram.

A selection operation such as *pick* or *first* applied to *Null* results in the fault *?address*.

Definition

```
     Null IS shape 0
```

Equations

```
   shape single A = Null
   shape Null = 0
   atomic A ==> Null = shape A
```

numeric type hierarchy

Class: concept

See Also: sum, or

The three numeric types: boolean, integer and real, are organized in a hierarchy in the order mentioned. For arithmetic and comparative binary operations, if the types of arguments are both numeric but differ in type, they are coerced to the higher of the two types. For example, if you add an integer to a real number, say 3 + 4.5, the integer is coerced to 3.0 and a real number addition is done. Boolean atoms are always coerced to integers if they are used with arithmetic operations.

There are six types of atoms in Nial. They are boolean, integer, real, character, phrase and fault. The first three are numeric types and are used for arithmetic operations. The last three are literal types and are used

for symbol manipulation. All six types of atoms are used in comparisons.

A boolean atom is the result of a comparison of array values; or the result of a test relating to a characteristic of an array or the content of an array. There are two boolean atoms: true and false, denoted by l and o respectively. When booleans are treated as numbers, true corresponds to one and false to zero.

An integer atom is a positive or negative whole number representing a quantity of units. A dash symbol (-) immediately preceding the integer denotes a negative integer. No space is permitted between the dash and the number, otherwise the dash is interpreted as the arithmetical operation of subtraction. Conversely, a space is required when subtraction is intended. An integer is represented by an internal form that limits its range of values in the Q'Nial implementation of Nial.

A real atom is a number which can represent any position on the real line. It may be written with a fractional part and/or with a decimal exponent. It is represented internally by a floating point number.

open

Class: file operation

Usage: `open Filename Mode`

See Also: close, readfile, writefile, readarray, writearray, readrecord, writerecord

The operation *open* prepares the host file named by the phrase or string *Filename* for use in one of seven possible modes. *Mode* is a character, string or phrase encoding the mode as follows:

Mode	Mode Name	Description
r	read	existing sequential text file, read only
w	write	create sequential text file, write only
a	append	existing sequential text file, write only starting at the end
d	direct	existing or new binary file, direct access for both read and write
c	communication	existing or new sequential text file, for both read and write
pr	pipe read	output from the command Filename is available
pw	pipe write	input to the command Filename is provided

In *read*, *write*, *append* or *communication* modes, *Filename* must include the file extension, if any. In direct mode, two host files are opened, one with extension *.ndx* and the other with extension *.rec*, standing for *index* and *record*, respectively. In this mode, *Filename* is the root of the file name, without any extension.

The *communication* mode can be used with special forms of *readfile* and *writefile* usage to access a special device or a communications port. This mode is available only on the EXTDOS Version of Nial.

The pipe modes treat the *Filename* as a host command. The *open* causes the command to be executed. In *pipe read*, the output from the command is obtained using *readfile*; in *pipe write*, input to the command is provided using *writefile*.

The result of *open* is a file designator, an integer specifying an input/output port. This value is used in later references to the file.

The effect of *open* may be system dependent. If the file exists and can be opened by the user in the requested mode, the result will be the file designator. If it cannot be opened or does not exist, the result will be a fault message generated from the error message provided by the host system. For portable programs, the specific fault message should not be used in an equality test. The result of an open should be tested for a fault value using *isfault* to catch unexpected failures.

operation

Class: concept

See Also: expression, transformer

An **operation** is a functional object that is given an argument array and returns a result array. The process of executing an operation by giving it an argument value is called an *operation call* or an *operation application*.

An operation can be constructed by defining one or more parameters and giving an algorithm to compute the result in terms of the parameters. An operation is usually given a name when it is defined. There are also program fragments that construct unnamed operations by composing operations, forming a list of operations or modifying an operation by use of a transformer.

A named operation is either a predefined operation or an operation that has been given an explicit name using the *IS* definition mechanism.

operation composition

Class: concept

See Also: curried operation

A sequence of simple-operation denotations forms an operation-sequence in the syntax of Nial. Its meaning is the same as the composition of the operations in the sequence which is explained by its effect when applied.

The result of applying an operation-sequence to an argument is determined by applying the simple-operations in the sequence in right-to-left order. The simple-operation on the right is applied to the argument giving an intermediate result. Then the simple-operation to the immediate left is applied to the result of the first application. Subsequent simple-operations are applied to the results in turn.

An example of this concept is the expression

```
     (first rest) 8 7 2 5 3
7
```

The expression sequence *first rest* is evaluated by applying *rest* to the argument resulting in 7 2 5 3 and then *first* is applied giving 7.

The relationship can be described by the equations:

```
(f g) A = f (g A)
(f g h) A = f (g (h A))
etc.
```

which is mathematically known as the rule of **function composition**.

operation form

Class: syntax

See Also: operation, definition

An **operation-form** is the syntactic structure used to describe an operation in terms of a parameterized expression-sequence. The identifiers following the keyword operation are called the **formal parameters**. The body of an operation-form is normally a block but it may be an expression-sequence without automatic localization.

```
operation-form ::=
   OPERATION { identifiers }+ block
   | OPERATION { identifiers }+
     ( expression-sequence )
```

An operation-form defines a local environment. The formal parameter names are names of local variables. If the body of the operation form is a block, the local environment of the block is extended to include the formal parameters. When the operation is applied, the formal parameter names are assigned from the value of the actual argument. If there is only one formal parameter, the actual argument is assigned to it as a whole; otherwise, the items of the actual argument are assigned to the formal parameters in corresponding order. If there is a length mismatch between the list of formal parameter names and the values of the actual argument, the fault *?op_parameter* is returned.

The value of the application of the operation is the value of the body of the operation-form, which is evaluated with the local variables in the parameter list assigned as described above. In determining the association for a name that appears in the body of an operation form, Q'Nial looks for the name in the local environment. If the name is not found locally, the name is sought in surrounding environments until it is found or until the global environment is searched. If it is not found, a fault *?unknown identifier:* is given when the operation-form is analyzed (parsed).

Operation-forms are most frequently used in definitions where they are given an associated name. However, an operation-form can appear directly in an expression provided it is enclosed in parentheses. In this usage, it can be an argument to a transformer name or can be applied to an array argument.

opposite

Class: arithmetic operation

Property: unary pervasive

Usage: `opposite A   opp A`

See Also: minus, not, reciprocal

The operation *opposite* returns the opposite value of the integer or real number *A*. The opposite value of a positive number A is the corresponding negative value and vice versa. The operation produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	opposite of the corresponding integer
integer	opposite integer
real	opposite real number
character	fault ?A
phrase	fault ?A
fault	argument A

The name *opp* is a synonym for *opposite* used as an abbreviation.

```
     opposite  1  -2  3.5  `a  "abc  ??err
-1 2 -3.5 ?A ?A ?err
```

In Nial, the symbol - is used as part of negative numbers and as a synonym for *minus*. Care must be taken to use a space after the symbol before a constant to get the expected subtraction.

Definition

```
     opposite IS OPERATION A (0 minus A)
```

Equations

```
   opposite opposite opposite A = opposite A
   A - B = A + opposite B
```

ops

Class: system expression

Usage: `Ops`

See Also: symbols, exprs, trs, vars

The expression *Ops* returns a list of phrases giving the names of all user defined operations in the workspace.

Definition

```
Ops IS {
   Names Roles := pack symbols 0;
   "op match Roles sublist Names }
```

or

Class: logic operation

Properties: multi pervasive, reductive

Usage: `or A   A or B`

See Also: and, not

The operation *or* applies to a boolean array *A* and does the boolean sum of its items. If any item of *A* is *true*, the result is *true*; otherwise it is *false*. Used in binary form, it implements the usual or-connective of logic. If *A* is a simple array but has a non-boolean item, the result is the logical fault *?L*. The operation extends to non-simple arrays by the multi pervasive mechanism.

```
     l or o
l

     or ooooloooo
l

     lloo or lolo
lllo
```

Or is a reductive operation in that it reduces an array of booleans to a single boolean. Applied to a pair of bitstrings, it produces a bitstring resulting from applying or to the bits of its arguments in corresponding positions.

Equations

```
A or B = B or A
not or A =f= and EACH not A
```

outer

Class: applicative transformer

Usage: `A OUTER f B   OUTER f A`

See Also: cart, inner

The transformer *OUTER* transforms an operation *f* to an operation that applies *f* to each item of the cartesian product of its argument. In the case where the arguments are a pair of lists *A* and *B*, the result is a table of the applications of *f* with every pair chosen with the first item from *A* and the second from *B*.

```
     1 2 3 4 5 OUTER * 1 2 3 4 5
1  2  3  4  5
2  4  6  8 10
3  6  9 12 15
4  8 12 16 20
5 10 15 20 25

     2 3 OUTER reshape 'abcd'
+---+---+---+---+
|aa |bb |cc |dd |
+---+---+---+---+
|aaa|bbb|ccc|ddd|
+---+---+---+---+
```

The first example shows *OUTER* * being used to generate a multiplication table. The second shows that it can be applied to an operation that has a non-atomic result.

Definition

```
OUTER IS TRANSFORMER f OPERATION A (EACH f cart A)
```

Equations

```
shape OUTER f A = link EACH shape A
and EACH (not empty) A B ==> A OUTER f B = mix (A EACHLEFT EACHRIGHT f B)
```

overflow

Class: concept

The term **integer overflow** is used to describe the situation when an integer that is too big for the internal representation of integers is generated. Q'Nial uses 32 bit integers and checks for overflow during computations. If an arithmetic computation produces an overflow, a fault value is produced.

An overflow can also develop during computation with real numbers. The computer hardware detects such a problem and an interrupt occurs which causes Q'Nial to jump to top level.

pack

Class: data rearrangement operation

Usage: `pack A`

See Also: eachall, eachboth

The operation *pack* interchanges the top two levels of an array with equally shaped items. If *A* is an array with items all of the same shape, the result is an array of that shape, with items having the shape that *A* has. If *A* does not have items of the same shape but all the items that have zero items or more than one item are of the same shape, it is modified by replicating the items with one item to that shape before interchanging levels. Otherwise it returns the fault *?conform*.

The items of the result are the arrays of items of *A* from corresponding positions. *Pack* is used implicitly in all binary pervasive and multi pervasive operations.

```
     pack (2 3 4) (4 5 6)
+---+---+---+
|2 4|3 5|4 6|
+---+---+---+

     pack 'ab' 'cd' 'ef' 'gh'
+----+----+
|aceg|bdfh|
+----+----+

     pack (tell 2 3) (count 2 3)
+---------+---------+---------+
|+---+---+|+---+---+|+---+---+|
||0 0|1 1|||0 1|1 2|||0 2|1 3||
|+---+---+|+---+---+|+---+---+|
+---------+---------+---------+
|+---+---+|+---+---+|+---+---+|
||1 0|2 1|||1 1|2 2|||1 2|2 3||
|+---+---+|+---+---+|+---+---+|
+---------+---------+---------+

     pack (2 3 4) 5 [2 3] 'abc'
+-----------+-----------+-----------+
|+-+-+---+-+|+-+-+---+-+|+-+-+---+-+|
||2|5|2 3|a|||3|5|2 3|b|||4|5|2 3|c||
|+-+-+---+-+|+-+-+---+-+|+-+-+---+-+|
+-----------+-----------+-----------+
```

The first example shows that *pack* of a pair of triples is a triple of pairs. The second shows that *pack* of a quadruple of pairs is a pair of quadruples. The third shows that *pack* of a pair of 2 by 3 tables is a 2 by 3 table of pairs. The final example shows a case where replication occurs. The first and last items have shape [3]. The second and third items are an atom and a solitary and hence can be replicated.

Equations

```
equal EACH shape A and not empty pack A ==> pack pack A = A
pack pack pack A = pack A
pack single A = EACH single A
pack solitary A = EACH solitary A
empty A ==> pack A = single A
simple A ==> pack A = single A
```

pair

Class: construction operation

Usage: `A pair B` `pair A B`

See Also: link, hitch, append, reshape

The operation *pair* has no effect on a pair but converts all other arrays to a pair by reshaping the array with shape 2. Used in infix notation, it has the effect of forming a pair from the two arguments. An explicit pair can also be achieved with list or strand notation. The operation is useful in functional expressions where it is an argument to a transformer.

```
     3 pair 4 5
+-+---+
|3|4 5|
+-+---+
     pair 3 4 5
3 4

     pair single 3 4 5
+-----+-----+
|3 4 5|3 4 5|
+-----+-----+
```

The major use of *pair* is as an operation applied using a transformer.

Definition

```
     pair IS OPERATION A (2 reshape A)
```

Equations

```
   pair pair A = pair A
   A OUTER pair B = A cart B
```

parse

Class: evaluation operation

Usage: `parse Tokenstream`

See Also: scan, deparse, descan, defedit, eval, execute

The operation *parse* carries out the internal parsing capability of the interpreter. Its argument is a token stream produced by *scan* or *deparse*. The result is the parse tree recording the semantic intent of the input. The parse tree is a highly encoded form of nested array designed for efficient processing.

```
     parse scan 'sum 23 45'
+---+-----------------------------+
|100|+-+-----+-------------------+|
|   ||4|3 4 3|+-+-------+-------+||
|   || |     ||8|1 23 23|1 45 45|||
|   || |     |+-+-------+-------+||
|   |+-+-----+-------------------+|
+---+-----------------------------+

     parse scan 'foo IS first'
+---+-----------------------------+
|100|+--+---------------+------+-+|
|   ||11|+--+----------+|3 47 3|1||
|   ||  ||22|2 160 7459||      | ||
|   ||  |+--+----------+|      | ||
|   |+--+---------------+------+-+|
+---+-----------------------------+
```

The first example gives the parse tree corresponding to an application of an operation to a strand of constants. The second example gives the parse tree for the definition of *foo* as the predefined operation *first*. The small integers that appear as the first items of the arrays serve as tags to indicate the kind of node that is represented in the parse tree. They are used by the *eval* and *deparse* operations. The representation of parse trees is implementation dependent and is not intended for program manipulation.

Equations

```
Ts a token stream ==> parse deparse parse Ts = parse Ts
S a string ==> eval parse scan S = execute S
```

partition

Class: apply-by-partition transformer

Usage: `I,J] PARTITION f A   I] PARTITION f A`

See Also: rank, split, blend

The transformer *PARTITION* is used to apply operation *f* to partitions of the array *A*, placing the results of the applications as partitions of the result array.

The partitions of *A* are determined by the axis numbers in *I*, using (*I split A*) to form them implicitly. The results of applying *f*, viewed as an intermediate array *R*, are combined into the final result, using (*J blend R*), to place the axes of the items of *R* into the final result. If the left argument is a solitary, it is assumed to be used as the argument to both *split* and *blend* implicitly.

For a table *A*, *I = 1*, *f* being *reverse*, the effect is to reverse the rows of *A*. In general, if *I = J* and is a single axis number and *f* maps a list to one of the same length, the effect is to apply *f* to the lists formed by pushing axis *I* down, resulting in an array of the same shape as *A*. Two dimensional partitions of a three dimensional array can be mapped using an *I* with tally 2.

```
     [1] PARTITION reverse (3 5 reshape count 15)
 5  4  3  2  1
10  9  8  7  6
15 14 13 12 11

     [0] PARTITION (2 rotate) (6 6 reshape tell 36)
12 13 14 15 16 17
18 19 20 21 22 23
24 25 26 27 28 29
30 31 32 33 34 35
 0  1  2  3  4  5
 6  7  8  9 10 11

     setformat '%4.2f';
     seed .5 ; A:=2 3 3 reshape random 18
0.50 0.43 0.12   0.03 0.81 0.24
0.77 0.73 0.89   0.58 0.98 0.47
0.98 0.66 0.16   0.24 0.16 0.50

     B gets 1 2 split A
+--------------+--------------+
|0.50 0.43 0.12|0.03 0.81 0.24|
|0.77 0.73 0.89|0.58 0.98 0.47|
|0.98 0.66 0.16|0.24 0.16 0.50|
+--------------+--------------+

     C gets EACH inverse B
+-----------------+-----------------+
|-7.43  0.18  4.69|-2.51  2.21 -0.90|
|11.78 -0.62 -5.55| 1.09  0.25 -0.76|
|-3.27  1.48  0.51| 0.83 -1.13  2.69|
+-----------------+-----------------+
```

```
     1 2 blend C
-7.43  0.18  4.69   -2.51  2.21 -0.90
11.78 -0.62 -5.55    1.09  0.25 -0.76
-3.27  1.48  0.51    0.83 -1.13  2.69

     [1 2] PARTITION inverse A
-7.43  0.18  4.69   -2.51  2.21 -0.90
11.78 -0.62 -5.55    1.09  0.25 -0.76
-3.27  1.48  0.51    0.83 -1.13  2.69
```

The transformer *RANK* is a simpler version of a partitioning transformer that always pushes down the right end axes.

Definition

```
PARTITION IS TRANSFORMER f OPERATION Ij A {
   IF empty Ij or (tally Ij > 2) THEN
      fault 'invalid left arg of PARTITION transform'
   ELSEIF empty A THEN
      A
   ELSE
      IF tally Ij = 1 THEN
         I := J := first Ij;
      ELSEIF tally Ij = 2 THEN
         I J := Ij;
      ENDIF;
      II gets axes A except I link I;
      B := tally I RANK f (II fuse A);
      IF J = Null and (shape B = Null) THEN
         first B
      ELSEIF tally J = (valence B - (valence A - tally I)) THEN
         JJ gets axes B except J link J;
         GRADE <= JJ fuse B
      ELSE
         fault 'left arg incompatible with function in PARTITION transform'
      ENDIF
   ENDIF }
```

Equations

```
   I split A = \[I,Null] PARTITION single A
   I blend A = \[Null,I] PARTITION first A
   EACH f A = \[Null,Null] PARTITION (single f first) A
```

pass

Class: reshaping operation

Usage: `pass A`

See Also: list, reshape

The operation *pass* is the identity operation for arrays. It returns the argument *A*. *Pass* is used when expressing computations in a functional style using an atlas (a list of operations).

```
     pass 2 3 4
2 3 4
     pass 'hello world'
hello world
```

Definition

```
pass IS OPERATION A { A }
```

Equations

```
pass A = first single A
pass A = first solitary A
```

paste

Class: picture operation

Usage: `Sw paste A`

See Also: picture, sketch, diagram, positions

The operation *paste* combines character tables in a two dimensional layout. It is used internally by *picture* and can be used to create modifications of the standard picture output to improve the display of results. The left argument *Sw* has six parameters that control the layout and the right argument *A* is the array of tables to be combined. The components of argument *Sw* are the following:

Argument	Position
Vertical edge spacing	an integer indicating the number of blank lines to be put between the rows of items
Horizontal edge spacing	an integer indicating the number of blank spaces to be put between the columns of items
Vertical line indicator	If the value is 1, vertical lines are drawn between columns. If it is 0, they are not
Horizontal line indicator	If the value is 1, horizontal lines are drawn between rows. If it is 0, they are not
Vertical justification indicator	This is either a single integer indicating the same justification for all fields or an array of integers of the same shape as A indicating the justification for each item of A. The codes are: 0 for top justification, 1 for centered and 2 for bottom justification
Horizontal justification indicator	This is either a single integer indicating the same justification for all fields, or an array of integers of the same shape as A indicating the justification for each item of A. The codes are: 0 for left justification, 1 for centered and 2 for right justification

If either line indicator is 1, the entire result is boxed.

```
     1 2 1 3 1 1 paste EACH sketch (2 4 reshape 'some' 'words' 'of' 'wisdom')
+--------+---------+------+----------+
|        |         |      |          |
|  some  |  words  |  of  |  wisdom  |
|        |         |      |          |
+--------+---------+------+----------+
|        |         |      |          |
|  some  |  words  |  of  |  wisdom  |
|        |         |      |          |
+--------+---------+------+----------+
```

pervasive

Class: concept

See Also: unary pervasive, binary pervasive, multi pervasive

Some of the operations of Nial that operate on simple arrays are extended to arbitrarily nested arrays by being applied to the atoms at the deepest level. These are called **pervasive** operations.

There are three classes of pervasive operations: *unary pervasive*, *binary pervasive* and *multi pervasive*. The first class applies the operations to the atoms of the array. The second class are binary operations that apply to pairs of atoms from corresponding positions in the pair of arguments. The third class applies the operation to the simple arrays formed from atoms in corresponding positions in the items of the argument, reducing the simple array to a single atom.

phrase

Class: conversion operation

Usage: `phrase S`

See Also: string, fault, tonumber, isphrase

The operation *phrase* converts a string into a phrase with the same content. It returns phrases and faults unchanged and converts other atoms after coercing them to their string equivalents. The resulting phrase may contain blanks. There is a one-to-one correspondence between phrases and strings.

```
     phrase '(613) 549-2222'
(613) 549-2222

     set "decor ; EACH phrase o 613 3.5 `A 'Now' "Wow ?error
"o "613 "3.5 "A "Now "Wow "error
```

Equations

```
   isphrase P ==> phrase string P = P
```

Pragmatics

The correspondence between phrases and strings is not complete in that phrase cannot hold a null character (representation 0) and attempting to build such a phrase using the operation *phrase* will result in a truncated phrase.

pi

Class: constant expression

Usage: `Pi`

See Also: random, ln

The expression *Pi* returns the real number which is the ratio of the circumference of a circle to its diameter. *Pi* is useful in scientific computing.

```
     setformat '%15.12f' ; Pi
 3.141592653590
```

Equation

```
   Pi = arccos -1 (within roundoff error)
```

pick

Class: selection operation

Property: binary

Usage: `I pick A   pick I A`

See Also: choose, take, first, address

Pick is the fundamental selection operation for arrays. The normal use is that *I* is an address for *A*. The result is the item that is located at address *I* in 0-origin addressing. If *I* is a solitary integer and *A* is a list, *I* is converted to an integer to make it an address for a list. If *I* is not a valid address for *A*, the result is the fault *?address*.

```
     3 pick 3 6 23 46 57 25
46

     [3] pick 3 6 23 46 57 25
46

     A := 2 3 reshape "a "list "of "words "as "phrases
a     list of
words as   phrases

     0 2 pick A
of

     5 5 pick A
?address
```

In the first example, *pick* selects the fourth item because of 0-origin indexing. In the second example, [3] is not an address of the list (its addresses are integers) but it is converted to integer 3. Thus, a solitary integer can be used as an address of a list for *pick*. The third example shows that an address for a table is a pair of integers. The last example shows that picking out of range returns a fault.

The concept of picking closely corresponds to subscript notation in mathematics. Q'Nial supports a direct notation for subscripting a variable: *Var@I*. This notation, called the *at notation*, can be used on both the left and right of an assignment expression. Its use on the right is equivalent to *pick*.

Equations

```
   I pick A = (list I) pick A
   I in grid A ==> I pick grid A = I
   I in grid A ==> I pick (EACH f A) = f (I pick A)
   A choose B = A EACHLEFT pick B
   I pick A = \[I] reach A
```

picture

Class: picture operation

Usage: `picture A`

See Also: sketch, diagram, display, paste, positions, set

The operation *picture* computes a character table that describes the array *A*.

An atom is displayed directly. A nonatomic array is displayed as a tableau of rows and columns arranged to allow sufficient space for the largest items. In most cases, the items of the array are displayed within a frame diagram.

The operation is affected by the setting of two internal switches: *diagram/sketch* and *decor/nodecor*. The first controls the use of frames in the pictures. The second controls the decoration of atoms, solitaries, strings and empty arrays.

In *diagram* mode, a nonatomic array is displayed with a frame and the upper left corner has an *o* if it is a single. In sketch mode, a simple array, one having atomic items, is displayed without a frame. A non-simple array has a frame and a nonatomic single has an *o* in the upper left corner of the frame.

The *decor/nodecor* switch pictures atoms in an undecorated manner if nodecor mode is set. If sketch mode is also set, strings and phrases appear the same. The decor mode pictures atoms so that they can be distinguished: characters are preceded by a grave symbol `, phrases by a double quote ″ and faults by a question mark *?*. In sketch mode with decor set, a string is enclosed in quotes and a simple solitary is placed in brackets.

In sketch mode an empty array has an empty picture, whereas in diagram mode, the picture of an empty array is one side of a frame diagram, with no items.

An array of dimensionality higher than two is laid out as an arrangement of tables. If the array has valence 3, the tables are arranged horizontally. If it has valence 4, the first axis is placed down the page and the second axis across the page. Thus, an array of shape 3 4 5 6 is pictured as 3 rows of 4 tables each of which is a 5 by 6 tableau. A space is left between each of the tables. For higher dimensions, the axes continue to alternate across and down the page with greater spacing between arrangements.

The result of *picture* coincides with the output displayed by Q'Nial in the top level loop.

In the top level loop, the picture is wrapped around as a whole if it is too wide to fit on the screen.

```
     set "sketch; set "nodecor;
     P := picture 1 1 7 7.0 `a "a 'a' (3 4 5) Null
+-+-+-+--+-+-+-+-----++
|1|1|7|7.|a|a|a|3 4 5||
+-+-+-+--+-+-+-+-----++

     set "diagram; set "nodecor; P
+-+-+-+--+-+-+---+-------+-+
|1|1|7|7.|a|a|+-+|+-+-+-+|+|
| | | |  | | ||a|||3|4|5||||
| | | |  | | |+-+|+-+-+-+|+|
+-+-+-+--+-+-+---+-------+-+
```

```
     set "sketch; set "decor; P
+-+-+-+--+--+--+---+-----++
|l|1|7|7.|`a|"a|'a'|3 4 5||
+-+-+-+--+--+--+---+-----++


     set "diagram; set "decor; P
+-+-+-+--+--+--+----+-------+-+
|l|1|7|7.|`a|"a|+--+|+-+-+-+|+|
| | | |  |  |  ||`a|||3|4|5||||
| | | |  |  |  |+--+|+-+-+-+|+|
+-+-+-+--+--+--+----+-------+-+

     set "diagram; set "decor;
     picture 3 4 5
+--+--+--+--+--+--+--+
|`+|`-|`+|`-|`+|`-|`+|
+--+--+--+--+--+--+--+
|`||`3|`||`4|`||`5|`||
+--+--+--+--+--+--+--+
|`+|`-|`+|`-|`+|`-|`+|
+--+--+--+--+--+--+--+
```

The first four examples illustrate the same array pictured with the four combinations of the mode settings. The final example illustrates that if the result of *picture* is displayed at top level in diagram - decor mode, the implicit use of *picture* on the resulting character table causes it to be decorated.

Equations

```
write A = writescreen picture A
valence picture A = 2
```

place

Class: insertion operation

Usage: `B I place A   place (B I) A`

See Also: pick, placeall, deepplace, update

The operation *place* is the fundamental insertion operation for arrays. The normal use is that *I* is the address where *B* is to be placed in a modified version of *A*. The result is the modified array. If *I* is a solitary integer and *A* is a list, *I* is converted to an integer before being used. If *I* is not a valid address for *A*, the result is the fault *?address*.

```
     "abc 2 place 2 3 4 5 6
2 3 abc 5 6

     place (`_ 2) 'my work'
my_work

     A := 4 5 6 7
4 5 6 7

     A := 27 2 place A; A
4 5 27 7
```

```
     50 100 place A
?address
```

The fourth example shows the use of *place* in updating the value of a variable *A*. The last example illustrates an out of range use of *place*.

Nial provides a simpler notation for updating a variable. *A@I* (pronounced *A at I*) can be used on the left of assignment to update the item at location *I*. If the address is out of range a fault occurs.

```
     A@1 := 25
4 25 27 7

     A@5 := 7
?address

     A
4 25 27 7
```

The semantics of using the *at notation* on the left in assignment can be achieved using the operation *update*. The *at notation* and the *update* operation are usually more efficient than *place* because they avoid doing a copy of the right argument of *place*.

Equations

```
I in grid A ==> (I pick A) I place A = A
B \[I] deepplace A = B I place A
I in grid A ==>shape(B I place A) = shape A
```

placeall

Class: insertion operation

Usage: `B I placeall A   placeall (B I) A`

See Also: place, deepplace, updateall

The operation *placeall* generalizes the operation *place* to modify a collection of items of *A*. It returns a modified version of array *A* with the items at the addresses *I* replaced by the items of *B*. It corresponds to the sequential use of *place* with corresponding items from *B* and *I*. If an item of *I* is repeated, its last occurrence will determine the effect on *A*. If *B* has fewer items than *I*, its items are used cyclically (the first item is used after the last and the list repeated).

```
     ("abc "def) (2 3) placeall 2 3 4 5 6
2 3 abc def 6

     Text := 'my work is fun';
     placeall ('_' (`  findall Text)) Text
my_work_is_fun

     A := 4 5 6 7 ;
     A := (27 28) (3 2) placeall A; A
4 5 28 27
```

The last example shows the use of *placeall* to update the value of a variable *A*.

Q'Nial provides a simpler notation for updating a variable at several locations. The notation *A#I* (pronounced *A at all I*) can be used on the left of assignment to indicate the update of the items at the addresses given by *I*. If an item of *I* is out of range, the result is a fault but the variable is modified by the assignments done to locations that precede the invalid one.

```
     A#(0 1) := 25 26
25 26 27 7

     A#(2 100) := 50 51
?addresses

     A
25 26 50 7
```

The semantics of using the *at all* notation on the left in assignment can be achieved using the operation updateall.

The *at all* notation and the *updateall* operation are usually more efficient than *placeall* as they avoid doing a copy of the entire right argument of *placeall*.

Placeall is not the same as *(pack B I) EACHBOTH place A*. The latter results in an array of modifications of *A*, one for each item to be replaced.

Equations

```
I allin grid A ==> (I choose A) I placeall A = A
I allin grid A ==> shape (B I placeall A) = shape A
```

plus

Class: arithmetic operation

Property: binary pervasive

Usage: `A plus B  plus A B`

See Also: sum, minus, mod, opposite, times

The operation *plus* is the same as *sum* except that *plus* enforces the rule that it must be applied to a pair.

Sum is multi pervasive and can add up any number of items. The symbol + is a synonym for *sum*.

```
     7 plus 9
16

     (2 3 4) plus (12 22 33)
14 25 37

     plus 2 3 4
?plus expects a pair
```

positions

Class: picture operation

Usage: `positions A`

See Also: sketch, diagram, picture, display, paste, setcursor

The operation *positions* is used in conjunction with *picture* on singles, lists or tables. For such arrays, it returns an array of the same shape with the items being the addresses at which the pictures of the corresponding items of *A* appear in the picture of *A*.

```
     A gets 2 3 reshape "cat 'fish' 34.5 "dog 'meat' 26.5
+---+----+----+
|cat|fish|34.5|
+---+----+----+
|dog|meat|26.5|
+---+----+----+

     positions A
+---+---+----+
|1 1|1 5|1 10|
+---+---+----+
|3 1|3 5|3 10|
+---+---+----+
```

The operation *positions* is useful in applications that do interactive editing of data from its display on the screen.

post

Class: reshaping operation

Usage: `post A`

See Also: reshape, list, pass, cols

The operation *post* restructures an array *A* to be a table with one column holding the items of *A* in row major order.

```
     post 3 4 5
3
4
5

     post 'Now is the time'
          'for all good men'
          'to come to the aid'
          'of the party'
+------------------+
|Now is the time   |
+------------------+
|for all good men  |
+------------------+
|to come to the aid|
+------------------+
|of the party      |
+------------------+
```

The major purpose of post is to restructure data for display.

Definition

```
post IS OPERATION A { (tally A) 1 reshape A}
```

Equations

```
shape post A = (tally A) 1
post post A = post A
```

power

Class: arithmetic operation

Property: binary pervasive

Usage: `A power B   power A B`

See Also: binary pervasive, sqrt, divide, ln, exp

The operation *power* returns the result of raising the number *A* to the power *B*. The result is an integer if both *A* and *B* are boolean or integer and *B* is non-negative. If *B* is an integer, the power is done by multiplication. Otherwise it uses *exp (B * ln A)*.

```
     R gets 1 2 2.5 `a "abc ??error
1 2 2.5 a abc ?error

     R OUTER power R
     1        1        1.  ?A  ?A  ?error
     2        4   5.65685  ?A  ?A  ?error
   2.5     6.25   9.88212  ?A  ?A  ?error
?A       ?A       ?A       ?A  ?A  ?A
?A       ?A       ?A       ?A  ?A  ?A
?error   ?error   ?error    ?A  ?A  ?error

     2 power 3
8

     2 power -3
0.125

     16 power .5
4.
```

The examples show that *power* has different meanings depending on the right argument *B*. If *B* is a positive integer, the result is *A* multiplied by itself *B* times. If *B* is a negative integer, the result is the reciprocal of the result when *B* is positive. The last case illustrates that if *B* is 0.5, the result is a real number that is a square root of

Equations

```
isreal B ==> A power B = exp (B * ln A)
sqrt A = A power 0.5   (within roundoff error)
```

predicate

Class: operation property

A **predicate** is an operation that tests a condition and returns true if the condition holds and false otherwise. A predicate is not pervasive.

The predefined predicate operations include:

Operation	Test
allin	set-like inclusion
atomic	an array is an atom
diverse	all items differ
empty	an array has no items
equal	all items are equal
isboolean	arg is a boolean atom
ischar	arg is a character atom
isfault	arg is a fault symbol
isinteger	arg is a integer number
isphrase	arg is a symbol
isreal	arg is a real number
isstring	arg is a list of characters
like	two arrays contain same items
regexp_match	string matches a regular expression
simple	all items are atoms
unequal	items are not all equal
up	lexical ordering

prefix notation

Class: syntax

See Also: infix notation

An operation-expression can precede an array-expression. This is called a **prefix** use of the operation expression.

```
        + 7 5
12

     reshape [2 3,1 2 3 4 5 6]
1 2 3
4 5 6
```

prelattice of atoms

Class: concept

See Also: lte, up, max

The ordering sequence of the characters for sorting purposes is fixed for each Q'Nial version.

The binary and multi pervasive comparative operations of Nial organize the atoms of Nial in a **prelattice** or sorting sequence. The operation *lte* (or <=) does a less than or equal comparison between atoms. The numeric atoms are comparable, with a coercion being done if the arguments are of different numeric type. The *nadir*, represented by the fault *?O*, is less than or equal to all atoms. The *zenith*, *?I*, is greater than or equal to all atoms. Except for these two special cases, literal atoms are incomparable with atoms of different type. However, they can be compared within the same type using a character collating sequence that is version specific.

Phrases are compared lexicographically, such that *"apple* is after *"ape* but before *"apt*. Strings, being character lists, produce a list of results when compared.

Comparisons are of two forms: binary pervasive predicates which return boolean values and multi pervasive predicates that obtain the largest or smallest item in a list. If two atoms are incomparable, the predicates return *false*, whereas *max* returns the *zenith* and *min* returns the *nadir*.

These rules were chosen so that the following laws hold for all arrays *A* and *B*:

```
max A <= B = and (A EACHLEFT <= B)
A <= min B = and (A EACHRIGHT <= B)
```

See the entry for *up* for a discussion of the lexicographic ordering in Q'Nial.

product

Class: arithmetic operation

Properties: multi pervasive, reductive

Usage: `product A   * A   A product B   A * B`

See Also: times, divide, sum, plus, minus

The operation *product* multiplies the items of a simple array of numbers, reducing them to a single number that is their product. The type of the result is the highest type of the items. The operation is extended to non-simple arrays by the multi pervasive mechanism. The symbol * and *prod* are synonyms for *product*. The operation *times* is *product* restricted to use on pairs.

For an empty array, the result is 1.

```
     R gets 1 2 2.5 `a "abc ??error
1 2 2.5 a abc ?error

     R OUTER product R
     1       2      2.5  ?A  ?A  ?error
     2       4       5.  ?A  ?A  ?error
   2.5       5.     6.25  ?A  ?A  ?error
?A       ?A       ?A       ?A  ?A  ?A
?A       ?A       ?A       ?A  ?A  ?A
?error   ?error   ?error    ?A  ?A  ?error

     product 3. 45. 23. 18. 3.5
195615.

     product (3 4 5)(2 3 2)(0 2 4)(1 1 1)
0 24 40
```

The first example shows the result of *product* on all combinations of types of atoms. The last example shows that pervasive extension of *product* multiples a list of triples in an item by item fashion.

Equations

```
A product B = B product B
product single A = EACH(product single) A
product EACH product A =f= product link A
shape cart A = product EACH tally A
product shape A = tally A
```

profile

Class: profiling operation

Usage: `profile Fnm`

See Also: setprofile, clearprofile, profiletable, profiletree, profiling

The operation *profile* is used to convert the profiling statistics into a displayable form. If *Fnm* is a file name, as a string or a phrase, then the output is written to the file. If it is the empty string, then the profiling

information is displayed on the screen. If logging is on, it will be copied to the log file. The gathering of profile statistics must be turned off when *profile* is called. A detailed example of the output is given in the help entry on profiling.

Examples of use:

```
     profile "myprofl

     profile ''
```

profiletable

Class: profiling expression

Usage: `Profiletable`

See Also: profile, setprofile, clearprofile, profiling

The expression *Profiletable* is used to summarize the internal data structures that are used in the gathering of profiling statistics to produce the same information that is displayed by the operation *profile*. It produces a list of entries, one for each definition that has been encountered during execution with profiling on. Each entry includes the name, the number of direct calls, the number of recursive calls, the execution time, and a list of subentries for the definitions directly called by the definition. The subentries summarize the information for the called definitions in the same format except that no further breakdown is given on definitions they call.

A detailed example of the output of *Profiletable* is given in the help entry on profiling.

```
     Profiletable
+-----------------+-------------------------------------+---
|+----+--+-+----++|+-----+--+-+----+------------------+|+--
||test|12|0|4.12||||tryit|11|0|4.12|+-----------------+|||ag
|+----+--+-+----++||     |  | |    ||+----+--+-+----++||||
|                 ||     |  | |    |||test|11|0|4.12||||||
|                 ||     |  | |    ||+----+--+-+----++||||
|                 ||     |  | |    |+-----------------+|||
|                 |+-----+--+-+----+------------------+|+--
+-----------------+-------------------------------------+---

-----------------------------------+
---+-+-+----+--------------------+|
ain|1|0|3.96|+------------------+||
   | | |    ||+-----+--+-+----++|||
   | | |    |||tryit|10|0|3.96|||||
   | | |    ||+-----+--+-+----++|||
   | | |    |+------------------+||
---+-+-+----+--------------------+|
-----------------------------------+
```

profiletree

Class: profiling expression

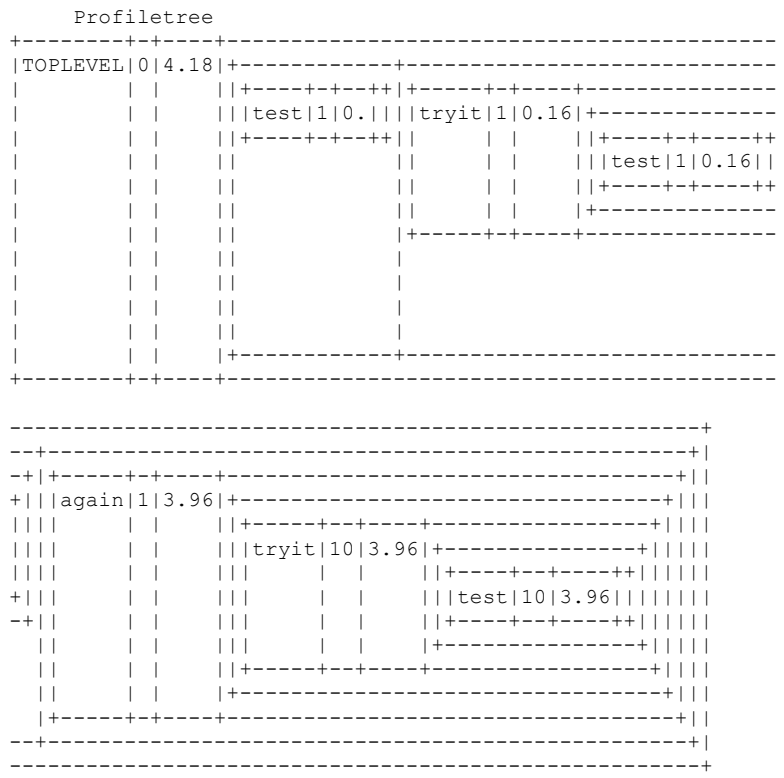
Usage: `Profiletree`

See Also: profile, setprofile, clearprofile, profiletable, profiling

The expression *Profiletree* is used to display the detailed information from the internal data structures that are used in the gathering of profiling statistics. It produces a nested list of entries showing the calling dynamic calling sequence from top level, with the amount of time and number of calls at each level. Each entry includes the name, the number of direct calls, the execution time, and a list of subentries for the definitions directly called by the definition. The subentries provide the same information for the called definitions in the same format including entries on all the definitions that they call in turn.

A detailed example of the output of *Profiletable* is given in the help entry on profiling.

Example of use:



profiling

Class: concept

See Also: setprofile, profile

Q'Nial Profiling Facility

Q'Nial has a profiling capability that is used to gather relative execution times for operations, transformers and defined expressions written in Nial. The profiling capability has the following features:

Command	Purpose
setprofile A	turns on or off the internal routines that collect the statistics
profile Fnm	that displays the profile data to the screen or writes it to a file
Clearprofile	clears the current profile information and reinitializes the profiling system
Profiletable	provides the profile information as a table
Profiletree	provides the detailed profile information in terms of the call tree

The operation *setprofile* is called with argument *l* to turn on the collection of data for a profiling session. It is called with argument *o* to suspend gathering statistics for a session. Both calls to *setprofile* should be in the same scope, and cannot be nested. Only one profiling session can be underway at a time.

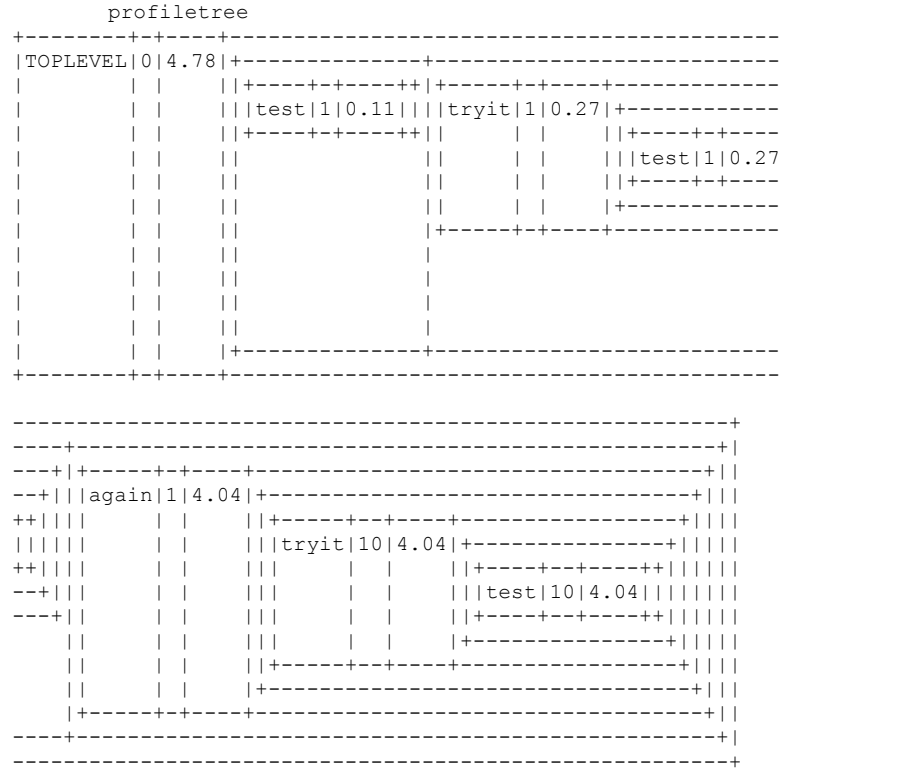
The operation *profile* is called with a text argument (string or phrase) that is used as the file name for the profile information. If an empty string is the argument, the output is sent directly to the screen. It computes the statistics on the profiling session underway, prepares the output, and writes it to the file.

The expression *Clearprofile* clears the two internal data structures that are built as execution is profiled so that a new profiling session can be started.

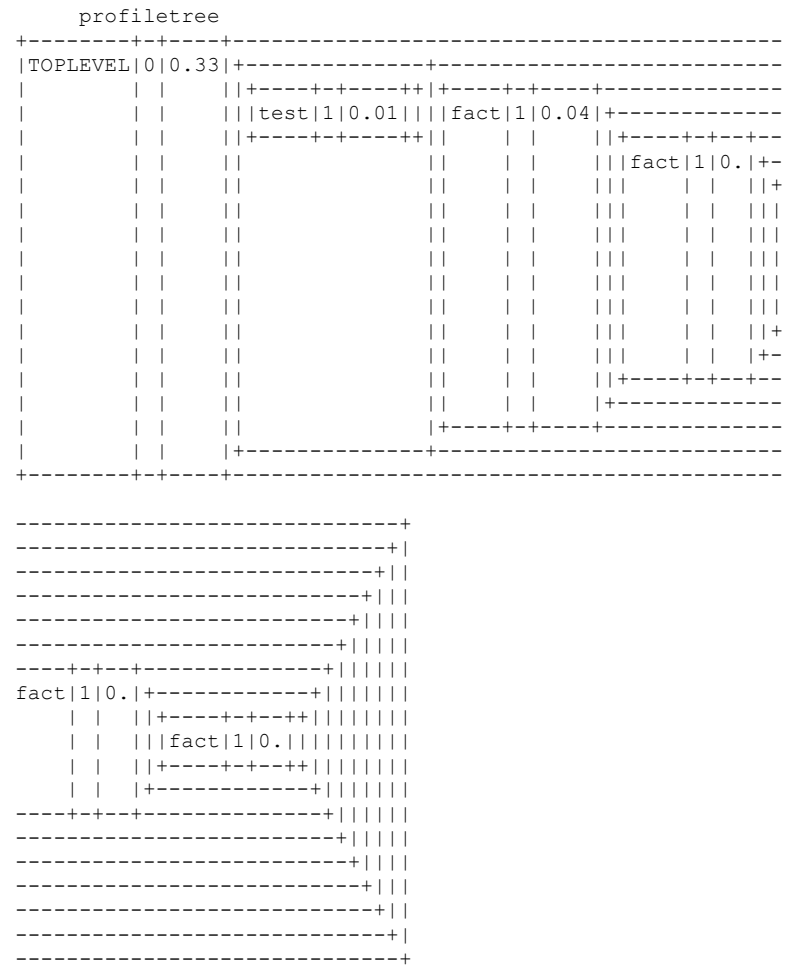
The expression *Profiletree* provides the summary data that is displayed in the output produced by *profile*, but stores it in a Nial array.

The expression *Profiletree* provides a dynamic call tree of the execution with related times. This provides a more detailed breakdown of the profile data. The following session illustrates the use of the profiling capability:

```
     setwidth 60;

     test is op n {
       z := random n n;
       max abs (z - inv inv z) }

     tryit is  op N { test N }

     again is op M N { sum EACH tryit (M reshape N) }

     fact is op n {
       IF n<=1 THEN 1 ELSE n * fact (n - 1) ENDIF }

# first profiling session

     setprofile 1
o

     test 20
3.86843e-15

     tryit 50
2.83107e-14

     again 10 75
6.72483e-13

     setprofile o
1

     profile ''

Total execution time of profile session:     4.780000
Total execution time in top level calls:     4.420000

op name[.tr arg]                calls[rec]    time time/call  % time
test.........................   12            4.42   0.3683    100.0<

tryit........................   11            4.31   0.3918     97.5<
 test........................   11            4.31   0.3918   100.00

again........................    1            4.04   4.0400     91.4<
 tryit.......................   10            4.04   0.4040   100.00
```




```
# second profiling session

     clearprofile

     setprofile l
o

     test 30
1.72085e-14

     fact 4
24

     setprofile o
l

     profile ''

Total execution time of profile session:     0.330000
Total execution time in top level calls:     0.050000

op name[.tr arg]                calls[rec]    time time/call  % time
test.........................    1           0.01   0.0100     20.0<

fact.........................    1[    3]    0.04   0.0400     80.0<
```




```
 % end of session
```

The profile statistics show the division of time between the various definitions. For each definition called during the profiling session there is a summary of the number of calls and time used. For each definition there is also a breakdown on the time spent in other definitions that were directly called from that definition.

Entries in the profile statistics that have a "<" to the right of the last value are top level calls. All other entries have been called by these top level calls.

The time spent in direct recursive calls is counted in the original call. The number of such calls is placed in brackets after the number of top level calls.

The statistics do not account for local definitions within global ones; their execution time is simply added to the time for the global definition.

A feature of the profiler not shown in the above example is that when timing a transformer definition, the time spent executing the argument operation(s) is computed in order to understand how much of the cost of the transformer is due to applying the argument operation(s).

The accuracy of the timing information is limited by the precision of the information available through system calls to the host system. For very small definitions the statistics may show no execution time.

Usually their effect can be estimated by considering the execution time of the calling definition.

The *Profiletable* result provides the summarized profile statistics as a quintuple consisting of the name of the definition, the number of calls, the number of recursive calls, the time, and a list of records for each definition it directly calls. Each of the latter records has the same information provided for the routine itself, but does not report on the definitions it itself calls.

The *Profiletree* result starts with a node indicating the toplevel and the total execution time. It has a list of subnodes for each definition called from the top level. Each of these has the number of calls, the execution time and nodes for each definition it calls. A recursive call is reported directly in a subnode and hence a deeply recursive routine will produce a deeply nested array of profile information.

program fragment

Class: concept

See Also: expression

Nial is a programming language specifically designed for use in an interactive environment. The formal description of the language describes the valid language constructs that can be entered in one interaction; and explains the meaning of one such entry in terms of the environment created by earlier interactions in the same session. The term **program fragment** is used to describe a meaningful piece of program text.

The rules for writing well formed program fragments in Nial are called the *syntax rules* of Nial. A program fragment that is well formed is said to be syntactically correct. The syntax rules are analogous to the rules of grammar that determine the correctness of English.

putfile

Class: file operation

Usage: `putfile Filename A`

See Also: getfile, appendfile

The operation *putfile* writes the rows of items of *A* as text records to the file with name *Filename*. It is used in conjunction with *getfile*. The file must not be open. Items of *A* are arrays of characters of any valence.

In the following example, three strings would be written to the file Maillist.

```
     putfile "Maillist ['Now is the time' 'for all good
men','to come to the aid']
```

Definition

```
putfile IS OPERATION File A {
   IF not isfault (Filenum := open File "w) THEN
      ITERATE (Filenum writefile) (link EACH rows A) ;
      close Filenum ;
   ELSE
      Filenum
   ENDIF }
```

quiet_fault

Class: conversion operation

Usage: `quiet_fault A`

See Also: fault

The operation *quiet_fault* is a version of operation *fault* that turns off fault triggering before producing the fault and restores the triggering switch to its prior value after producing the fault.

Definition

```
quiet_fault IS OPERATION Str {
   Oldsetting := settrigger o;
   Res := fault Str;
   settrigger Oldsetting;
   Res }
```

quotient

Class: arithmetic operation

Property: binary pervasive

Usage: `A quotient B   quotient A B`

See Also: mod, floor, divide

The operation *quotient* returns the quotient of dividing integer *A* by integer *B*. If the divisor *B* is zero, the result is zero. If it is negative, the result is the fault *?negative divisor*. The operation *quotient* converts boolean arguments to integer but otherwise produces a fault if either argument is not an integer.

```
     R gets 1 2 2.5 `a "abc ??error;
     R OUTER quotient R
     1       0  ?A      ?A  ?A  ?error
     2       1  ?A      ?A  ?A  ?error
?A       ?A      ?A      ?A  ?A  ?error
?A       ?A      ?A      ?A  ?A  ?A
?A       ?A      ?A      ?A  ?A  ?A
?error   ?error  ?error  ?A  ?A  ?error

     (5 quotient 2) (-5 quotient 2)
2 -3

     (5 quotient -2) (5 quotient 0)
?negative divisor 0
```

The first example illustrates all combinations of atom types for the two arguments to *quotient*.

The quotient on division by a positive integer *B* is always an integer on the same side of the origin as *A*. The result of *A quotient 0* is 0 rather than a fault since this rule is compatible with the choice that *A mod 0* is *A*. The operation *quotient* is used in place of *divide* in situations where the result must be an integer.

Equation

```
isinteger A ==> A quotient 0 = 0
floor (A - (B*(A quotient B))) =f= A mod B
A quotient B =f=  floor (A / B)
```

raise

Class: nesting restructuring operation

Property: binary

Usage: `N raise A   raise N A`

See Also: split, rows, cols, mix, rank

The operation *raise* is used to partition an array *A* along its axes by indicating that the first *N* axes are to be retained in the result. The remaining axes become axes of the items of the result. *N* must be an integer in the range from 0 to *valence A*. The result is an array of shape given by taking the first *N* items of *shape A*. The items of the result have the shape given by dropping the first *N* items of *shape A*. Thus, the *2 raise* of an array of shape 3 4 2 is a 3 by 4 table of pairs. The *1 raise* of the same array is a triple of 4 by 2 tables.

```
     A := 3 4 2 reshape 'ABCDEFGHIJKLMNOPQRSTUVWX'
AB  IJ  QR
CD  KL  ST
EF  MN  UV
GH  OP  WX

     2 raise A
+--+--+--+--+
|AB|CD|EF|GH|
+--+--+--+--+
|IJ|KL|MN|OP|
+--+--+--+--+
|QR|ST|UV|WX|
+--+--+--+--+

     1 raise A
+--+--+--+
|AB|IJ|QR|
|CD|KL|ST|
|EF|MN|UV|
|GH|OP|WX|
+--+--+--+
```

Definition

```
raise IS OPERATION A { N drop axes A split A }
```

Equations

```
N a nonnegative integer <= valence A and not empty A ==>
                          shape (N raise A) = N take shape A
shape first (N raise A) = N drop shape A ==> mix (N raise A) = A
```

random

Class: array generation operation

Usage: `random S`

See Also: seed

The operation *random* is used to generate pseudo-random real numbers in the range from 0. to 1. The argument *S* is a shape and *random* generates *product S* numbers storing them in an array of shape *S*. *Random* uses a congruential method with period 2147483647. The random generator generates a sequence of integers using the formula: *R := 16807 * R mod 214783647 starting with R = 314159262. The result is the real R/2147483647. The sequence can be initialized using the operation seed* with an argument *N*, where *0. < N < 1.0*.

Seed resets the random number generator to use R = floor (R1 * 214783647.).

```
     seed 0.314159; random 5
0.070 0.743 0.240 0.627 0.309

     floor ( 100. * random 2 5 )
98 19 94 47 36
88 44 47 82 38
```

The first example shows the five random numbers generated after a seed of 0.314159 is used. The second example illustrates how to use the operation *random* to generate 10 random integers between 0 and 99 and store them in an array of shape 2 5.

rank

Class: apply-by-partition transformer

Usage: `N RANK f A`

See Also: byrows, bycols, partition, raise

The transformer *RANK* applies an operation *f* to arrays formed from the last *N* axes of *A*. The results of the applications of *f* must all be the same shape and are combined to form a result array using *mix*.

```
     2 RANK reverse (3 4 5 reshape count 60)
20 19 18 17 16   40 39 38 37 36   60 59 58 57 56
15 14 13 12 11   35 34 33 32 31   55 54 53 52 51
10  9  8  7  6   30 29 28 27 26   50 49 48 47 46
 5  4  3  2  1   25 24 23 22 21   45 44 43 42 41

     1 RANK sum (3 4 reshape count 12)
10 26 42
```

The first example reverses the planes of the generated array. The second example sums the rows of the generated matrix.

Definition

```
RANK IS TRANSFORMER f OPERATION N A {
   mix EACH f (N lower A) }
```

reach

Class: selection operation

Property: binary

Usage: `P reach A   reach P A`

See Also: pick, choose, deepplace

The operation *reach* is used to select an array at an arbitrary path *P* within the nested structure of *A*. *P* is a list of addresses where the first item of *P* selects an item of *A*, the second selects an item of that item, etc. If *P* is empty, the result is *A*. The operation is implemented as an iterated *pick* and hence the addresses in the path are converted using first if they are solitary integers. If one of the picks attempts a selection out of bounds, the result is fault *?path*.

```
     A := 3 4 reshape (pack 'ABCDEFGHIJKL' '1234567890ab')
+--+--+--+--+
|A1|B2|C3|D4|
+--+--+--+--+
|E5|F6|G7|H8|
+--+--+--+--+
|I9|J0|Ka|Lb|
+--+--+--+--+

     (2 0) 1 reach A
9
```

In the example, the path (2 0) 1 picks the item with address 1 of the location with address 2 0.

Q'Nial supports a direct notation for doing a *reach* into an array associated with a variable: *Var@@I* (pronounced *Var at path I*). The *at path* notation can be used on both the left and right of an assignment expression.

Definition

```
reach IS P A {
   IF empty P THEN
      A
   ELSE
      rest P reach (first P pick A)
   ENDIF }
```

Equations

```
P reach A = (EACH list P) reach A
items of P are valid addresses where used ==> (P reach A) P deepplace A = A
```

read

Class: interactive input/output operation

Usage: `read P`

See Also: readscreen, write, writescreen, execute

The operation *read* is used in an interaction with the user running a program to obtain an array value from the keyboard after issuing a prompt given by the string or character *P*. After receiving the prompt, the user types a sequence of characters and then presses the *Return* key. The sequence of characters, excluding the *Return*, becomes the string entered. The string is executed to give the result of *read*. If the string being executed contains an integer representation that would convert to a number outside the range of integers, the corresponding real number is returned.

```
     A := read 'Type an expression : ';
Type an expression : 3 + count 10

     A
4 5 6 7 8 9 10 11 12 13
```

The operation *read* is similar to operation *readscreen*. The difference is that in *read* the string received from the keyboard is evaluated, whereas in *readscreen*, the string is returned.

Definition

```
     read IS OPERATION P { execute readscreen P }
```

readarray

Class: nial direct access file operation

Usage: `readarray Fnum N`

See Also: readrecord, writearray, filetally, eraserecord, open, close

The operation *readarray* is used to read the component or components indicated by *N* from the direct access file with file port *Fnum*. The file must have been created using *writearray* and must have been opened as a direct file using *open*.

If *N* is an integer, the result is the array stored at record position *N* of the file.

If *N* is a list of integers, the result is a list of the corresponding records. If there is no record at position *N*, the result is the fault *?missing*. If *N* is greater than *filetally A*, the result is the fault *?eof*.

```
     readarray Fnum [23,24,25];
```

If *N* is a solitary integer, the result is a solitary holding the selected item.

readchar

Class: interactive input/output expression

Usage: `Readchar`

See Also: read, readscreen, writechars, write, editwindow

The expression *Readchar* is used to obtain the result of a single keystroke on the keyboard. The character is not displayed on the screen automatically. If the keystroke corresponds to a single displayable ASCII character, the result is that character. Otherwise, the result is a phrase indicating the function key pressed.

Q'Nial supports all the standard keys of the IBMPC keyboard. On a non-IBMPC version of Q'Nial, these keys are available through use of escape sequences.

The input from *Readchar* can be displayed by the use of *writechars*.

```
     writechars Readchar
Backspace
```

In the example, the user responded by pressing the backspace key.

Readchar is used when developing an interactive program in which the user uses keystrokes to indicate requested actions. The decision whether or not to display a character on the screen can be made by the program rather than done automatically.

The following table lists the non-character keystrokes recognized on the IBMPC keypad and gives the default sequences accepted by an ASCII terminal when the corresponding keystroke is not present. The two keyboard layouts on SUN workstations behave like an IBMPC keyboard.

IBMPC

IBMPC Keystroke	Sun Console	Ascii Terminal
Return	Return	Return
Backspace	Backspace	Backspace
Del	Del	Del
Ins	Esc Esc i	Esc Esc i
Up_Arrow	Up_Arrow	Up_Arrow
Down_Arrow	Down_Arrow	Down_Arrow
Right_Arrow	Right_Arrow	Right_Arrow
Left_Arrow	Left_Arrow	Left_Arrow
Home	Home	Esc Esc h
End	End	Esc Esc e
PgUp	PgUp	Esc Esc u
PgDn	PgDn	Esc Esc d
Ctrl Left_Arrow	Esc Left_Arrow	Esc Esc c l
Ctrl Right_Arrow	Esc Right_Arrow	Esc Esc c r
Ctrl PgUp	Esc PgUp	Esc Esc c u
Ctrl PgDn	Esc PgDn	Esc Esc c d
Ctrl Home	Esc Home	Esc Esc c h
Ctrl End	Esc End	Esc Esc c e
Tab	Tab	Tab
Shift Tab	Esc Tab	Esc Tab
Esc	Esc Esc Esc	Esc Esc Esc
F1 to F10	F1 to F10	Esc 1 to Esc 0
<Alt F1> to <Alt F10>	Esc F1 to Esc F12	Esc Esc 1 to Esc Esc 0
<Alt A> to <Alt Z>	<Esc a> to <Esc z>	<Esc a> to <Esc z>

The IBMPC keyboard support also recognizes the function keys modified by Shift and Ctrl and <Alt 0> to <Alt 9> but these are not mapped onto corresponding escape sequences in the Unix version.

The keyboard mappings are set by default but may be controlled by description files stored in the *initial* directory.

readcursor

Class: window mode input/output expression

Usage: `Readcursor`

See Also: setcursor, screensize, setwindow

The expression *Readcursor* returns a pair of integers indicating the row and column position of the cursor in the active window.

It is used in developing an interactive application in which the cursor must be returned to an expected position after some task such as displaying a pop up window.

readfield

Class: host direct access file operation

Usage: `readfield Filename Start Size`

See Also: readrecord, readarray, filelength

The operation *readfield* is used to read a portion of an existing host file as a character string. The string or phrase *Filename* is the name of the file, the integer *Start* is the offset to the beginning of the field to be read and integer *Size* is the size of the field to be read. The result is character string of the items read. If the logical field extends across a line boundary the field will include the newline indicator appropriate for the host system (either linefeed or return-linefeed).

Example

```
readfield "Myfile 80 20
```

The result is the string beginning at offset 80 in file *Myfile* of length 20.

readfile

Class: file operation

Usage: `readfile Fnum  readfile Fnum N`

See Also: writefile, getfile, open, close, readscreen

The operation *readfile* is used to read the next record from the sequential text file designated by file port *Fnum*. The file port *Fnum* is the integer returned by *open*, which must have been called before executing the *readfile*.

A record of text in a text file is a sequence of characters up to but excluding an end-of-line indication. The end-of-line indication is system dependent and may be one or more characters. The interface between Q'Nial and the host system recognizes end-of-line and, on successive readfile requests, returns the records of the text file with the end-of-line indications removed.

If *readfile* is used with a second argument *N*, it reads the next *N* bytes of the text file and returns them as a string. In this usage, end-of-line characters are processed as any other character and are not removed. This second form is intended for reading information from device drivers or a communications port, where end-of-line indications may not be given.

The following example shows a text file, Myfile being opened and its records being gathered as a list of strings. The example has the same effect as the expression *getfile "Myfile*.

```
File_number := open "Myfile "r ;
Records := '' ;
Record := readfile File_number ;
WHILE Record ~= ??eof DO
     Records gets Records append Record ;
     Record := readfile File_number ;
ENDWHILE ;
close File_number ;
```

readrecord

Class: nial direct access file operation

Usage: `readrecord Fnum N`

See Also: readarray, writerecord, filetally, eraserecord, open, close

The operation *readrecord* is used to read the component or components indicated by *N* from the direct access file with file port *Fnum*. If *N* is an integer, the result is the string stored at component position *N* of the file. The file must have been opened as a direct file and must have been created using *writerecord*.

If *N* is a list of integers, the result is a list of the corresponding records. If there is no component at *N*, the result is the fault *?missing*. If *N* is greater than *filetally A*, the result is the fault *?eof*.

A direct access file is stored as two host system files: a *.ndx* file holding index information and a *.rec* file holding the records.

```
readrecord Fnum [23,24,25];

convert_records IS OP Filein Fileout {
   Fin := open Filein "d ;
   Fout := open Fileout "d ;
   Num := filetally Fin ;
   writerecord Fout (tell Num)
   (EACH convert_op (readrecord Fin (tell Num)));
   EACH close Fin Fout ; }
```

The first example shows a call of *readrecord* that returns three records. The second example shows an operation *convert_records* that reads all the records in direct file *Filein*, modifies them by *convert_op* and writes them to direct file *Fileout*. For a very large file, it may be necessary to replace the use of *EACH* with a loop.

The *readrecord* operation can access fields of a fixed format file created external to Q'Nial. If the component number used in a *readrecord* is beyond the end of file as indicated by the *filetally Fnum*, stored in the *.ndx* file, the *.rec* file is checked to see if the host file information indicates that the *.rec* file extends beyond the length expected for a file created by the Q'Nial operations. If so, the *.rec* file is assumed to consist of a sequence of blocks of records, with each block having fields of the sizes indicated by the *.ndx* file. The index information for the record at *(N mod filetally Fnum)* is used to find the requested component within the block of records found from *(N quotient filetally Fnum)*.

To use this feature, a separate process must be followed to create the *.ndx* file corresponding to one block of records in the external file. This is done by writing a sequence of strings of the lengths of the records to a dummy direct access file and then renaming the resulting index file and the external file to be names expected by the Q'Nial direct access facility.

readscreen

Class: interactive input/output operation

Usage: `readscreen P`

See Also: read, readchar, readfile, writescreen

The operation *readscreen* is used to obtain a string from the keyboard after issuing the prompt *P*. The prompt *P* must be a character or a string. After receiving the prompt, the user types a sequence of characters and then presses the *Return* key. The sequence of characters, excluding the *Return*, is the string returned as the result of *readscreen*.

```
     Name := readscreen 'Your name? : ' ;
Your name? : Mike

     Name
Mike

     Answer := first 'Do you wish to continue? y/n: ' ;
     IF Answer in 'yY' THEN ...
```

The operation *readscreen* is similar to the operation *read*, the difference being that in *read*, the string received from the keyboard is evaluated, whereas with *readscreen*, the string is returned.

reciprocal

Class: arithmetic operation

Property: unary pervasive

Usage: `reciprocal A  recip A`

See Also: divide, quotient, opposite

The operation *reciprocal* or *recip* for short, returns a real number which is the reciprocal value of a numeric atom.

Reciprocal produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	reciprocal of the corresponding real
integer	reciprocal of the corresponding real
real	1.0 / A if A ~= 0.; ?div if A = 0.
character	fault ?A
phrase	fault ?A
fault	argument A

```
     reciprocal 1 -2 3.5 `a "abc ??error
1. -0.5 0.285714 ?A ?A ?error

     reciprocal -2 (3 -4.5)
+----+------------------+
|-0.5|0.333333 -0.222222|
+----+------------------+
```

Definition

```
     reciprocal IS OPERATION A (1 divide A)
```

Equations

```
   A divide B = A * reciprocal B
   reciprocal reciprocal reciprocal A = reciprocal A   (within roundoff error)
```

recover

Class: user defined operation

Usage: `recover Msg`

See Also: toplevel, latent, checkpoint, restart

The operation *recover* is prepared by the user of Q'Nial to handle the action to be taken on detection of <*Ctrl c*> or any other situation that forces a return to top level. If a *recover* operation does not exist in the workspace, Q'Nial returns to the top level loop on a <*Ctrl c*>. This operation is necessary to provide control over **closed** applications which do not permit the user to access Q'Nial at the top level loop. On its call the argument provided to the operation will be a character string giving the nature of the interruption, for example *'user interrupt'* if a Ctrl-C has occurred or *'programmed interrupt'* if *toplevel* has been called.

```
     recover IS OP Msg { Continue; }
```

```
recover IS Op Msg {
   IF Msg = 'user interrupt' THEN
      IF first readscreen 'Do you wish to restart? y or n : ' in 'yY' THEN
         Restart
      ELSE
         Continue
      ENDIF
   ELSE
      Continue
   ENDIF }
```

In the first example, if <*Ctrl c*> is pressed or some other action forces a jump to top level, the session ends saving the workspace in *continue.nws*. The message parameter is ignored.

In the second example of a definition for *recover*, a jump to top level by a <*Ctrl-C*> interrupt results in the display of the prompt:

```
Do you wish to restart? y or n :
```

If the response is text that begins with y or Y, the expression *Restart* is executed. If the session had begun with a workspace named on the command line, the workspace would be reloaded and its *Latent* expression, if any, evaluated. If the response begins with something other than y or Y, or the interrupt was of some other kind then the current session ends by executing *Continue*.

Recur

Class: recursion transformer

Usage: `RECUR [test, endf, parta, joinf, partb] A`

See Also: across, down

RECUR is a general recursion transformer with five operation arguments: *test* checks that the argument meets an end condition, *endf* is applied to the end argument before starting to build the result, *parta* computes the left value from the argument, which is stacked, *joinf* combines the left and right values as the recursion unwinds, and *partb* gives the value to be recurred on to produce the right value.The recursion terminates provided the repeated application of the operation *partb* results in an array that satisfies *test*.

```
     RECUR [ 0 =, 1 first, pass, product, -1 +] 4
24
     RECUR [ empty, 0 first, first, plus, rest ] 3 4 5
12
```

Definition

```
RECUR is TR test endf parta joinf partb OP A {
   Elements := Null;
   WHILE not test A DO
     Elements := Elements append parta A;
     A := partb A;
   ENDWHILE;
   Res := endf A;
   FOR E WITH reverse Elements DO
     Res := E joinf Res;
   ENDFOR;
   Res }
```

Equations

```
   RECUR [test, endf, parta, joinf, partb] A = FORK [test, endf, joinf [parta, RECUR
[test, endf, parta, joinf, partb] partb]] A
   RECUR [ 0 =, 1 first, pass, product, -1 +] N = factorial N
   RECUR [ empty, 0 first, first, plus, rest ] A = sum A
```

reduce

Class: reduction transformer

Usage: `REDUCE f A`

See Also: accumulate, reductive

The transformer *REDUCE* transforms an operation *f* into an operation which, when applied to an array *A*, has the same effect as evaluating an expression in which *f* is placed between the items of the array, with grouping done in a right-to-left manner. If *f* is an operation that maps a pair of atoms to an atom, *REDUCE f* reduces a simple array to an atom. For the built-in reductive operations: *sum*, *product*, *and*, *or*, *max*, *min*, and *link*, *REDUCE f* is the same as *f*.

```
     REDUCE divide 3 4 5 6
.625

     3 divide (4 divide ( 5 divide 6 ) )
.625

     REDUCE plus (count 10)
55

     REDUCE hitch 'abcde'
abcde

     REDUCE append 'abcde'
+-+----------+
|a|+-+------+|
| ||b|+-+--+||
| || ||c|de|||
| || |+-+--+||
| |+-+------+|
+-+----------+

     REDUCE pass 73 45 39 97
+--+----------+
|73|+--+-----+|
|  ||45|39 97||
|  |+--+-----+|
+--+----------+
```

Definition

```
REDUCE IS TRANSFORMER f OPERATION A {
   % if f is reductive, apply f directly;
   IF empty A THEN
      Res := ??identity;
   ELSE
      Res := last A;
      FOR B WITH reverse front A DO
         Res := B f Res;
      ENDFOR;
   ENDIF;
   Res }
```

Equations

```
REDUCE f solitary A = A
REDUCE f single A = A
REDUCE f A B C = A f (B f C)
```

reducecols

Class: apply-by-partition transformer

Usage: `REDUCECOLS f A`

See Also: reduce, reducerows, bycols

The transformer *REDUCECOLS* does the reduction of the columns of *A* with the operation *f*.

```
    REDUCECOLS sum (5 6 reshape count 30)
65 70 75 80 85 90
```

The example returns the sum of the columns of the generated table.

Definition

```
REDUCECOLS IS TRANSFORMER f OPERATION A {
   [valence A - 2 max 0,axes first A] PARTITION REDUCE f A }
```

reducerows

Class: apply-by-partition transformer

Usage: `REDUCEROWS f A`

See Also: reduce, reducecols, byrows

The transformer *REDUCEROWS* does the reduction of the rows of *A* with the operation *f*.

```
    REDUCEROWS product (3 4 reshape count 12)
24 1680 11880
```

The example returns the product of the rows of the generated table.

Definition

```
REDUCEROWS IS TRANSFORMER f OPERATION A {
   BYROWS (REDUCE f) A }
```

reductive

Class: operation property

See Also: multi pervasive, reduce

A **reductive** operation is one that extends a functional capability normally defined on a pair to a list, reducing the result by pairwise application of the function.

The predefined reductive operations include:

Operation	Function
and	logical "and" of a boolean array
link	the list of all the items of the items in the array
max	highest item in the array
min	lowest item in the array
or	logical "or" of a boolean array
product	arithmetic product of an array of numbers
sum	arithmetic sum of an array of numbers

All of the above operations except *link* are also multi pervasive. The transformer *REDUCE* can be used to produce a reductive operation from a binary one.

refresh

Class: window mode input/output expression

Usage: `Refresh`

See Also: putscreendata, clearwindow, setwindow, host

Q'Nial keeps a copy of the values and attributes of the characters displayed on the screen in its memory. The expression *Refresh* restores the screen to the display corresponding to the screen memory values and attributes. It is used after a *host* operation has overwritten the screen display to restore the screen to its previous display.

```
host 'dir' ;
Readchar ;
Refresh ;
```

When Q'Nial is in window mode, the screen management output routines update an internal image of the screen as characters are written to the screen but the screen itself is updated only occasionally. Sometimes

it is necessary to insert a *Refresh* expression to have the screen updated at the time desired.

regexp

Class: string manipulation operation

Usage: `R regexp S   regexp R S`

See Also: regular expression notation, regexp_match, regexp_substitute

The operation *regexp* is used to find the first substring in a string *S* that matches a regular expression pattern given by *R*. The pattern can contain subgroups indicated by parentheses. If there are no subgroups in *R* then the result is a pair consisting of a boolean indicating if the search succeeded and the first substring in *S* matched by *R*. Otherwise it is a longer list with the additional items being the substrings that match the subgroups in *R*.

```
     'def' regexp 'abcdefghi'
+-+---+
|l|def|
+-+---+

     'deg' regexp 'abcdefghi'
+-++
|o||
+-++

     '^(.)[bg](..)[eh]' regexp 'abcdefghi'
+-+-----+-+--+
|l|abcde|a|cd|
+-+-----+-+--+
```

In the first example, the string *'def'* matches a substring of the right argument. In the second example the string *'deg'* does not match. In the third example, the regular expression pattern searches for a string anchored at the beginning (by using ^) and consisting of any character followed by a "b" or a "g", followed by any two characters followed by a "c" or a "d". The result finds a match and picks out the two substrings corresponding to the two subgroups.

regexp_match

Class: string manipulation operation

Property: predicate

Usage: `regexp_match R S [O]`

See Also: regular expression notation, regexp, regexp_substitute

The operation *regexp_match* is used to test whether there is a substring in a string *S* that matches a regular expression pattern given by *R*. It returns *true* if a match is found and *false* otherwise. If the optional argument *O* is *"i* then the search is case insensitive.

```
     regexp_match 'def' 'abcdefghi'
l

     regexp_match 'deg' 'abcdefghi'
o

     regexp_match '^.[bg]..[eh]' 'abcdefghi'
l

     regexp_match '^(.)[bg](..)[eh]' 'abcdefghi'
l

     regexp_match 'DEF' 'abcdefghi' "i
l
```

In the first example, the string *'def'* matches a substring of the right argument. In the second example the string *'deg'* does not match. In the third example, the regular expression pattern searches for a string anchored at the beginning (by using ^) and consisting of any character followed by a "b" or a "g", followed by any two characters followed by a "c" or a "d". A match is found. Example four shows that subgroups are allowed in the pattern and do not affect the match. The final example shows a case insensitive search.

regexp_substitute

Class: string manipulation operation

Usage: `regexp_substitute R S T [O]`

See Also: regular expression notation, regexp, regexp_match

The operation *regexp_substitute* is used to replace one or more substrings in string *T* that match the regular expression pattern *R* with string *S*. If the optional argument is *"i* then the search is case insensitive. If it is *"g* then all substitutions are done.

```
     regexp_substitute 'def' 'XX' 'abcdefghi'
abcXXghi

     regexp_substitute 'xyz' '456' 'abcdefghi'
abcdefghi

     regexp_substitute 'DEF' 'YY' 'abcdefghi' "i
abcYYghi

     regexp_substitute 'a' '3' 'all able apes pay attention' "g
3ll 3ble 3pes p3y 3ttention

     regexp_substitute 'A' '3' 'all able apes pay attention' "ig
3ll 3ble 3pes p3y 3ttention
```

In the first example, the string *'def'* is replaced by *'XX'*. In the second example the string *'deg'* does not match and *T* is returned. In the third example, the regular expression pattern matches with a case insensitive search and the substitution is done. Examples four and five show multiple substitutions first with a case sensitive pattern and then with a case insensitive one.

registerdllfun

Class: system operation

Usage: `registerdllfun Nm Dllnm Path Restype Argtypes   registerdllfun Nm`

See Also: calldllfun, dlllist, callc

A Prototype DLL calling interface has been added to the Windows versions of Q'Nial (Console, GUI and DLL version). This ability allows NIAL code to call external routines in DLL libraries. These libraries can be user written, part of the Operating System, or 3rd party DLLs.

The DLL interface routines manage an internal table that keeps track of the currently installed DLLs and their argument types and result type. This table also tracks if the actual library has been loaded (by a call to *calldllfun*). The table is preserved as part of the workspace, and once DLL functions have been registered in a workspace, they are still available if the workspace is saved and subsequently loaded. The CallDLL facility assures that the DLL files are loaded and unloaded appropriately.

Every time a workspace is saved all DLL libraries are unloaded and the internal table is adjusted to reflect that. When a workspace is loaded, none of the DLL libraries will be loaded until a specific call to *calldllfun* brings in the DLL. This process assures that resources are not wasted on registered DLL calls that are never used.

The *registerdllfun* routine must always be called first to enter DLL function into the internal mapping table. This table keeps track of the name and DLL file for the routine and all of the argument types and function results. The DLL file is **not** loaded until the routine is called with the *calldllfun* routine. So errors involving paths to the DLL file or other related problems are not reported until the first call to *calldllfun*.

The operation *registerdllfun* initiates an interface to an external routine implemented as a DLL for 32-bit Windows. The name *Nm* is the name associated with the routine, *Dllnm* is the actual name of the DLL routine, *Path* is the path/filename to the DLL executable file, *Restype* is the result type and *Argtypes* is the list of argument types.

Nm is a phrase or string that is used to identify the requested DLL routine when using the *calldllfun* operation. It can be any name you desire. Duplicates are not allowed and the registration routine will overwrite previous table entries if the same name is used again.

Dllnm must be the exact name of the routine in the DLL library (case sensitive, including underscores, etc.).

Path must the full path to the DLL library file. If the DLL library file is in the standard search path for DLLs then the full path is not required.

Restype must be a string or phrase of one the valid types described below.

Argtypes must be a list of the valid types described below.

If the only argument is *Nm* and this names an already installed DLL routine, then the routine is unregistered and the DLL is unloaded.

Calling Conventions (C or Pascal)

The facility is able to map in DLL routines compiled for both C and Pascal calling conventions. The facility makes this determination by checking for a leading underscore on the internal name of the DLL routine. If one exists, then it is assumed that the C calling style should be used, otherwise the PASCAL calling style is used. It is possible to create DLL routines that do not follow this standard convention, so be careful.

When using either style calling conventions, the facility has a limit of up to 10 words of arguments (10 - 4 byte integers). Pascal calling convention is strict in that the number of supplied arguments must be the same as the number of required arguments. The limit of 10, is a compiler option and can be extended if necessary.

Arguments to DLL routines

Currently, Q'Nial is able to pass only simple types to DLL routines. The following is a list of the types that can be passed and returned as results:

Type in DLL routine	**Nial Type**
CHAR	character
SHORT	integer
LONG	integer
WCHAR	integer
LPSTR	string
LPTSTR	string
HANDLE	integer
DWORD	integer
LPDWORD	integer
WORD	integer
BOOL	integer
HWND	integer
UINT	integer
DOUBLE	real
FLOAT	real
LRESULT	integer
LPARAM	integer
LRESULT	integer
HMENU	integer
LPCTSTR	string
SCHAR	char
SDWORD	integer
SWORD	integer
UDWORD	integer
UWORD	integer

SLONG	integer
SSHORT	integer
ULONG	integer
USHORT	integer
INT	integer

These type are the simple base types used in the Windows (Win32) API (plus common types which are bolded). The above type names (in string or phrase form) can be used as the type names required as arguments to the *registerdllfun* operation. Each particular type affects Nial values in a different way. Use of types other than common types generally requires a knowledge of C/C++ types and casting. The Windows types are supplied to easily map in Windows API DLL calls.

When calling DLL routines using *calldllfun*, the facility will properly cast the incoming Nial type/value to the type of the particular argument specified by the *registerdllfun* operation. If the Nial type cannot be converted, then *calldllfun* will report an error and exit.

Variable Parameters

Variable parameters are supported. Arguments that are marked as **variable parameters**, have the returning value of the parameter added to the result list with the first element being the normal result of the called DLL routine. The Nial user must then select the portions of the result that are desired.

This feature also provides the ability to pass buffers to DLL routines. Many Windows OS DLL routines require a (char *) type pointer to a pre-allocated buffer and a buffer size. The routine places the result in the buffer up to the specified size. The interface to such a routine can be accomplished by sending a Nial string of the desired buffer size and the accompanying size integer as the arguments, having specified the first argument as a variable parameter. The resulting string is added to the result list with the first element being the normal result of the DLL routine call.

When passing variable parameters, the facility makes an internal copy of the data being passed. The called DLL routine uses this copy. This approach helps prevent the possibility of a DLL routine incorrectly manipulating the actual contents of a Nial array and causing a corruption of the workspace.

To specify that an argument is a variable parameter in the *registerdllfun* call prepend a * (asterisk) to the beginning of the type name (i.e. *CHAR). This indicates that the type is to be passed as a pointer (if it is not already a pointer), and that the result for that parameter is to be appended to the result of the entire call. Any, none or all of the parameters may be variable.

Functions Requiring Pointers to Data

Another common requirement of Windows OS DLL calls (and others too), is that they require the *address* of a variable to be passed as an argument instead of the actual value. The DLL function uses the value at the supplied memory address.

This capability is supported, and can be denoted by prepending a *&* (ampersand) to the beginning of the type name (i.e. &INT). At the Nial level, you still pass the *calldllfun* call a plain integer for that particular argument, but the CallDLL interface allocates memory for that argument, copies the value into the new memory, and passes the address of the memory cell to the DLL call. Upon return, the allocated memory is

deallocated.

The & modifier can be used in conjunction with the * modifier. This usage generally denotes an argument that is a pointer to a type and whose value you wish returned along with the result.

Example 1

Let say we have a DLL routine that takes 3 argument. The first argument is the true argument and the next two are pointers to memory locations where two additional results are to be placed. The function also returns an integer result.

The routine squares the first argument and places the result at the memory location specified by the second argument. It also cubes the first argument and place the result in the location specified by the third argument. The function result is what is supplied as the first argument.

```
     registerdllfun "sq "_squarecube 'c:\myops.dll' "INT ["INT,"&*INT,"&*INT]

     calldllfun "sq 5 -10 -10
+---+----+-----+
| 5 | 25 | 125 |
+---+----+-----+
```

So the call passes in 5 as the true argument and two -10 values as dummy values for the next two arguments (it is possible that the function might require all arguments to specify proper values). The result is an array of 3 items, one is the function result and an additional item for every variable parameter in the argument list.

Example 2

The following is a call to a Windows API routines that requires a string buffer to be passed to the call:

```
     registerdllfun "_GetComputerName "GetComputerNameA
        'kernel32.dll' "BOOL ['*LPTSTR', 'LPDWORD'];

GetComputerName IS {
   res name := calldllfun "_GetComputerName (1000 reshape ' ') 1000;
   IF res = 1 THEN
     name
   ELSE
     'Error Getting Computer Name'
   ENDIF
}

     GetComputerName
INIGO
```

Notice how the above code calls the DLL routine with an empty string buffer (1000 reshape ' ') and also passes the size of the buffer (1000). The Windows API requires this in order to safely write the Computer name in the buffer without overflowing the string buffer. The buffer argument is marked as a variable argument, so the result is passed back in the *calldllfun* call.

A Word of Warning

Use of DLL routines can cause unexpected results if incorrect parameters as passed. It is reasonable easy to crash the Nial application with a bad DLL call. Additionally, memory allocated inside a DLL function is not deallocated by Nial. Unless there is a symmetric DLL routine to deallocate memory, a memory leak will result if such a routine is used.

regular expression notation

Class: concept

See Also: regexp, regexp_match, regexp_substitute

The following table gives the standard notations used in regular expressions:

Notation	Matching String
.	any character
*	the previous character zero or more times
+	the previous character 1 or more times
^	when at the beginning of a regular expression, matches the beginning of the string.
$	when at the end of a regular expression, matches the end of the string.
[xyz]	any character in the sequence of characters "xyz" where "xyz" can be almost any sequence of characters.
[^xyz]	any character NOT in the sequence of characters "xyz".
[x-y]	any character in the range of x to y.
|	allows the optional match of the regular expression on the left or the right of the "|".
()	bracketing allows the specification of groups. Anything that is matched between a set of brackets can later be extracted. Also used for bracketing regular expressions to force order of precedence.
<other char>	itself
\<any char>	the character

repeat-loop

Class: control structure

Usage: `REPEAT expression_sequence UNTIL conditional_expression ENDREPEAT`

See Also: while-loop, for-loop

The *REPEAT-loop* notation is used when executing an expression sequence repeatedly until a conditional expression returns *true*.

```
F := open Filenm "r;
Lines := '';
Done := o;
REPEAT
  Line := readfile F;
  IF isfault Line THEN
    Done := l;
  ELSE
    Lines := Lines append Line;
  ENDIF;
UNTIL Done ENDREPEAT;
```

reserved words

Class: syntax

A **reserved word** or **keyword** is one that has a special usage in a Nial construct and must be used only for that purpose. A *block* delimits a local environment. It allows new uses of names which do not interfere with uses of those names outside the block. For example, within a block, a predefined operation name can be redefined and used for a different purpose. Only the reserved words of Q'Nial cannot be reused in this fashion.

An identifier, which is spelled the same, ignoring case, as any of the reserved words given below cannot be used to name a variable or a definition. In a local environment, an identifier can be chosen that is the same as a predefined or user-defined global definition name. Such a choice makes the global use of the name unavailable in the local context.
A reserved word is displayed in upper case in canonical form.

Reserved Word	Construct
BEGIN	Synonym for { in block
CASE	case-expression
DO	for-expression, while-expression
ELSE	if-expression, case-expression
ELSEIF	if-expression
END	case-expression, synonym for } in block
ENDCASE	case_expression
ENDFOR	for-expression
ENDIF	if-expression
ENDREPEAT	repeat-expression
ENDWHILE	while-expression
EXIT	exit-expression
EXPRESSION	declaration
EXTERNAL	declaration

FOR	for-expression
FROM	case-expression
GETS	assign-expression
IF	if-expression
IS	definition
LOCAL	declaration
NONLOCAL	declaration
OP	synonym for operation
OPERATION	operation-form, declaration
REPEAT	repeat-expression
THEN	if-expression
TR	synonym for transformer
TRANSFORMER	transformer-form, declaration
UNTIL	repeat-expression
VARIABLE	declaration
WHILE	while-expression
WITH	for-expression

reshape

Class: reshaping operation

Property: binary

Usage: `A reshape B   reshape A B`

See Also: post, shape, tally

The operation *reshape* is the major mechanism in Nial for creating multivalent arrays. The operation requires *A* to be a shape, either *Null* or a list of integers. If *A* is an integer, it is converted to the solitary holding the integer. If *A* is not a shape, the result is the fault *?shape*.

The result of *reshape* is an array of shape *A* with items chosen from the list of items of *B*. If the number of items to be used to fill the result is less than the tally of *B*, the remaining items of *B* are ignored. If the number is more than the tally of *B*, the items of *B* are used cyclically. If *B* has no items, fault *?fill* is used as the items of the result.

```
     2 3 reshape 4 6 (4 5) 3 "abc
+-+---+---+
|4|  6|4 5|
+-+---+---+
|3|abc|  4|
+-+---+---+
```

```
     2 3 reshape 'abcdefghij'
abc
def
```

```
      5 reshape solitary 3 4
+---+---+---+---+---+
|3 4|3 4|3 4|3 4|3 4|
+---+---+---+---+---+
```

The last example illustrates that to replicate an arbitrary array, in this case the pair 3 4, *reshape* is applied to the solitary holding the array.

Equations

```
shape A reshape A = A
shape A reshape list A = A
shape A reshape (first A hitch rest A) =f= A
A a shape implies ==> shape (A reshape B) = A
A reshape list B = A reshape B
A is a shape ==> EACH f (A reshape B) = A reshape EACH f B
tally A reshape A = list A
(tally A) 1 reshape A = post A
single A = Null reshape solitary A
solitary A = [ 1 ] reshape single A
Null = [ 0 ] reshape A
```

rest

Class: selection operation

Usage: `rest A`

See Also: first, front, drop, sublist

The operation *rest* returns the list of *A* after dropping the first item of the list. If the argument of *rest* is not a list, it is treated as a list.

```
     rest 3 4 5 6
4 5 6

     rest tell 2 3
+---+---+---+---+---+
|0 1|0 2|1 0|1 1|1 2|
+---+---+---+---+---+

     rest 7 = Null
1
```

The last example shows that the rest of an atom is the empty list *Null*.

Definition

```
     rest IS OPERATION A (1 drop list A)
```

Equations

```
rest A = rest list A
not empty A ==> first A hitch rest A = list A
tally rest A = 0 max (tally A - 1)
first rest A = second A
rest Null = Null
```

restart

Class: system expression

Usage: `Restart`

See Also: clearws, load, latent, recover, toplevel

The expression *Restart* is used to restart a Q'Nial session. It causes the workspace to be cleared and Q'Nial to be reinitialized to be in the same state it was when the session began. If the original invocation used a named workspace or a definition file, it is reloaded as before.

The advantage of *Restart* over *Clearws* is that the user can start the application again using *Restart* and possibly trigger a *Latent* expression in the starting workspace.

resume

Class: debugging command

Usage: `resume`

See Also: break, step, next, toend, debugging, setdeftrace

When *break debug mode* is initiated by means of the *Break* expression or the operation *breakin* (or by using <*Ctrl b*> while awaiting input in window mode on a console version), a break loop is initiated with output like:

```
-------------------------------------------------------------
    Break debug loop: enter debug commands, expressions or
      type: resume    to exit debug loop
      <Return> executes the indicated debug command
    current call stack :
foo
-------------------------------------------------------------
?.. C := A + ( + A + A )
-->[stepv]
```

In the above example the break loop awaits input with the default command *stepv* assumed. If *resume* is typed then the break loop is exited and control returns to the expression following the break point.

```
     Break
-------------------------------------------------------------
    Break debug loop: enter debug commands, expressions or
      type: resume    to exit debug loop
      <Return> executes the indicated debug command
    current call stack :
-------------------------------------------------------------
?.. C := A + ( + A + A )
-->[stepv] resume
```

reverse

Class: data rearrangement operation

Usage: `reverse A`

See Also: transpose, rotate, list

The operation *reverse* returns an array of the same shape as *A* having the items in reverse order.

```
     reverse 4 5 6 7
7 6 5 4

     reverse 'able was I ere I saw Melba'
ableM was I ere I saw elba

     EACH reverse 'This' 'seems' 'too' 'wonderful'
+----+-----+---+---------+
|sihT|smees|oot|lufrednow|
+----+-----+---+---------+

     reverse count 3 4
+---+---+---+---+
|3 4|3 3|3 2|3 1|
+---+---+---+---+
|2 4|2 3|2 2|2 1|
+---+---+---+---+
|1 4|1 3|1 2|1 1|
+---+---+---+---+

     reverse (10 take) reverse sketch 1.23
      1.23
```

The fourth example illustrates that the reverse of a 3 by 4 table is a 3 by 4 table with the list of items reversed. The last example shows the use of *take* and *reverse* to right-justify text in a field.

Definition

```
     reverse IS OPERATION A {
        shape A reshape (tally A - count tally A choose A) }
```

Equations

```
   shape reverse A = shape A
   reverse reverse A = A
   reverse A=shape A reshape reverse list A
   front A = reverse rest reverse A
   last A = first reverse A
```

role

Class: concept

See Also: global environment

The term **role** is used to describe the class of object associated with an identifier in Nial. The possible roles are: reserved word, variable, expression, operation or transformer. An identifier that corresponds to a reserved word can play no other role in a workspace. An identifier that corresponds to a predefined object in Nial cannot be changed in the global environment, but can take on a different object association in a local environment. In the global environment, once the role of an identifier is established, it must keep the same role, but may have its association changed to another object of the same role.

An external-declaration assigns a role to a name in the global environment so that the name can be used in other definitions before it is completely specified. This mechanism is useful for creating mutually recursive definitions. An external declaration is made only in the global environment.

If the name is already defined with the same role, the declaration has no effect. If the name has another role, a fault is reported.

If the expression on the right of *IS* in a definition uses the name being defined, the definition is assumed to be recursive. The name is assigned a role compatible with its use on the right if it does not already have a role.

If a definition appears within a block, the association between the name of the identifier and its meaning is made in the local environment. Otherwise, the association is made in the global environment and the definition assigns a role to the name as representing that kind of expression.

If the name being associated in a definition is already in use, the new definition must be for a construct of the same role and the earlier definition is replaced. The use of a defined name always refers to its most recent definition.

rotate

Class: data rearrangement operation

Property: binary

Usage: `A rotate B  rotate A B`

See Also: reverse, transpose, list

The operation *rotate* shifts the items of array *B* to the left *A* places, inserting the items that drop off the front of the list on the back. If *A* is a negative number, the shift is to the right.

```
     2 rotate 24 35 46 57 12 24 39
46 57 12 24 39 24 35

     -5 rotate tell 3 5
+---+---+---+---+---+
|2 0|2 1|2 2|2 3|2 4|
+---+---+---+---+---+
|0 0|0 1|0 2|0 3|0 4|
+---+---+---+---+---+
|1 0|1 1|1 2|1 3|1 4|
+---+---+---+---+---+
```

The second example shows that on a table *rotate* works on the list of items in row major order.

Definition

```
rotate IS OPERATION N A {
   Newlist :=  tally A + N + grid A mod tally A choose list A;
   shape A reshape Newlist }
```

Equations

```
N an integer ==> shape (N rotate A) = shape A
N rotate A = shape A reshape (N rotate list A)
N rotate A = N mod tally A rotate A
N rotate A = opposite(tally A-N) rotate A
```

rows

Class: nesting restructuring operation

Usage: `rows A`

See Also: cols, split, mix, rank

The operation *rows* is normally used on a table *A*. In this case, it returns a list of lists, with each item being a list of items from one row of the table. For a multidimensional array, the result is an array of lists of length equal to the length of the last axis of *A*. The valence of the result is one less than the valence of *A*.

```
     A := 2 3 reshape "Bill "Tom "Mary "Jack "Dawn "Alice
Bill Tom  Mary
Jack Dawn Alice

     rows A
+-------------+---------------+
|Bill Tom Mary|Jack Dawn Alice|
+-------------+---------------+

     A :=  2 2 2 3 reshape count 32
1 2 3       7  8  9
4 5 6      10 11 12


     rows A
+-----+---------+  +--------+--------+
|1 2 3|4 5 6    |  |13 14 15|16 17 18|
+-----+---------+  +--------+--------+
|7 8 9|10 11 12|  |19 20 21|22 23 24|
+-----+---------+ +--------+--------+
```

Definition

```
     rows IS OPERATION A ( valence A - 1 max 0 raise A)
```

Equations

```
   shape rows A = front shape A not empty A ==> shape first rows A = last shape A
   not empty A ==> mix rows A = A
   T a variable associated with a table ==> I pick rows T = T|[I,]
```

save

Class: system operation

Usage: `save Wsname`

See Also: load, checkpoint, recover, latent, restart

The operation *save* is used to save a workspace with the name given by the phrase or string *Wsname*. The convention in Q'Nial is to name workspaces with the extension *.nws* so that they are easy to find in the directory. When saving or loading a workspace, the extension may be omitted.

The effect of saving a workspace is to place the contents of the current workspace in the named file in an internal format. The contents of the current workspace are unchanged by doing a *save*.

```
save "mywork
```

A *save* operation may be executed within an operation or expression. The effect is to interrupt the execution and do the save at the top level. If the saved workspace contains a user-defined expression with the name *Checkpoint*, at the termination of the *save*, *Checkpoint* is executed. *Checkpoint* can be used to restart the computation.

If the saved workspace contains an expression with the name *Latent*, whenever the workspace is loaded, the *load* operation will execute the *Latent* expression. This can be used to have a workspace automatically begin an application, or to set up desired default values for various switch settings.

scan

Class: evaluation operation

Usage: `scan S`

See Also: parse, eval, descan, deparse, execute

The operation *scan* translates Nial program text to internal form. This process is called tokenizing as it results in a list containing tokens. The argument *S* is a string. The result is a list beginning with 99 and followed by an alternating sequence of integer codes and phrases.

```
     scan 'A := 3 * 5.2'
99 2 A 1 := 16 3 2 * 18 5.2

     scan 'sum 2 (35 + 42.7)'
99 2 SUM 16 2 1 ( 16 35 2 + 18 42.7 1 )

     scan 'foo IS first rest'
99 2 FOO 1 IS 2 FIRST 2 REST
```

The first example has 5 tokens, the second has 7 and the third has 4. In a token stream, the token for an identifier is in upper case.

The *scan* token codes are given below:

Code	Meaning
1	reserved word or delimiter
2	identifier
14	string
15	phrase
16	integer
18	real number
22	fault
42	atomic character
40	atomic boolean or bitstring

Equations

```
S a string ==> execute S = eval parse scan S
S a string ==> scan descan scan A = scan A
```

scope of a variable

Class: concept

See Also: local environment, role

The use of an assign-expression indicates that a name (identifier) is to be treated as a variable in the context surrounding the assign-expression. This context is called the **scope** of the variable. The context may be global, in which case the variable may be visible at all levels; or it may be local to some region of program text. A local scope is created for the parameters of operation forms and for variables created within a block.

Because operation forms or blocks may appear within other operation forms or blocks, it is possible to have one scope for a name nested within another. A name is said to be visible at a point in a program text if it has a local meaning at that point or has a meaning in some surrounding scope or is a global name. When a name is used in a local scope, it is the local association in the innermost scope that is used, instead of an association with the same name in a surrounding scope.

A block is a scope-creating mechanism that permits an expression-sequence to be created so that it has local definitions and variables which are visible only inside the block. A block may appear as a primary-expression or as the body of an operation-form.

Definitions that appear within the block have local scope. That is, the definitions can be referenced only in the body of the block. Variables assigned within the block may or may not have local scope, depending on the appearance of a local and/or a nonlocal declaration. If there is no declaration, all assigned variables have local scope. Declaring some variables as local does not change the effect on undeclared variables that are used on the left of assignment. They are automatically localized.

If a nonlocal declaration is used, an assigned name that is on the nonlocal list is sought in surrounding scopes. If the name is not found, a variable is created in the global environment.

screensize

Class: window mode input/output expression

Usage: `Screensize`

See Also: clearscreen, setcursor, readcursor, setwindow, window

The expression *Screensize* returns the effective size of the screen in use by Q'Nial. The result is a pair of integers giving the number of rows and columns. For the EXTDOS version it is always 25 80.

```
     Screensize
25 80
```

Screensize is useful in writing portable applications that use the alphanumerics window package. The size of windows can be tailored to the available size.

second

Class: selection operation

Usage: `second A`

See Also: first, third, pick, take

The operation *second* returns the second of the items of *A*. If *A* has only one item or it is empty, it returns the fault *?address*.

```
     second 4 5 6
5

     second tell 3 4
0 1

     second Null
?address
```

Definition

```
     second IS OPERATION A (1 pick list A)
```

Equations

```
   second list B = second B
   second B = first rest B
```

see

Class: system operation

Usage: `see Defname`

See Also: descan, deparse, getdef, defedit

The operation *see* displays the definition of the user defined object named by the phrase or string *Defname*. The canonical form of the definition appears directly on the screen. The operation cannot be used on predefined names. The display is in *sketch/nodecor* mode regardless of the current settings.

```
     foo IS first rest;
     see "foo
foo IS first rest
```

It is possible to capture the display given by *see* using the composition of operations *descan deparse getdef Defname*, which returns the list of lines of the display. This technique is used in the operation *defedit*.

Definition

```
     see IS OPERATION A {
        Settings gets EACH set "sketch "decor;
        ITERATE writescreen descan deparse getdef A;
        EACH set Settings; }
```

seed

Class: array generation operation

Usage: `seed Num`

See Also: random, floor

The operation *seed* sets the initial number for the random number generator used by *random*. The argument *Num* is a positive decimal fraction less than 1. *Seed* is used to get predictable results from a program that uses *random*. *Seed* returns the value that would have been used to generate the next random number.

```
     seed .25374;
     floor (100. * random 3)
60 73 83
```

seek

Class: search operation

Property: binary

Usage: `A seek B` `seek A B`

See Also: in, find

The operation *seek* returns a pair of values. The first is a boolean indicating whether or not *A* is an item of *B*. The second is the address of the first occurrence of *A* as an item of *B*, searching *B* in row major order. If *A* does not occur in *B*, the second item of the result is the gage of the shape of *B*. *Seek* is the combination of *in* and *find* in one operation.

```
     3 seek 56 34 3 23 57 3
1 2

     `a seek 'hello world'
o 11
```

Seek is useful when determining whether or not *A* is in *B* and, if it is, obtaining its position in *B*. Both results are obtained in one internal computation rather than two separate ones.

Definition

```
     seek IS OPERATION A B { A [in,find] B }
```

Equations

```
   first (A seek B) = A in B
   second (A seek B) = A find B
```

Pragmatics

The operation *seek* uses a linear search on the items of B if the array has not been sorted, or uses a binary search algorithm if it has. The latter fact suggests that an array that is searched frequently should be kept in lexicographical order by applying *sortup* to it when it is created.

seeprimcalls

Class: system operation

Usage: `seeprimcalls Mode`

See Also: break, debugging, breaklist

The operation *seeprimcalls* sets an internal flag to the setting of the boolean argument *Mode*. If *Mode* is *true*, subsequent execution will monitor the use of all primitive defined expressions, operations and transformers and will print out a message indicating when the named primitive has completed execution. If *Mode* is *false* it turns off the flag that controls monitoring.

This mode of execution is useful for following the execution flow during debugging.

```
     seeprimcalls True
o
```

seeusercalls

Class: system operation

Usage: `seeusercalls Mode`

See Also: break, debugging, breaklist

The operation *seeusercalls* sets an internal flag to the setting of the boolean argument *Mod*. If *Mod* is *true*, subsequent execution will monitor the use of all user defined expressions, operations and transformers and will print out a message indicating when definitions are entered and executed. If *Mod* is *false* it turns off the flag that controls monitoring.

This mode of execution is useful for following the execution flow during debugging.

```
     seeusercalls True
o
```

separator

Class: constant expression

Usage: `Separator`

See Also: nialroot, library, host, link

The expression *Separator* returns the character used to separate names in the directory structure for the operating system. For DOS and Windows systems, it is the backslash symbol "\". For Unix, it is the slash symbol "/".

set

Class: system operation

Usage: `set Sw`

See Also: setdeftrace, setformat, setinterrupts, setlogname, setprompt, setwidth

The operation *set* changes internal switches in the Q'Nial interpreter. The switches are used to control the behaviour of a variety of optional features of Q'Nial. The argument *Sw* is a string or phrase in either case. The result of set is a phrase giving the setting of the switch as it was prior to the execution of the set operation. The result can be used subsequently to restore the switch to its original value. The valid phrases and their purposes are tabulated below:

Setting	Description of Result
diagram	set default display mode to diagram
sketch	set default display mode to sketch
decor	turn on decoration of atoms and empty arrays
nodecor	turn off decoration of atoms and empty arrays
trace	turn on tracing of expressions at the top level
notrace	turn off tracing of expressions at the top level
log	turn on the automatic logging of the session
nolog	turn off the automatic logging of the session

```
set "diagram ;
displayinsketchmode IS OPERATION Result {
   A := set "sketch; write picture Result; set A ; }
```

In the above definition, sketch mode is set at the beginning, saving the previous setting in the variable *A*. When the definition is ended, the mode is reset to the value in *A*. The default settings are *sketch*, *nodecor*, *notrace* and *nolog*.

setdeftrace

Class: system operation

Usage: `setdeftrace Defname Mode`

See Also: set, step, break, defedit

The operation *setdeftrace* changes the trace mode for the user defined expression, operation or transformer definition named by the string or phrase *Defname*. *Mode* is an optional argument. If it is omitted, the trace mode is toggled. If *Mode* is present and has the value 1, it turns trace on. Otherwise it turns trace off. The effect when the trace mode is on is to trace the execution of the body of the operation whenever the operation is executed.

The trace mechanism shows intermediate values in the evaluation of expressions.

The operation returns the previous setting. If the result of *setdeftrace* is assigned to a variable, the previous

setting can be restored later.

```
     library "average

     setdeftrace "average
0

     average 3 4 5
...trace call to operation
...average
...the arguments for the opform are
...A
...A
3 4 5
...A
3 4 5
...sum A
12
...sum A / tally A
4.
...end of operation call
4.
```

The tracing facility for expressions does not trace all intermediate computations. Parentheses around an expression will force its result to be shown during a trace.

setformat

Class: picture operation

Usage: `setformat String`

See Also: sketch

The operation *setformat* controls the format used for picturing real numbers. *String* is a format specification for real numbers using the conventions for the C library routine printf. There are three styles of format:

Control	Effect
%f	displays a fixed number of places after the decimal point in a fixed size space with no scaling of the number
%e	displays the number in scientific notation with an exponent scaling the number to have one digit before the decimal point
%g	displays the number in f format if possible but defaults to e format if the number is not within a suitable range

A code can be inserted between the percent sign and the letter. Format *'%15.5f'* uses a field of width 15 to display a number in *f* format with 5 decimal places. The first digit is the width of the field. For *f* format, the second digit gives the number of places after the decimal while for *e* and *g* it indicates the number of significant digits to be displayed. Either digit can be omitted.

Q'Nial requires that the display of a real number includes a decimal point and that the display of an integer does not include a decimal point. In a *%g* format, printf does not include a decimal point if the real number matches an integer to the specified precision. In this case, Q'Nial makes the field one space wider than that specified in order to accommodate the decimal point. If an *f* format is not wide enough to accommodate the number, it is widened so that the number is displayed.

If the result of an *f* format requires more than *40* digits to the left of the decimal point, it is converted to *e* format.

The format *'%.17g'* is used by operation display in depicting real numbers as this format is accurate enough to reproduce the same number when executed on most systems. The default format is *'%g'*. It works as in C except that a period may be added to distinguish the result from an integer constant. If the argument to setformat is *''* then it resets the formatting to the default value.

```
     setformat '%.5f' ;
     post exp 10. 80. 90. 100.
                                    22026.46579
    5540622384393509830000000000000000.00000
    1220403294317840830000000000000000000000.
2.688117141816135610000000000000000000000e+43

     X := 2 2 reshape  9777.985 4464784.692 4070333.511 0.079;

     setformat '%15.7f' ; [pass,sqrt] X
+--------------------------------+--------------------------------+
|   9777.9850000 4464784.6920000|     98.8836943    2113.0037132|
|4070333.5110000       0.0790000|   2017.5067561       0.2810694|
+--------------------------------+--------------------------------+

     setformat  '%15.7g' ; [pass,sqrt] X
+--------------------------------+--------------------------------+
|        9777.985         4464785.|       98.88369        2113.004|
|        4070334.            0.079|       2017.507       0.2810694|
+--------------------------------+--------------------------------+

     setformat '%15.7e' ; [pass,sqrt] X
+--------------------------------+--------------------------------+
|  9.7779850e+03   4.4647847e+06|  9.8883694e+01   2.1130037e+03|
|  4.0703335e+06   7.9000000e-02|  2.0175068e+03   2.8106939e-01|
+--------------------------------+--------------------------------+
```

setinterrupts

Class: system operation

Usage: `setinterrupts Boolean`

See Also: recover, toplevel

The operation *setinterrupts* permits or prevents the interruption of computation by the use of <*Ctrl C*>. If the argument is *true*, interrupts are permitted. If it is *false*, they are blocked. The default value is *true*.

Setinterrupts is used in a closed application to prevent the user from interfering with a computation in process. This may be critical if a file is being updated in a sequence of steps, which must all be completed for the data base to remain consistent.

setlogname

Class: system operation

Usage: `setlogname Filename`

See Also: set

The operation *setlogname* is used to set the name of the host file in which a log of the session is recorded. The current log file is closed and a new one opened. The argument *Filename* must be a phrase or a string.

```
     setlogname "dec17pm
```

The main purpose of *setlogname* is to capture the history of part of a session in a particular file for later use. The log file can produce documentation, for example.

setprofile

Class: profiling operation

Usage: `setprofile A`

See Also: profile, clearprofile, profiletable, profiletree, profiling

The operation *setprofile* is used to turn on or turn off the gathering of profiling statistics. The calls must be made at the same scope level in order for profiling to work correctly. This can either be at the top level, or within the body of some expression or operation.

The argument is a boolean value: *true* starts the profiling process and *false* stops it. Profiling can be turned on and off several times within one profiling session.

A detailed explanation of the profiling mechanism is given in the help entry on *profiling*.

Example of use:

```
     setprofile True
o
```

setprompt

Class: system operation

Usage: `setprompt S`

See Also: set

The operation *setprompt* sets the Q'Nial prompt to be the string or phrase S. The default prompt is the string with 5 blank spaces. The maximum size of a prompt is forty characters. *Setprompt* provides the facility to set a visible prompt which can be distinguished from prompts issued by other software.

```
     setprompt 'qnial>'
qnial>
```

settrigger

Class: system operation

Usage: `settrigger Switch`

See Also: quiet_fault, fault, fault triggering

The operation *settrigger* sets the action taken when a fault is generated in an operation. If *Switch* is *true*, whenever a fault is generated, computation is interrupted, a message is displayed indicating the fault that has occurred and the expression where it occurred. Then the *Callstack* is displayed and a loop is entered that allows you to explore the cause of the fault. If you press *Return* at the prompt, control is sent to the top level loop.

If *Switch* is *false*, fault triggering is turned off.

The default setting at the start of a session is fault triggering on. The triggering of an interrupt by fault generation can be turned off for the session by using the -s command line option in console versions or by setting preferences in the GUI version.

```
     settrigger o
1
```

setwidth

Class: system operation

Usage: `setwidth N`

See Also: set, writescreen

The operation *setwidth* sets the width of the display and log lines. The argument is an integer *N*. The result is the previous setting. The default setting is 80.

Setwidth is useful in controlling the format of display on output saved for documentation purposes. For example, a setting of 130 allows wide Nial diagrams to be printed on a line printer even though they may not display properly on the screen. Narrower settings are convenient for use in reports.

```
     setwidth 50
80
```

shape

Class: measurement operation

Usage: `shape A`

See Also: tally, valence, reshape

The operation *shape* returns an array that describes the rectangular structure of array *A*. Every array has a shape. For a single, including atoms, the shape is the empty list *Null*. For a list, it is a solitary holding the integer giving the length of the list. For a table or higher valence array, it is a list of integers giving the extent (number of items) along each axis.

```
     shape 5 6 7
3
     A := 2 3 reshape 4 5 6 7 8 9;
     shape A
2 3
     B := 2 3 4 5 0 2 reshape 2 ;
     shape B
2 3 4 5 0 2
```

The operation *shape* always returns a list whereas *tally* returns an integer. An empty array such as *B* is one with a zero in its shape.

Equations

```
product shape A = tally A
tally shape A = valence A
list shape A = shape A
shape A reshape list A = A
```

simple

Class: structure testing operation

Property: predicate

Usage: `simple A`

See Also: atomic, twig

The operation *simple* tests whether or not an array has all atomic items or is an empty array. The result is *true* if the array is simple and *false* if it is not.

```
     (simple 2 3 4) (simple "abc)
ll

     simple Null
l

     simple (2 3) (4 5 6)
o

     (simple tell 2 3)
o
```

The first two examples show that a list of integers, an atom and the empty list *Null* are all simple arrays. The next two examples show that a pair of lists and the result of a *tell* are not simple.

Definition

```
     simple IS OPERATION A (EACH single A = A)
```

Equation

```
simple A = and EACH atomic A
```

sin

Class: scientific operation

Property: unary pervasive

Usage: `sin A`

See Also: sinh, cos, pi

The operation *sin* implements the sine function of mathematics. It produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	sine of the corresponding real
integer	sine of the corresponding real
real	sine of angle A given in radians
character	fault ?sin
phrase	fault ?sin
fault	argument A

```
     sin  1  -1  0.5  `a  "abc  ??error
0.841 -0.841 0.479 ?sin ?sin ?error
```

Equations

```
   sin opposite A = opposite sin A
   (sin A power 2) + (cos A power 2) = 1.0   (within roundoff error)
```

single

Class: construction operation

Usage: `single A`

See Also: solitary, null, atomic, reshape

The operation *single* returns an array with no axes holding *A* as its only item. The result is an array with shape *Null* and is said to be a **single**.

```
     single 2 3
o---+
|2 3|
+---+
     3 = single 3
1
```

The first example illustrates that a single of a pair contains the pair as its item. In both *diagram* and *sketch* modes, the display of a nonatomic single is decorated with an *o* in the upper left corner. The second example illustrates that the single of an atom is the atom itself.

Definition

```
single IS OPERATION A { Null reshape solitary A }
```

Equations

```
first single A = A
shape single A = Null
atomic A = single A equal A
EACH f single A =  single f A
A EACHLEFT f B = A EACHBOTH f single B
A EACHRIGHT f B = single A EACHBOTH f B
```

sinh

Class: scientific operation

Property: unary pervasive

Usage: `sinh A`

See Also: cosh, tanh, sin

The operation *sinh* implements the hyperbolic sine function of mathematics. It produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	hyperbolic sine of the corresponding real
integer	hyperbolic sine of the corresponding real
real	hyperbolic sine of angle A given in radians
character	fault ?sinh
phrase	fault ?sinh
fault	argument A

```
     sinh  1  -1  0.5  `a  "abc  ??error
1.1752 -1.1752 0.521095 ?sinh ?sinh ?error
```

Equation

```
sinh opposite A = opposite sinh A
```

sketch

Class: picture operation

Usage: `sketch A`

See Also: picture, diagram, paste, positions, display, set

The operation *sketch* computes a character table that gives the picture of the array *A* as it is displayed in *sketch* mode. The details of the display of atoms and empty arrays is affected by the setting of the

decor/nodecor switch.

```
sketch tell 1 3
```

The sketch of an array provides an abbreviated display that often serves as an adequate output format for data. The entry for *picture* has a more complete description of the picturing mechanism.

Definition

```
sketch IS OPERATION A {
   Old_setting := set "sketch;
   Result := picture A;
   set Old_setting;
   Result }
```

Equation

```
sketch sketch A = sketch A
```

solitary

Class: construction operation

Usage: `solitary A`

See Also: single, list, shape, hitch, link

The operation *solitary* returns a list of length one holding *A* as its only item. The result is said to be a **solitary**.

```
     solitary 2 3
+---+
|2 3|
+---+

     set "diagram; solitary 3
+-+
|3|
+-+

     set "sketch; solitary 3
3
```

The first example illustrates that a solitary of a pair contains the pair as its item. The second example illustrates that the solitary of an atom is different from the atom itself. However, as the third example shows, in sketch mode, the display of the solitary of an atom is not framed and hence may have the same display as the atom.

Definition

```
solitary IS OPERATION A { A hitch Null }
```

Equations

```
first solitary A = A
solitary A = A hitch Null
solitary A = Null append A
solitary A = list solitary A
```

solve

Class: linear algebra operation

Usage: `A solve B   solve A B`

See Also: inverse, innerproduct, inner, outer

The operation *solve* solves the set of linear equations described by the equation $A\ x = B$ in matrix notation, where A is an N by N matrix and B is a vector of length N. Provided that the matrix A is not singular, the result is the vector of length N satisfying the equation (within roundoff error).

In the computation, a numerical estimate is made of the singularity of A. If A cannot be shown to be non-singular with a safe margin, the result of the operation is the fault *?singular*.

The operation is extended to solve more than one right hand side. If B is an N by M matrix, columns of N are viewed as right hand sides and the result is an N by M matrix with each column being the solution of the corresponding column.

```
     seed .314159;
     A := ceiling (100. * (3 3 reshape random 9))
 8 75 24
63 31 99
20 95 48

     B := ceiling (10. *  random 3)
4 9 5

     X := solve A B
0.787745 0.111836 -0.445402

     A innerproduct X
4. 9. 5.

     I := 0 1 2 OUTER = 0 1 2 * 1.0
1. 0. 0.
0. 1. 0.
0. 0. 1.

     A solve I
 0.614865   0.102516 -0.518872
0.0810811 0.00745573 -0.055918
-0.416667 -0.0574713  0.347701

     inverse A
 0.614865   0.102516 -0.518872
0.0810811 0.00745573 -0.055918
-0.416667 -0.0574713  0.347701

     A solve I = inverse A
1
```

The final example shows that if the right hand side is set to the unit matrix of size N, solving it with A is equivalent to computing the inverse.

For large examples, it is always faster and more accurate to solve a system of equations directly using *A solve B* than to compute the inverse and do the inner product operation *inverse A innerproduct B* . The first form does about half as many arithmetic steps.

sort

Class: sorting transformer

Usage: `SORT f A`

See Also: grade, sortup

The transformer *SORT* returns the list of items of array *A* ordered according to the comparator *f*. The comparator *f* is the operation to be used in comparisons. The operations <= and >= are the usual comparators. If >= is the comparator in *SORT*, the items are returned in decreasing order. If <= is the comparator in *SORT*, the items are returned in increasing order.

```
     SORT >= (count 10)
10 9 8 7 6 5 4 3 2 1

     SORT <=("some "words "not "in "order)
in not order some words
```

Definition

```
     SORT IS TRANSFORMER f OPERATION A { GRADE f A choose A }
```

Equations

```
   shape SORT f A = shape A
   N an integer ==> SORT <= tell N = tell N
   SORT f (SORT f A) = SORT f A
```

The sort is done internally using either a radix sort (on integers) or a list merge sort algorithm from Knuth's Vol 3 Searching and Sorting, Algorithm 5.2.5-L, improved according to exercise 12.

sortup

Class: sorting operation

Usage: `sortup A`

See Also: gradeup, sort, up

The operation *sortup* returns the list of items of array *A* ordered according to the lexicographical ordering comparator *up*.

```
     sortup 3 7 5 4 9 8 2 1 6 10
1 2 3 4 5 6 7 8 9 10

     sortup ("some "words "not "in "order)
in not order some words
```

Definition

```
     sortup IS SORT up
```

Equation

```
   gradeup A choose A = sortup A
```

split

Class: nesting restructuring operation

Usage: `I split A   split I A`

See Also: blend, raise, rows, cols, rank

The operation *split* restructures array *A* by partitioning the items of *A* into arrays using the given axis numbers in *I* to determine the partition. The axis numbers in *I* indicate the axes that are to become axes of the items. The remaining axes are axes of the result. The argument *I* must be an array of integers in *tell valence A* without duplicates.

```
     A := 2 3 reshape count 6
1 2 3
4 5 6

     1 split A
+-----+-----+
|1 2 3|4 5 6|
+-----+-----+

     B := 2 3 4 reshape count 24
1  2  3  4   13 14 15 16
5  6  7  8   17 18 19 20
9 10 11 12   21 22 23 24

     2 split B
+-----------+-----------+-----------+
|1 2 3 4    |5 6 7 8    |9 10 11 12 |
+-----------+-----------+-----------+
|13 14 15 16|17 18 19 20|21 22 23 24|
+-----------+-----------+-----------+

     1 split B
+--------+--------+--------+--------+
|1 5 9   |2 6 10  |3 7 11  |4 8 12  |
+--------+--------+--------+--------+
|13 17 21|14 18 22|15 19 23|16 20 24|
+--------+--------+--------+--------+

     1 2 split B
+----------+-----------+
|1  2  3  4|13 14 15 16|
|5  6  7  8|17 18 19 20|
|9 10 11 12|21 22 23 24|
+----------+-----------+
     1 0 split B
+----+-----+-----+-----+
|1 13| 2 14| 3 15| 4 16|
|5 17| 6 18| 7 19| 8 20|
|9 21|10 22|11 23|12 24|
+----+-----+-----+-----+
```

The first example shows that splitting along the last axis of an array is equivalent to taking its *rows*. The second example shows that the *split* of the middle axis of a trivalent array is a table, with its shape given by the first and third axes. The last two examples illustrate two axes being "pushed down". The result is a list with tables as items. The order of the axes in the item is determined by the order of the items in *I*.

The operation *blend* uses the same control argument to undo the effect of a *split*. A common requirement is to partition an array along its axes, apply some operation *f* to the resulting items and to rebuild the result

into its original form. Partition is used to do this without computing all the intermediate structures.

Definition

```
split IS OPERATION I A {
   IF empty A THEN
      fault '?empty right arg in split'
   ELSEIF not(I allin axes A and diverse I) THEN
      fault '?invalid left arg in split'
   ELSE
     J gets axes A except link I;
     tally J raise (J link I fuse A)
   ENDIF }
```

Equations

```
valence A ~= 0 ==> valence A - 1 split A = rows A
diverse I and (I allin tell valence A) ==>
              shape first (I split A) = list (I choose shape A)
diverse I and (I allin tell valence A) ==>
              shape (I split A) = shape A except (I choose shape A)
```

sqrt

Class: scientific operation

Property: unary pervasive

Usage: `sqrt A`

See Also: exp, ln, log, abs, power

The operation *sqrt* implements the square root function of mathematics. It produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	square root of the corresponding real
integer	square root of the corresponding real
real	square root of A
character	fault ?sqrt
phrase	fault ?sqrt
fault	argument A

Example

```
     sqrt  o  1  0.5  `a  "abc  ??error
0. 1. 0.707107 ?sqrt ?sqrt ?error

     sqrt  4.0
2.
```

Definition

```
sqrt IS OPERATION A (A power .5)
```

Equation

```
sqrt A power 2 = A (within roundoff error)
```

standard definitions

Class: concept

See Also: clear workspace

Q'Nial has a large set of predefined expressions, operations and transformers. Most of these are implemented directly in the interpreter. However, there are a number of them which are defined as standard definitions in the file *defs.ndf*. This file is placed in the *initial* subdirectory of nialroot that is used in initializing Q'Nial. The definitions are embedded in the interpreter as a default set of standard definitions in case the interpreter cannot find the file.

When Q'Nial is invoked and the clear workspace is not found, the workspace is initialized with the standard definitions from *defs.ndf*. All definitions included in this stage cannot be modified or erased and the operation *see* does not display their text. A user can create their own initial set of definitions by editing *defs.ndf* to add or remove definitions.

status

Class: system expression

Usage: `Status`

See Also: filestatus

The expression *status* provides seven integers indicating the use of memory by Q'Nial. In order of display, the items are as follows:

Index	Quantity
0	Number of free words in the workspace
1	Number of words in the largest free block
2	Number of free blocks
3	Total number of words in workspace
4	Stack size in words
5	Atom table size in words
6	Internal buffer size

The first entry is an indication of how full the workspace is when compared with the fourth entry. A word can contain an integer, a reference to an array item, or 4 characters. The second entry gives an upper limit to the number of integers that can be in a largest size array. The third item is a measure of the

fragmentation of memory. The number of free words divided by the number of free blocks gives the average block size. The fourth item gives the current size of the workspace. It can grow provided the system has space available. The remaining entries give the sizes of internal areas that can grow as necessary.

The starting workspace size can be specified as a parameter to the *nial* command that starts a session in console versions or in a dialog box in a GUI version. The workspace and the other areas can grow in size provided there is sufficient space and workspace growth is allowed (the default).

In Q'Nial for Windows the status entries can be displayed in a pop up window by selecting the Status field in the Workspace menu.

```
     status
64198 63828 5 100000 4000 6000 1000
```

step

Class: debugging command

Usage: `step`

See Also: debugging, break, resume, next, toend, stepin

The command *step* is used in debugging a definition that has been suspended using Break or <Ctrl B>. The effect of *step* is to execute the next expression in an expression sequence and to suspend execution again. If the current expression involves a call on a defined operation or expression, execution is suspended on the first expression in its body. If the current expression is the last one in the expression sequence where the break began, the effect is the same as using *resume*.

The related command *stepv* displays the result of the expression executed on a step before displaying the next expression.

stepin

Class: debugging command

Usage: `stepin`

See Also: debugging, break, resume, next, toend, step

The command *stepin* is used in debugging a definition that has been suspended using Break or <Ctrl B>. The effect of *stepin* is to execute the next expression in an expression sequence tracing the intermediate values generated and to suspend execution again. If the current expression involves a call on a defined operation or expression, execution is suspended on the first expression in its body. If the current expression is the last one in the expression sequence where the break began, the effect is the same as using *resume*.

strand notation

Class: syntax

See Also: bracket-comma notation

A primary-sequence of length two or greater is called a **strand**. The value of a strand is a list of values. Each item of the list has the value of the primary-expression in the corresponding position in the primary-sequence.

The elements of the primary-sequence can be constants, parenthesized expressions or lists in bracket-comma notation.

```
     8 9 10
8 9 10

     8 (3 4 5) 18
+-+-----+--+
|8|3 4 5|18|
+-+-----+--+

     8 'abc' "apple
+-+---+-----+
|8|abc|apple|
+-+---+-----+

     6 [7,8,9] 10
+-+-----+--+
|6|7 8 9|10|
+-+-----+--+
```

string

Class: conversion operation

Usage: `string A`

See Also: phrase, fault, isstring, char, charrep, tolower, toupper

The operation *string* is used to convert an atom to a string that corresponds to the display of the atom. The form of the string is not sensitive to the setting of the *decor/nodecor* display mode, returning the sketch of the undecorated atom. The string of a string is the string itself.

```
     set "diagram ; string "abc
+-+-+-+
|a|b|c|
+-+-+-+
     string 'abc'
+-+-+-+
|a|b|c|
+-+-+-+

     string 2.35
+-+-+-+-+
|2|.|3|5|
+-+-+-+-+
```

The string of a phrase is the string of the message in the phrase. Thus, *string* is the left inverse of *phrase*.

Equations

```
isstring S ==> string S = S
isstring S ==> string phrase S = S
list string S = string S
```

string_split

Class: string manipulation operation

Usage: `string_split C S [N]`

See Also: cut, cutall, regexp

The operation *string_split* is used to break string *S* into substrings using the characters in *C* as places to break the string. If two characters in *C* are adjacent in *S* then an empty string is placed in the result. The optional third argument is an integer which limits the number of substrings returned.

```
     string_split ' .' 'The quick brown fox.'
+---+-----+-----+---+
|The|quick|brown|fox|
+---+-----+-----+---+

     string_split ' .,' 'The quick, brown fox.'
+---+-----++-----+---+
|The|quick||brown|fox|
+---+-----++-----+---+
```

The first example shows a string being broken on spaces and the period. The second is also broken on a comma and results in an empty string as one of the items of the result.

string_translate

Class: string manipulation operation

Usage: `string_translate C D S [O]`

See Also: cut, cutall, regexp

The operation *string_translate* is used to translate characters in string *S* based on mapping characters in *C* to the corresponding ones in *D*. The optional control argument *O* is a string or phrase where *'d'* deletes the characters in *C*, *'c'* complements characters by replacing characters not in *C* by the last character in *D*, and *'s'* translates characters in *C* to those in *D* but also squeezes many occurrences to one.

```
     string_translate 'abc' 'def' 'hello able cats'
hello dele fdts

     string_translate ' ' '_' 'a whole lot of     loving' 's'
a_whole_lot_of_loving

     string_translate 'xz' '' 'extraneous zeal' 'd'
etraneous eal

     string_translate 'xz' 'b' 'extraneous zeal' 'c'
bxbbbbbbbbbzbbb
```

The first example illustrates the translation of characters directly. The second shows translation with squeezing. The third example shows the use the *'d'* option to delete characters. Example four shows the use of the complement option.

sublist

Class: selection operation

Property: binary

Usage: `A sublist B  sublist A B`

See Also: cull, except, front, rest

The operation *sublist* returns a list of items of *B* chosen according to the list of booleans given in *A*, selecting those items of *B* where the corresponding item of *A* is *true*. If the tally of *A* is not the same as the tally of *B*, it is coerced to have the same tally as *B* using *reshape*. If *B* is not a list, the result is the same as applying *sublist* to the list of *B*. The tally of the result is the sum of *A* after it has been extended, if necessary.

```
     lolloll sublist 2 3 4 8 5 10 6
2 4 8 10 6

     lo sublist 'hello world'
hlowrd
```

The first example shows the use of *sublist* to select the items of the list that are even numbers. The second example uses the left argument cyclically.

Equations

```
shape (A sublist B) = sum (tally B reshape A)
list (A sublist B) = A sublist B
A sublist list B = A sublist B
l sublist A = list A
o sublist A = Null
```

sum

Class: arithmetic operation

Properties: multi pervasive, reductive

Usage: `sum A  + A  A sum B  A + B`

See Also: plus, minus, times, divide, opposite

The operation *sum* adds the items of a simple array of numbers, reducing them to a single number that is their sum. The type of the result is the highest type of the items. The operation is extended to non-simple arrays by the multi pervasive mechanism. For an empty simple array, the result is 0. The symbol + is a synonym for *sum*. The operation *plus* is *sum* restricted to use on pairs.

```
     R gets 1 2 2.5 `a "abc ??error;
     R OUTER + R
     2       3     3.5  ?A  ?A  ?error
     3       4     4.5  ?A  ?A  ?error
   3.5     4.5      5.  ?A  ?A  ?error
?A      ?A      ?A      ?A  ?A  ?A
?A      ?A      ?A      ?A  ?A  ?A
?error  ?error  ?error  ?A  ?A  ?error

     sum 3. 45. 23. 18. 3.5
92.5

     sum (3 4 5) (2 3 2) (0 2 4) (1 1 1)
6 10 12

     sum Null
0
```

The first example shows the result of *sum* on all combinations of types of atoms. The second last example shows that pervasive extension of *sum* adds a list of triples in an item by item fashion. The last example shows that the sum of an empty array is 0.

Equations

```
A sum B = B sum A
sum single A = EACH (sum single) A
sum EACH sum A =f= sum link A
tally link A = sum EACH tally A
```

symbols

Class: system operation

Usage: `symbols Sw`

See Also: status, exprs, ops, trs, vars, see

The operation *symbols* is used to get information on the use of names in the workspace. If *Sw* is 0, the result is a list of pairs giving the names and roles of all the user defined names. If *Sw* is 1, the result is the similar list for both system and user names. The roles and their meaning are as follows:

Role	Meaning
ident	identifier
var	variable
expr	expression
op	operation
tr	transformer
res	reserved word

```
     A := count 5;

     foo IS first rest

     symbols 0
+-----+------+
|A var|FOO op|
+-----+------+
```

synonym

Class: syntax

A **synonym** is an alternate symbol or name that represents a Nial term. For example, the symbol

The list of all synonyms in Nial follows:

Alternate	Role	Symbol
div	operation	divide
Falsehood, o	expression	False
inv	operation	inverse
ip	operation	innerproduct
istruthvalue	operation	isboolean
opp	operation	opposite
prod	operation	product
recip	operation	reciprocal
Truth, l	expression	True
vacate	operation	Null first
void	operation	Null first
+	operation	sum
-	operation	minus
*	operation	product
/	operation	divide

The following synonyms are available for keywords or delimiters used in the syntax rules:

[	<<
]	>>
:=	GETS
{	BEGIN
}	END
OPERATION	OP
TRANSFORMER	TR

system

Class: constant expression

Usage: `System`

See Also: copyright, version, nialroot

The expression *System* returns a phrase indicating the operating system in use. Possible values are EXTDOS, Windows or UNIX.

```
     System
Windows
```

take

Class: selection operation

Property: binary

Usage: `A take B   take A B`

See Also: drop, takeright, front

The operation *take* selects a number of items from *B* as indicated by *A*.

If *B* is a list and *A* is a non-negative integer, the result is the list formed from taking *A* items from the front of *B*. If *A* is negative, it takes items from the right end of *B*.

If *B* is a table and *A* is a pair of non-negative integers, the result is the table formed by taking the number of rows and columns indicated by *A* from the upper left corner of *B*. If one or both of the items of *A* are negative, the items are taken from the other end of the axis.

For higher dimensional arrays *B*, the tally of *A* must equal the valence of *B* and the result is obtained by taking from the front or back of the extents along each axis according to the sign of the corresponding item of *A*.

In all of the above cases, if an item of *abs A* is longer than the extent along the corresponding axis in *B*, the corresponding positions in the result are filled with the type of the first item of *B*. If *B* is empty then the result is filled with the fault value *?fill*.

If *B* is a single, the result is an array of shape *abs A*, with all its items equal to the item of *B*.

```
     2 take "Able "Baker "Charlie "Dog
Able Baker

     T1 := tell 2 4
+---+---+---+---+
|0 0|0 1|0 2|0 3|
+---+---+---+---+
|1 0|1 1|1 2|1 3|
+---+---+---+---+


     1 2 take T1
```

```
+---+---+
|0 0|0 1|
+---+---+

     3 -5 take T1
+---+---+---+---+---+
|0 0|0 0|0 1|0 2|0 3|
+---+---+---+---+---+
|0 0|1 0|1 1|1 2|1 3|
+---+---+---+---+---+
|0 0|0 0|0 0|0 0|0 0|
+---+---+---+---+---+

     2 3 take single 3 5
+---+---+---+
|3 5|3 5|3 5|
+---+---+---+
|3 5|3 5|3 5|
+---+---+---+
```

The last example shows that taking from a single repeats the item of the single in every item of the result.

Equations

```
shape A take A = A
tally A = valence B and (and EACH isinteger A) ==> shape (A take B) = list abs A
valence B = 0 and (and EACH isinteger A) ==> A take B = abs A reshape B
```

takeright

Class: selection operation

Usage: `A takeright B   takeright A B`

See Also: take, dropright, last

The operation *takeright* is an obsolete operation that takes items from the ends of extents. It is provided to retain compatibility with earlier versions of Q'Nial.

Definition

```
takeright IS OPERATION A B { opposite A take B }
```

tally

Class: measurement operation

Usage: `tally A`

See Also: shape, valence, grid, list

The operation *tally* returns an integer indicating the number of items of the array at the first level of nesting. This is called the tally of the array. The tally of a single is 1. The tally of a list is the number of items in it. The tally of a table is the product of the number of rows and columns. In general, the tally of an array is the product of its shape.

```
     tally "abc
1
     tally 3 4 5
3
     tally 'hello world'
11
     tally (2 3) (4 5 6)
2
     tally tell 3 4
12
```

Definition

```
     tally IS OPERATION A (product shape A)
```

Equations

```
   tally A = first shape list A
   tally shape A = valence A
   tally tally A = 1
   tally A reshape A = list A
```

tan

Class: scientific operation

Property: unary pervasive

Usage: `tan A`

See Also: arctan, sin, cos, tanh

The operation *tan* implements the tangent function of mathematics. It produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	tangent of the corresponding real
integer	tangent of the corresponding real
real	tangent of angle A given in radians
character	fault ?A
phrase	fault ?A
fault	argument A

```
     tan  1  -1  0.5  `a  "abc  ??error
1.55741 -1.55741 0.546302 ?A ?A ?error
```

Definition

```
     tan IS OPERATION A (sin A divide cos A)
```

Equation

```
   tan opposite A = opposite tan A
```

tanh

Class: scientific operation

Property: unary pervasive

Usage: `tanh A`

See Also: tan, arctan, sinh, cosh

The operation *tanh* implements the hyperbolic tangent function of mathematics. It produces the following results when applied to atoms of the six types:

Atomic Type	Result
boolean	hyperbolic tangent of the corresponding real
integer	hyperbolic tangent of the corresponding real
real	hyperbolic tangent of angle A given in radians
character	fault ?A
phrase	fault ?A
fault	argument A

```
     tanh  1  -1  0.5  `a  "abc  ??error
0.761594 -0.761594 0.462117 ?A ?A ?error
```

Definition

```
     tanh IS OPERATION A (sinh A divide cosh A)
```

Equations

```
   tanh opposite A = opposite tanh A
```

team

Class: applicative transformer

Usage: `TEAM f A`

See Also: each, eachboth, eachall, fold, iterate

The transformer *TEAM* applies the operations of atlas *f* to corresponding items in the list of *A*. There must be the same number of items in *A* as there are operations in the atlas *f*. The result has the shape of *A*. If *f* is an operation that is not an atlas, *TEAM f A* is the application of *f* to *A*.

```
     TEAM [+,*,max,min] (3 2)(4 5)(2 3 4)(2 3 4)
5 20 4 2

     TEAM [+,*,max,min] count 2 2
2 2
2 2

     TEAM second count 5
2
```

tell

Class: array generation operation

Usage: `tell A`

See Also: count, gage, grid, cart

The operation *tell* is used to generate an array of addresses from a shape. For a non-negative integer *N*, it generates the list of integers from 0 to *N - 1*. For a list of non-negative integers, *tell* generates the array of all combinations of tell applied to the items of the list. If *list A* is not a shape, the result is the fault *?shape*.

```
     tell 5
0 1 2 3 4

     tell 3 4
+---+---+---+---+
|0 0|0 1|0 2|0 3|
+---+---+---+---+
|1 0|1 1|1 2|1 3|
+---+---+---+---+
|2 0|2 1|2 2|2 3|
+---+---+---+---+
```

The examples show *tell* being applied to a shape and returning the array of addresses.

Equations

```
tell shape A choose A = A
isinteger A or isshape A ==> tell A = cart EACH tell A
N an integer and (N >= 0) ==> tell (N + 1) = (tell N append N)
tell Null = single Null
tell 0 = Null
tell 1 = solitary 0
tell solitary 1 = solitary solitary 0
```

third

Class: selection operation

Usage: `third A`

See Also: first, second, find, last, pick

The operation *third* returns the third of the items of *A*. *Third* is defined in terms of *pick* and its behaviour is affected by that of *pick*. The third of a triple is its last item. The third of a table is the third item in the list of items in row major order. The third of an array with two or fewer items is the fault *?address*.

```
     third 4 5 6
6

     third tell 3 4
0 2

     third Null
0
```

Definition

```
third IS OPERATION A { 2 pick list A }
```

Equation

```
third list A = third A
```

time

Class: system expression

Usage: `Time`

See Also: timestamp

The expression *Time* returns a real number giving the time in seconds spent in executing Q'Nial since the beginning of the session. On systems permitting multi-processing, the time represents central processor time in seconds.

The expression *Time* is useful for estimating the relative costs of Q'Nial operations in terms of processor time.

times

Class: arithmetic operation

Property: binary pervasive

Usage: `A times B  times A B`

See Also: product, divide, reciprocal, plus

The operation *times* is the same as *product* (and synonym *) except that it enforces the rule that it must be applied to a pair. *Product* is multi pervasive and can add up any number of items.

```
     7 times 9
63

     (2 3 4) times (12 22 33)
24 66 132

     times 2 3 4
?times expects a pair
```

timestamp

Class: system expression

Usage: `Timestamp`

See Also: time

The expression *Timestamp* gives the current date and time in the standard format for the host system. The details of this expression are implementation dependent.

The result is reported as a string giving the date and time. *Timestamp* is useful for dating reports and messages.

```
     Timestamp
Mon Jan 27 14:07:37 1997
```

toend

Class: debugging command

Usage: `toend`

See Also: debugging, break, resume, next, step, stepin

The command *toend* is used in debugging a definition that has been suspended using Break or <Ctrl B>. The effect of *toend* is to execute all the expressions to the end of the current loop or the end of the definition. If there is a loop then execution suspends on the expression following the loop. If the *toend* is issued when not in a loop, execution suspends on the expression being returned from the definition.

tolower

Class: conversion operation

Usage: `tolower A`

See Also: toupper, tonumber, char, charrep, phrase, string

The operation *tolower* is used to ensure that a string has all its letters in lower case. Applied to a string *A*, it results in a string with all the upper case letters converted to lower case leaving all other characters unchanged. It can also be used on a single character.

```
     tolower 'abcDE*?12xyZ'
abcde*?12xyz

     tolower 'STATEMENT IN LOWER CASE'
statement in lower case
```

tonumber

Class: conversion operation

Usage: `tonumber A`

See Also: tolower, toupper, execute, char, charrep, string

The operation *tonumber* converts the string *A* holding a character representation of a number to its corresponding numeric representation.

```
     tonumber '345'
345

     tonumber '37.456'
37.456

     tonumber '123456789012345'
1.23457e+14

     tonumber '1'
1

     tonumber '"abc'
?not a number
```

Equation

```
   Numstr a string representing a number ==> tostring Numstr = execute Numstr
```

top level loop

Class: concept

See Also: toplevel, loaddefs

Nial program fragments are entered during interactive input with a process called the **top level loop**; or brought into the system under the control of a systems operation, *loaddefs*. This systems operation has the effect of loading a sequence of program fragments from a file as though the fragments had been entered interactively in the order they appear in the file.

The global environment is the collection of associations between names and meanings that are known at the top level loop. Such names have global scope in that they can be referenced by any program text. All other names have a local scope that associates a meaning with the name only during execution of a specific portion of a program text.

In direct input at the top level loop, a remark ends at the end of the line unless a backslash symbol (\ ) is used to extend the line. In a definition file, a remark ends at the first blank line. A remark cannot appear within a definition or expression-sequence.

The expressions in an expression-sequence are evaluated in left-to-right order. If the sequence does not terminate with a semicolon, the array returned is the result of the last expression. If the sequence does end with a semicolon, the array returned is the fault *?noexpr*. At the top level loop, if the array returned is the fault *?noexpr*, it is not displayed.

In window mode for a console version of Q'Nial when a window is used interactively from the top level loop, the terminal acts as though it has a screen the size of the window. In particular, as the cursor attempts to move below the bottom line of the window, the text is scrolled one line at a time. The speed of scrolling can be changed using the operation *setscroll*.

topic

Class: system operation

Usage: `topic Name`

See Also: help, symbols

The operation *topic* uses the Q'Nial Help Facility for console versions to display help on the subject given as the argument to the operation. *Name* should be a phrase or string naming a particular subject.

toplevel

Class: system expression

Usage: `Toplevel`

See Also: recover, continue, restart, save, status

The expression *Toplevel* ends the execution of the definition currently being executed and returns control to the top level loop where an operation named *recover* is executed if it is present in the workspace.

The major purpose for *Toplevel* is to handle an exception condition within a definition by forcing the definition to terminate. Use of a *recover* operation makes it possible to recover gracefully in such cases.

toraw

Class: conversion operation

Usage: `toraw A`

See Also: fromraw, tonumber

The operation *toraw* converts the simple array *A* of real number, integers, or characters to a boolean array corresponding to the internal bit pattern for the data.

```
     toraw '345'
ooooooooollololoolloloo

     toraw 345
ooooooooooooooooooooolololloool

     toraw 34.5
ooooooooooooooooooooooooooooooooloooooolooooololooooooooooooo

     toraw lollo
lollo
```

Equation

```
simple A and and (type A match type first A) => A = fromraw (toraw A) (first A)
```

toupper

Class: conversion operation

Usage: `toupper A`

See Also: tolower, tonumber, char, charrep, string

The operation *toupper* is used to ensure that a string has all its letters in upper case. Applied to a string *A*, it results in a string with all the lower case letters converted to upper case leaving all other characters unchanged. It can also be used on a single character.

```
     toupper 'abcDE*?12xyZ'
ABCDE*?12XYZ

     toupper 'a statement in upper case'
A STATEMENT IN UPPER CASE
```

transformer

Class: concept

See Also: expression, operation, definition

A **transformer** is a functional object that is used to construct a new operation from a given operation argument, usually producing a modified version of the given operation. Most transformers used in Nial are provided in the core language. However, there is a mechanism that constructs a named transformer in terms of one or more operation parameters. A user-defined transformer describes the modified operation as a parameterized algorithm for manipulating data.

A transformer usually specifies a general algorithm which can have an operation as a parameter. For example, the *EACH* family of transformers generalizes a number of looping mechanisms for applying an operation to items of arrays.

A user-defined transformer could provide the skeleton for processing the records of a file and allow an arbitrary operation to be applied to each record. Such a transformer is often called a filter.

The process of evaluating an operation call of an operation modified by a transformer requires two steps. The modified operation is formed; and then the modified operation is given the array argument which it uses to produce the result.

```
     TWICE is TRANSFORMER f (f f)

     TWICE rest 4 5 6 7 8
6 7 8
```

transformer form

Class: syntax

See Also: transformer, definition

A **transformer-form** is the syntactic structure used to describe a transformer in terms of an operation expression involving formal operation parameters. The names that follow the keyword transformer in the transformer-form are called formal operation parameters. The body of a transformer-form is the operation-expression which uses these names. The first rule requires that the operation-expression be an operation-form; the second allows any operation- expression to be used.

```
transformer-form ::= TRANSFORMER
   { identifier }+ operation-form
   | TRANSFORMER { identifier }+
     ( operation-expression )
```

The effect of applying a transformer-form to an operation-expression is the effect of an operation formed in the body of the transformer, such that wherever one of the formal operation parameters occurs, it is replaced with the corresponding argument operation-expression.

On the other hand, if the formal operation parameters consist of only one name, the operation formed is associated with the argument operation-expression. If the operation formed has two or more names, the operation-expression must denote an atlas of the same length; and the formal operation parameters are associated with the operations of the atlas in their sequence.

The associations are made with the argument operation-expression in the environment where the transformer is applied. If there is a mismatch between the number of formal operation parameters and the argument, the result of applying the transform is the fault *?tr_parameter*.

transpose

Class: data rearrangement operation

Usage: `transpose A`

See Also: fuse, reverse, pack

The operation *transpose* is used to reverse the axes of a table or array of higher valence. If *A* is an *M* by *N* table, the transpose is the *N* by *M* table whose rows are the columns of *A*. *Transpose* has no effect on a single or a list.

```
     transpose tell 2 3
+---+---+
|0 0|1 0|
+---+---+
|0 1|1 1|
+---+---+
|0 2|1 2|
+---+---+

     A := 2 3 4 reshape count 24
1  2  3  4   13 14 15 16
5  6  7  8   17 18 19 20
9 10 11 12   21 22 23 24

     transpose A
1 13    2 14    3 15    4 16
5 17    6 18    7 19    8 20
9 21   10 22   11 23   12 24
```

Definition

```
     transpose IS OPERATION A ( reverse tell valence A fuse A)
```

Equations

```
   shape transpose A = reverse shape A
   transpose A = reverse axes A fuse A
   transpose transpose A = A
   transpose single A = single A
   transpose list A = list A
```

trs

Class: system expression

Usage: `Trs`

See Also: symbols, exprs, ops, vars

The expression *Trs* returns a list of phrases giving the names of all user defined transformers in the workspace.

Definition

```
     Trs IS {
        Names Roles := pack symbols 0;
        "tr match Roles sublist Names }
```

true

Class: constant expression

Usage: `True`

See Also: false, isboolean

The constant expression *True* denotes the boolean atom for *true*, which Nial also denotes by *1*. It is the result of comparing two identical arrays for equality.

Definition

```
True IS (0 equal 0)
```

Equations

```
tally True = 1
shape True = Null
single True = True
not True = False
max True = True
abs True = 1
```

twig

Class: distributive transformer

Usage: `TWIG f A`

See Also: leaf, each

The transformer *TWIG* transforms an operation *f* into an operation that applies *f* to every simple array in the nested structure of *A*. The resulting operation is called the *TWIG* transform of *f*. The result of applying the *TWIG* transform of *f* to *A* has the same shape as *A*. If *f* maps simple arrays to simple arrays of the same shape, the result has the same structure as *A*.

```
     T1 := count 2 3
+---+---+---+
|1 1|1 2|1 3|
+---+---+---+
|2 1|2 2|2 3|
+---+---+---+

     TWIG tally T1
2 2 2
2 2 2

     TWIG tell T1
+-----+---------+-------------+
|+---+|+---+---+|+---+---+---+|
||0 0|||0 0|0 1|||0 0|0 1|0 2||
|+---+|+---+---+|+---+---+---+|
+-----+---------+-------------+
|+---+|+---+---+|+---+---+---+|
||0 0|||0 0|0 1|||0 0|0 1|0 2||
|+---+|+---+---+|+---+---+---+|
||1 0|||1 0|1 1|||1 0|1 1|1 2||
|+---+|+---+---+|+---+---+---+|
+-----+---------+-------------+
```

The first example shows that the result of applying a *TWIG* transform to a table is a table of the same shape. The items of *T1* are simple and hence have been mapped by tally to 2. In the second example, the structure of the result is not preserved because *tell* maps a pair to a table of pairs.

Definition

```
TWIG IS TRANSFORMER f OPERATION A {
   IF simple A THEN
      f A
   ELSE
      EACH (TWIG f) A
   ENDIF }
```

Equations

```
shape TWIG f A = shape A
f unary pervasive ==> TWIG f A = f A
(TWIG f) (TWIG g) A = TWIG (f g) A
TWIG f list A = list TWIG f A
A is a shape ==> TWIG f (A reshape B) = A reshape TWIG f B
```

type

Class: measurement operation

Property: unary pervasive

Usage: `type A`

See Also: isboolean

The operation *type* maps an atom *A* to the representative value of the corresponding atomic type. It is extended to arbitrary arrays by being unary pervasive.

Atomic type	Representative atom
boolean	o
integer	0
real	0.
character	<blank>
phrase	"
fault	?

```
    set "decor;
    type 1  3  3.3  `3  '3.3'  "33
+-+-+--+--+-----+--+
|o|0|0.|` |'   '|""|
+-+-+--+--+-----+--+
```

The operation *type* is provided as a computational way of transforming an array to a standard value, while preserving structure and type information. Executing a type test predicate, *isboolean*, *isinteger*, etc., is equivalent to testing that the type of an array is equal to the corresponding representative atom.

unary pervasive

Class: operation property

Usage: `f A`

See Also: binary pervasive, multi pervasive, pervasive, each

A **unary pervasive** operation maps an array to another array with identical structure, mapping each atom by the function's behaviour on atoms. All of the scientific operations and the unary operations of arithmetic and logic are unary pervasive.

The scientific operations are implemented using the library routines provided with the C compiler used to construct Q'Nial. The accuracy of the result is determined by the precision of the floating point number system of the computer and the accuracy of the library routine approximation.

The following table describes the unary pervasive operations:

Operation	Function
abs	absolute value
arccos	inverse cosine function
arcsin	inverse sine function
arctan	inverse tangent function
ceiling	lowest integer above a real number
char	integer to character conversion
charrep	character to integer conversion
cos	cosine function
cosh	hyperbolic cosine function
exp	exponential function
floor	next higher integer above a real number
ln	natural logarithm
log	logarithm base 10
not	opposite of a boolean value
opposite	opposite of a number
reciprocal	reciprocal of a number
sin	sine function
sinh	hyperbolic sine function
sqrt	square root of a number
tan	tangent function
tanh	hyperbolic tangent function
type	representative atom of same type

Equations

```
f A = EACH f A
shape f A = shape A
```

unequal

Class: logic operation

Property: predicate

Usage: `A unequal B   A ~= B   unequal A`

See Also: equal, diverse, notin

The operation

```
     2 3 4 ~= [2,3,4]
o

     Null ~= ''
o

     2 (3 4) ~= (2 3) 4
l

     unequal (2 3) (2 3) (2 3)
o
```

Definition

```
     unequal IS OPERATION A { not equal A }
```

up

Class: comparison operation

Properties: binary, predicate

Usage: `A up B   up A B`

See Also: lte

The operation *up* is used to do a lexicographic comparison of two arrays returning *true* if *A* is lexicographically less than or equal to *B* and *false* otherwise. If *A* and *B* are atoms of the same type, then their values are compared using <=, otherwise *A* is viewed as less than *B* if *A* has a lower type than *B*. The types are ordered lowest to highest by *boolean*, *integer*, *real*, *character*, *phrase*, and *fault*.

If one of *A* and *B* is not atomic then the list of items of *A* are compared lexicographically to those of *B*. The comparison is based on the first position where the items differ, and the result is the lexicographic comparison of the two items. If the lists of items agree up to the point where one is exhausted, then the array with the shorter list precedes the longer one.

If *A* and *B* are arrays with the same list of items, then the comparison is decided by comparing their shapes lexicographically.

```
     2 up 1.5
1

     [2] up 3
1

     5 up 3 4 5
0

     tell 10 up tell 2 3
1

     (2 2 reshape 'abcd') up (1 4 reshape 'abcd')
0
```

The first example indicates that 2 precedes 1.5 because the two atoms are of different type and type integer precedes type real. The second example is *true* because the lists of items differ in the first item, and 2 precedes 3. In the third example the lists of items differ in the first item and 5 does not precede 3. In the fourth example, the arrays differ in the first item and the *up* comparison of 0 and 0 0 is *true* because they agree in the first item and 0 is shorter. In the final example, the result is *false* because the items are the same and the *up* comparison of 2 2 and 1 4 is *false*.

Equations

```
(A up B) and (A ~= B) ==> not (B up A)
A = B ==> A up B
(A up B) and (B up A) <==> A = B
```

Definitions

```
sortup IS SORT up
gradeup IS GRADE up
```

update

Class: evaluation operation

Usage: `update Nm I A`

See Also: updateall, deepupdate, place

The operation *update* provides the semantics of the *Nm@I := A* form of assignment expression. *Nm* must be an existing variable represented by a string, phrase or a cast. *I* is the address of the location to be updated. *A* is the array to be placed in the variable.

```
     X := 3 4 5; update "X 0 8
8 4 5

     update !X 2 "goodbye
8 4 goodbye
```

The major purpose of *update* is to allow a selective update at one location in the array associated with a global variable without forcing a copy. By passing the name of the variable to the operation that is using *update*, rather than its value, no sharing of the internal data is made and hence the update can be done in place.

updateall

Class: evaluation operation

Usage: `updateall Nm I A`

See Also: update, deepupdate, placeall

The operation *updateall* provides the semantics of the *Nm#I := A* form of assignment expression. *Nm* must be an existing variable represented by a string, phrase or a cast. *I* is the array of addresses of the locations to be updated. *A* is the array of values to be placed in the variable.

```
     X := 3 4 5;
     updateall "X (0 2) (8 10)
8 4 10

     updateall !X (1 2) ("hello "goodbye)
8 hello goodbye
```

The major purpose of *updateall* is to allow a selective update at several locations in the array associated with a global variable without forcing a copy. By passing the name of the variable to the operation that is using *update*, rather than its value, no sharing of the internal data is made, and hence the update can be done in place.

user primitives

Class: system operation

Usage: `<defined for each primitive>`

See Also: calldllfun, callc

The Nial Tools software packages for Windows and for Unix provide a capability to extend the Q'Nial interpreters provided with each package by adding new user written primitives corresponding to expressions, operations and transformers. The new routines are written in "C" to meet the abstract machine interface for the Q'Nial interpreter.

This requires that the licensee have a C compiler that can produce object modules compatible with the object files provided with the package. The extended interpreter is built by linking the user written routines with object files.

This approach to extending Nial is appropriate if the new primitive is called frequently and one wants to avoid the overhead of using a DLL interface (under Windows) or the *callc* operation (under Unix).

Vacate

Class: reshaping operation

Properties:

Usage: `vacate A`

See Also: reshape, void

The operation *vacate* is a renaming of the operation *(0 reshape)*. It is used in Version 4 of Array Theory to produce an empty array with the same prototype as its argument. In Version 6 of Array Theory it always returns *Null*.

```
     vacate 3 (4 5)= Null
1
```

Definition

```
vacate is OPERATION A ( 0 reshape A )
```

Equations

```
   vacate A = Null
   vacate A = void first A
```

valence

Class: measurement operation

Usage: `valence A`

See Also: shape, tally

The operation *valence* returns an integer indicating the number of axes of *A*. If *A* is a single, list or table, the result is 0, 1 or 2 respectively.

```
     valence 3
0

     valence 4 5 6
1

     valence tell 2 3
2

     valence (0 0 0 0 1 1 0 1 reshape 3)
8
```

Definition

```
     valence IS OPERATION A (tally shape A)
```

Equations

```
shape valence A = Null
tally valence A = 1
```

value

Class: evaluation operation

Usage: `value Nm`

See Also: assign, eval

The operation *value* provides the semantics of *value of a variable* that is implicit whenever a variable is used in a value context. The argument *Nm* is either a string or phrase denoting a variable or it is the cast of a variable. The result is the value of the variable. If *Nm* is a string or phrase, the value is sought in the environment at the point of application of *value*. If *Nm* is a cast, the value is sought in the environment where the cast was created. The major use of *value* is in conjunction with *assign* which allows arbitrary variables to be stored and later retrieved using *value*.

```
     X := 3 4 5 ; value "X = value !X
1
```

variable

Class: syntax

See Also: assign expression, local environment, vars, role, indexing, scope of a variable

A **variable** is a name associated with an array value. Its syntactic form is that of an identifier.

```
     variable ::= identifier

     indexed-variable ::=
          variable @ primary-expression
        | variable @@ primary-expression
        | variable # primary-expression
        | variable | primary-expression
```

A variable is given an association with an array value by its use on the left side of an assign-expression, its appearance in a local or nonlocal declaration, its designation as a variable in an external-declaration or its use as the first argument of the operation assign.

When a variable is used as a primary-expression, its meaning is the array value associated with the identifier. If the variable exists but has not been assigned, it will have as its default value the fault *?no_value*. If an identifier is mentioned as a primary-expression but has not yet been given an association, a parse error will occur with the fault *?undefined identifier:*.

Role of a Variable

A variable gives a name to the result of a computation. If the same result is needed later in the program, the named variable can be used, thereby avoiding the necessity of repeating the computation. A variable

can be assigned different array values throughout the computation.

Although an identifier can be of any length up to 80 characters, a compromise is usually made between choosing explicit variable names and choosing brief names to avoid unnecessary typing. An identifier used as a variable cannot be a Q'Nial reserved word. In a local environment, a variable identifier can be chosen the same as a predefined or user-defined global definition name. Such a choice makes the global use of the name unavailable in the local context.

In any context, an identifier can name only one of: a variable, an array- expression, an operation-expression, or a transformer-expression. During one session, the role of a name, i.e. the class of syntactic object it names, cannot be changed.

Indexed Variable

An **indexed variable** is a variable for which a part of the associated array value is referenced. An **index** is the value of the primary-expression within an indexed-variable which specifies the location or locations of the part or parts of the array that are selected.

vars

Class: system expression

Usage: `Vars`

See Also: symbols, exprs, ops, trs

The expression *Vars* returns a list of phrases giving the names of all user variables in the workspace.

Definition

```
Vars IS {
   Names Roles := pack symbols 0;
   "var match Roles sublist Names }
```

version

Class: constant expression

Usage: `Version`

See Also: copyright, system

The expression *Version* returns a string containing the release and version number of Q'Nial.

```
     Version
Q'Nial (Public Domain Edition) Version 6.3 PC Windows Aug  8 2005
```

Void

Class: reshaping operation

Usage: `void A`

See Also: reshape, vacate

The operation *void* is an obsolete operation that has the same effect as the operation *(0 reshape)*. It is used in Version 4 of Array Theory to produce an empty array with A as its prototype. In Version 6 of Array Theory it always returns *Null*.

```
     void 3 (4 5) = Null
1
```

Definition

```
void is OPERATION A ( 0 reshape solitary A )
```

Equations

```
   void A = Null
   vacate A = void first A
```

watch

Class: system operation

Usage: `watch Var Expr`

See Also: debugging, watchlist

The operation *watch* provides a means to watch when a variable has its value changed. The first argument to *watch* is a variable reference expressed as a cast. For a global variable *X*, the cast is *!X*; for a local variable *Y* in definition *G*, the cast is *!G:Y*.

The second argument is a string of program text to be executed when the variable is changed. If it is empty then it indicates that the watch should be removed. The program text can display the value being set, execute a break, or take some other action.

The result of *watch* is the variable reference and the previous watch expression if any. This can be used to restore the watch setting by storing it and using it as the argument to *watch* at a later time.

```
     X gets count 5;
     watch !x 'write ''X changed to: '' X'
+------------------++
|+---+------------+||
||100|2 7926 42354|||
|+---+------------+||
+------------------++
```

```
     X gets 'abc'
```

```
+--------------+---+
|X changed to: |abc|
+--------------+---+
abc
```

```
     X gets 3 4 5 6
+--------------+-------+
|X changed to: |3 4 5 6|
+--------------+-------+
3 4 5 6
```

```
     foo is op A { B gets count A; reverse B }
```

```
     watch !foo:b  'write ''B in foo changed to: '' B'
+--------------------------------------++
|+---+--------------------------------+||
||100|+--+------------+--------------+|||
||   ||50|2 7926 42470|23 7798 7798 B||||
||   |+--+------------+--------------+|||
|+---+--------------------------------+||
+--------------------------------------++
```

```
     watchlist
+-----------------------------+----------------------------------------+
|+--+------------------------+|+------+------------------------------+|
||!X|write 'X changed to: ' X|||!FOO:B|write 'B in foo changed to: ' B||
|+--+------------------------+|+------+------------------------------+|
+-----------------------------+----------------------------------------+
```

```
     watch !x ''
+------------------+------------------------+
|+---+------------+|write 'X changed to: ' X|
||100|2 7926 42354||                        |
|+---+------------+|                        |
+------------------+------------------------+
```

```
     watchlist
+---------------------------------------+
|+------+------------------------------+|
||!FOO:B|write 'B in foo changed to: ' B||
|+------+------------------------------+|
+---------------------------------------+
```

To clear all watches use:

```
     EACH first Watchlist EACHLEFT watch ''
```

watchlist

Class: system expression

Usage: `Watchlist`

See Also: watch, debugging

The execution of *Watchlist* prints out a list of the variable watches that are in effect.

Its main use is to assist in clearing the watches as debugging proceeds.

```
     Watchlist
+-----------------------------+----------------------------------------+
|+--+------------------------+|+------+-------------------------------+|
||!X|write 'X changed to: ' X|||!FOO:B|write 'B in foo changed to: ' B||
|+--+------------------------+|+------+-------------------------------+|
+-----------------------------+----------------------------------------+

     watch !x ''
+------------------+------------------------+
|+---+------------+|write 'X changed to: ' X|
||100|2 7926 42354||                        |
|+---+------------+|                        |
+------------------+------------------------+

     watchlist
+--------------------------------------+
|+------+-------------------------------+|
||!FOO:B|write 'B in foo changed to: ' B||
|+------+-------------------------------+|
+--------------------------------------+
```

To clear all watches use:

```
EACH first Watchlist EACHLEFT watch ''
```

while-loop

Class: control structure

Usage: `WHILE conditional expression DO expression sequence ENDWHILE`

See Also: repeat-loop, for-loop

The *WHILE-loop* notation is used for executing an expression sequence repeatedly as long as a conditional expression returns *true*.

```
F := open Filenm "r;
Lines := '';
Line := readfile F;
WHILE not isfault Line DO
  Lines := Lines append Line;
  Line := readfile F;
ENDWHILE;
```

window

Class: window mode input/output expression

Usage: `Window`

See Also: setwindow, clearwindow, editwindow

The expression *Window* returns the parameters of the current window, *Position*, *Size*, *Textcolor*, *Cursorcolor*, *Bordercolor* and *Title*. If no window is active, it returns a *Null* result. The parameters returned are:

Argument	Purpose
Position	a pair of integers giving the row and column of the upper left corner of the window in screen coordinates
Size	a pair of integers giving the height and width of the window in units of character positions
Textcolor	a control tuple specifying the attributes of window text
Cursorcolor	the control tuple specifying the attributes of cursor line characters
Bordercolor	the control tuple specifying the attributes of the border characters
Title	a string centered in the top of the window border

write

Class: interactive input/output operation

Usage: `write A`

See Also: writescreen, writechars, read

The operation *write* displays the picture of an array on the display screen. The result is the no-expression fault *?noexpr*. The picture displayed depends on the settings of the *diagram/sketch* and *decor/nodecor* switches. The effect of *write* is the same as applying *writescreen* to the picture of the array.

```
     write 3 (4 5);
+-+---+
|3|4 5|
+-+---+
```

Definition

```
write IS OPERATION A { writescreen picture A }
```

writearray

Class: nial direct access file operation

Usage: `writearray F N A`

See Also: writerecord, readarray, readrecord, eraserecord, writefile, filetally, open, close

The operation *writearray* writes array *A* to the direct access file designated by *F* at component number *N*. If a component already exists at that component number and *A* can fit in the space taken by that component, the existing one is overwritten. If not, the new component is written at the end of the *.rec* file and the *.ndx* file is updated to indicate the starting point for the new component. The space used by the previous value is recorded for a future compression.

The second argument *N* can be a list of numbers, in which case the items of *A* are written to the components designated by the integers in *N*. The file must be open for direct access. The other direct access file operations, *readrecord* and *writerecord*, must not be used on a file that is created using

writearray. However, *eraserecord* and *filetally* are used for both kinds of direct access files.

After a *writearray* or *writerecord*, if the total unused space is a significant amount of the space occupied by the *.rec* file, the file is compressed with the records being placed in order.

The content of an array component is placed in the file in a binary form independent of the workspace in which it was created. Data that is written to a direct access file, erased, and then read in again, may take up more space in the workspace after this process due to lack of sharing of internal representations.

writechars

Class: interactive input/output operation

Usage: `writechars S`

See Also: writescreen, write, readchar

The operation *writechars* writes *S* to the screen at the current cursor position, without supplying a trailing "new line" character. Thus, the cursor is left at the position just after the text. *S* may be a string, a phrase or a single character. The principal use of *writechars* is in applications where the display screen is managed as a fixed object and scrolling must be avoided.

```
writechars 'hello world'
```

writefield

Class: host direct access file operation

Usage: `writefield Filename Start Str`

See Also: writerecord, writearray, filelength

The operation *writefield* is used to write a character string to a portion of an existing host file. The first argument is the file name, followed by the integer offset to the beginning of the field to be written and by the string to be written. The result is the *?noexpr* fault.

Example

```
writefield "phonefile 200 'Mike Smith  389-4444'
```

In the example, *writefield* overwrites the contents of file *phonefile* starting at position 200 by the string given as the third argument.

writefile

Class: file operation

Usage: `writefile N S   writefile N S [Eolsw]`

See Also: readfile, putfile, open, close, appendfile

The operation *writefile* is used to write the character array *S* to the file designated by file designator *N*. The file designator is an integer returned by *open*. The file must have been opened for writing, appending or communication.

The argument *S* may be a character, a string, or an array of characters of any valence. If it is a character or string, a single record is written to the file. If it is a table, the rows of the table are written as records to the file. The appropriate end-of-line indication is added for each record written. If the argument *S* is empty, an empty record is written.

If *writefile* is successful, the result is the no-expression fault *?noexpr*. If the argument is not a character, string or character table, the fault *?not text data* is returned.

If the optional third argument is present, it is used to indicate whether the end-of-line indication should be written. If the third argument is *o* or *0* the indication is omitted; if it is *l* or *1*, the indication is placed in the file. This form of *writefile* is used in communications mode to send information to a device driver on the EXTDOS version.

```
writefile 5 'a line of text';
```

writerecord

Class: nial direct access file operation

Usage: `writerecord F N S`

See Also: readrecord, writearray, readarray, eraserecord, open, close

The operation *writerecord* writes string *S* to the direct access file designated by *F* at component number *N*.

If a component already exists at that component number and *S* can fit in the space taken by that component, the existing one is overwritten. If not, the new component is written at the end of the *.rec* file and the *.ndx* file is updated to indicate the starting point for the new component.

The second argument *N* can be a list of numbers, in which case the items of *S* must be strings and are written to the components designated by the integers in *N*. The file must be open for direct access. The other direct access file operations, *readarray* and *writearray*, must not be used on a file that is created using *writerecord*. However, *eraserecord* and *filetally* are used for both kinds of direct access files.

The data written in the *.rec* file by *writerecord* does not contain any end-of-line indications unless they explicitly are part of the strings being written.

writescreen

Class: interactive input/output operation

Usage: `writescreen S`

See Also: write, writechars, writefile, putscreendata, putstrings, readscreen

The operation *writescreen* is used to write character data to the display screen. The argument *S* may be a character, a string, or a character table. If it is a character or string, a single line is displayed. If it is a table, the rows of the table are written as lines. If the argument *S* is empty, an empty line is displayed. In window mode, the data is written within the bounds of the active window.

The result of the operation is the no-expression fault *?noexpr*. If the argument is not a character, string or character table, the fault *?not text data* is returned.

After the text is displayed, the cursor is placed on the line following the last line written.

```
writescreen  'End of Manual' ;
```

The operation *writescreen* is similar to the operation *writefile*. For most host systems, *writefile* may be used with the *standard output device* to get the same effect as a *writescreen*. The argument *S* may be a character, a string, or a character table.