QNial the Language Definition

Michael Jenkins

01 August 2015

The Language Definition

Version 6.3

August 2006

Nial Systems Limited

Preface

The Language Definition manual consists of nine chapters that describe the Nial language at a technical level and provides summary informa tion on the capabilities of the Q'Nial interpreter.

Chapter 1 introduces some of the concepts and terminology of Nial. *Chapter 2* describes the data objects of the language. *Chapters 3 and 4* cover the predefined operations and transformers in Nial. *Chapter 5* gives a formal description of the syntax and semantics of Nial using BNF style syntax descriptions and natural language descriptions of the meaning of the constructs. *Chapter 6* covers the file input and output operations. *Chapter 7* describes the operations that provide access to the mechanisms of the Q'Nial interpreter. *Chapter 8* introduces the Q'Nial environment. *Chapter 9* describes the debugging and profiling capabilities of the Q'Nial interpreter.

Chapter 1 Introduction

Nial is an acronym for the **N**ested **I**nteractive **A**rray **L**anguage. Nial is a *language* intended for users of computers who wish to describe a computational procedure (often called an algorithm) to solve a particular problem or process data in a desired way. The word *interactive* refers to the idea of having a conversation of sorts with the computer. In essence, the program mer types a fragment of a program into the computer and the computer replies immediately indicating the result of processing that part of the program. The word *array*

refers to a data structure which is manipulated by Nial. An array can be a single number or character, a list of numbers, a table of numbers or almost any other structure desired. The word *nested* refers to the fact that an item of an array can be another array or an operation can contain another operation. The idea is akin to Russian dolls that can be taken apart to reveal a smaller doll inside.

The Nial language was designed by Mike Jenkins and Trenchard More during a collaborating that began in 1979 and continued for many years. The current version is a refinement developed by NIAL Systems Limited based on its experience with the practical use in many application areas.

Q'Nial is a computer program initially developed at Queen's University, Kingston that implements the Nial language. It was used to support research in computer science on programming languages, knowledge-based systems and other topics at Queen's and other universities. NIAL Systems Limited has took over responsibility for the development and marketing of Q'Nial and delivered it to customers in many countries. Mike Jenkins continues to use Q'Nial and has made it freely available through the company web site at www.nial.com.

Q'Nial is intended for use as both a rapid prototyping tool for exploring how to design a program to address a particular problem, and as a rapid development tool to build a program for a known task quickly. It implements a very high level language in that many details of programming are done by Q'Nial without explicit specification. A programmer can think about and work on whole data structures rather than just the individual items that make up the structure. By using a prototyping approach, many of the design considerations in a large program can be resolved before a great investment in programming occurs. As well, users can work with a prototype version of the ultimate system, using a working model that has the screens and some of the functionality of the final system.

The emphasis in Nial on powerful data operations that apply to entire data sets means that Nial programs tend to be 1/5 of the size or smaller of an equivalent C or Pascal program. This makes Nial well-suited for rapid development of data intensive programs.

Language Concepts and Terminology

The programming language Nial and its implementation, Q'Nial, are designed to make the learning of programming concepts easy. In normal use of Q'Nial, a programmer enters a small fragment of a program through the keyboard. Q'Nial interprets the entry and returns the result immediately. This instant response provides feedback on the correctness of the program text. Programs can be entered interactively and a workspace of functional and data objects can be built while exploring how to handle a programming requirement. As well, Q'Nial is

designed to support the creation of programs as text stored in one or more text files.

The Objects of Nial

The major focus of Nial is on the creation and manipulation of data objects which are organised under a structuring scheme known as **nested rectangular arrays** (or **arrays** for short).

An **expression** is a construction used to create arrays. Operations and transformers are functional objects used in the construction of expressions.

When the Q'Nial interpreter evaluates a program fragment that is a Nial expression, it returns an array data structure. Examples of expressions are constants, list constructions, conditional expressions, loops, assignment expressions and operation calls. An expres sion may also be made up of a number of steps which describe an algorithm or procedure for computing an array.

An **operation** is a *functional* object that is given an argument array and returns a result array. An operation is used by **applying** it to an argument; this process of executing an operation by giving it an argument value is called an **operation call**.

An operation can be constructed by defining one or more parameters and giving an algorithm to compute the result in terms of the parameters. There are also program fragments (operation expressions) that construct unnamed operations by composing operations, forming a list of operations, or modifying an operation by use of a transformer.

A **transformer** is a *functional* object that is used to construct a new operation from a given operation argument, producing a modified version of the given operation. Most transformers used in Nial are provided in the core language. However, there is a mechanism to construct a new transformer in terms of one or more operation parameters. A user-defined transformer describes the modified operation as a parameterized algorithm (a **schema**) for manipulating data.

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.

The object construction mechanisms are extended by two naming conventions: **variable assignment** and **definition**. A variable assignment associates a name with an array value dynamically during execution of expressions. A definition associates a name with a program fragment in a static manner.

Chapter 2 Data Objects

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 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 characteris tic of an array or the content of an array. There are two boolean atoms: **true** and **false**, denoted by**l** and**o** respectively. When boolean atoms 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 interpre ted as the arithmetical operation of subtraction. Conversely, a space is required when subtraction is intended and the right argument is a number. 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.

A **character** atom represents a single glyph within the standard character set in use. The notation for a single character in Nial is the grave symbol (‘), preceding the character.

A **phrase** is an atom of literal information in which a sequence of characters is taken as a unit. Phrases are used in the place of character strings when the internal structure of the string does not need to be manipulated. Constant phrases are denoted in Nial text by placing the double quote symbol (") before the sequence of characters. Phrases may also be constructed by applying the operation *phrase* to a character string.

A **fault** is an atom of literal information similar to a phrase, used to identify data concerning extraordinary or erroneous conditions. Constant faults are denoted in Nial by placing the question mark symbol (?) before the sequence of characters. Faults may also be constructed by applying the operation *fault* to a character string. By convention, the text of a fault normally begins with a question mark. For example, *?A* denotes an arithmetic fault and *?L* denotes a fault in a logic operation. To denote*?A* in Nial text ??A is typed.

There are a small number of special fault values *?noexpr, ?eof, ?O* and *?I*. These arise in the execution of Nial expressions to indicate that no value is available for the expression. All other faults can be viewed as error conditions. A fault triggering mechanism is available in Q'Nial to detect when faults other than the four special ones are created. The triggering mechanism is on by default but can be turned on or off under program control or at start up.

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 objects at locations that are positioned relative to a set of directions at right angles to each other. In mathematical terminology, an array is said to have a set of orthogonal axes. 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 number of axes of an array is referred to as its dimensionality. The number of axes is the number of dimensions. In mathematical terminology this dimensionality is called the **valence** of the array. The following terms describe arrays by their valence:

Table 1: **Valence Table**

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

The concept of a structure with no directions is, at first glance, paradoxical. If a structure has no directions, how can it have a location? However, allowing a structure to have no directions permits atomic data such as an integer or a

character to be treated as a data structure in a manner consistent with the treatment of structures having one or more directions.

The rectangular structure of an array can be des cribed by its length in each direction. The list of these lengths is called the **shape** of the array. The *shape* of an array has one item for each direction and hence is of length equal to the dimensionality, or *valence*, of the array. The *shape* of an array is itself an array.

The array displayed above is a table that has, as its shape, the array:

```
+-+-+
|4|6|
+-+-+
```

The shape of an array with no axes, a **single**, is the predefined empty list **Null**. All atoms are defined to be singles and hence have *Null* as their shape.

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. By convention in mathematics, the product of the empty list *Null* is one; thus, a single always contains exactly one item. An atom is characterized by the fact that it is self-containing, that is, an atom is its own item.

Lists, arrays with one dimension, are encountered frequently in Nial. The following terminology is used for the most common ones:

Number of Items In the List	Description
0	empty list
1	solitary
2	pair
3	triple
4	quadruple

The shape of a list is a solitary list holding the length of the list. The notation [A, B,. . .] is used to denote a list whose items are A, B, . . . etc. Thus, the expression [4, 6] is a pair and its shape is the solitary list [2].

A solitary and a single both have one item. They differ in rectangular structure in the following way:

	single	solitary
valence	0	1
shape	Null	[1]
tally	1	1

Two forms of lists are also classified by their content. A **string** is a list of characters. A **bitstring** is a list of booleans.

A **location** in an array is described by a list of integers. Each item of the list of integers indicates the position of the location along the corresponding axis. The list of integers is called the **address** of the location in the array. The counting scheme uses zero origin indexing. Thus, in the table above, the address of 496 is [2,3]. For simplicity, the addresses of a list are integers: 0, 1, etc. For convenience, in situations where an address or a shape of a list is expected, an integer and the solitary holding the integer are treated equivalently. The operation *gage* is used to convert a solitary list of integers to an integer and returns any other list of integers directly.

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. The grid of a list is a list of integers. The grid of table Tabove is the following table of pairs of integers.

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

Nesting Structure

The arrays of Nial are a **recursive data type**, that is, the items of an array are also arrays. The above example of the grid of a 4 by 6 table is an example of a nested array. Each of the items is a pair, a list of length two, of integers.

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 the **site** of a data object at some depth within the array. For example, the path [[2, 5], 1] is the path to the second item in the pair at address [2, 5] in the grid of T shown above.

An array is said to be **simple** if all its items are atomic. The table of numbers, T above is simple but the table of its grid is not.

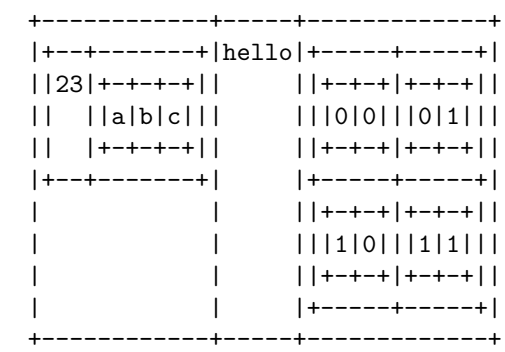
A **part**of an array is a data object that is contained at some level within the array. The atomic parts of an array are called the **leaves** of the array. The simple parts are called **twigs**.

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 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.

```
[ 23 'abc', "hello , tell 2 2 ]
```



Empty Arrays

An **empty** array is one that has no items. The most commonly encountered empty array is the empty list; it is the shape of a single. The empty list can be denoted in three ways in Nial: by the predefined symbol *Null*; by the list constructor notation with no items; and by the string notation with no characters. This is summarized by:

Construction	Denotation
Predefined name	Null
List Constructor	[ ]
String Notation	”

Figure 2-6 Ways of Denoting the Empty List

An empty array may have length in some directions. For example, an empty table of shape [2,0] is one with two rows but no columns. Two empty arrays are equal only if their shapes are equal. By definition, an empty array is simple.

Array Diagrams

The result of a Nial expression is indicated by displaying a **picture** of the result. An array picture is a table of characters that shows the content and structure of an array. The structure of the array is displayed by laying out pictures of the items in a two-dimensional format with or without a frame. The size of the layout is adjusted to leave room for the size of the pictures of the items, which may themselves be pictures of arbitrary complexity.

The form of the picture display is controlled by two independent mode switches. The first switch has two settings, **diagram** and **sketch**. The picture of an atom shows the atom itself in either sketch or diagram mode. **Diagram**mode shows the complete nesting structure of an array. It provides a frame around each item. **Sketch**mode eliminates the frame around the items of a simple array.

The other switch has two settings, **decor** and **nodecor**. When decor is set, characters, phrases and faults are decorated so that the type of data is obvious. Once a mode is set, it remains in use until it is reset.

The picture of an array of three or more dimensions is displayed by a tabular layout of the pictures of the two-dimensional subarrays along the last two dimensions. An array with three axes is displayed as three tables across the page with space between each table. An array with four axes is displayed as a table of pictures of the two-dimensional subarrays with space between each table. As the dimensionality increases, the process of alternating the direction of display continues with increasing space between the pictures of lower dimensional arrays.

The sketch displays of an array of integers of shape 2 3 2 3 and of a 2 3 table of 2 3 tables are:

```
a := 2 3 2 3 reshape count 36

1   2  3  7  8  9 13 14 15
4   5  6 10 11 12 16 17 18
19 20 21 25 26 27 31 32 33
22 23 24 28 29 30 34 35 36


2 raise A

+--------+--------+--------+
```

```
|  1 2 3 |   7 8 9|13 14 15|
|  4 5 6 |10 11 12|16 17 18|
+--------+--------+--------+
|19 20 21|25 26 27|31 32 33|
|22 23 24|28 29 30|34 35 36|
+--------+--------+--------+
```

A single has no axes and hence cannot be diagrammed in a conventional way. If a non-atomic array is a single, its picture has the character o in the upper left corner.

```
single 1 2 3

o-----+
|1 2 3|
+-----+
```

The diagram of a solitary has the usual corner box character in the upper left corner.

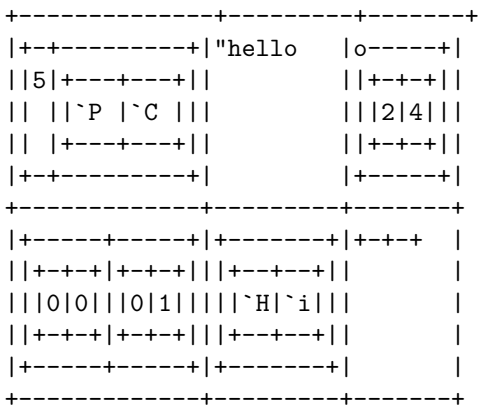
```
solitary 1 2 3

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

In sketch mode, the display of an empty array is suppressed. In diagram mode, the display of an empty array shows the frame border. For example, the display of an empty list is the left border of the frame diagram for a list.

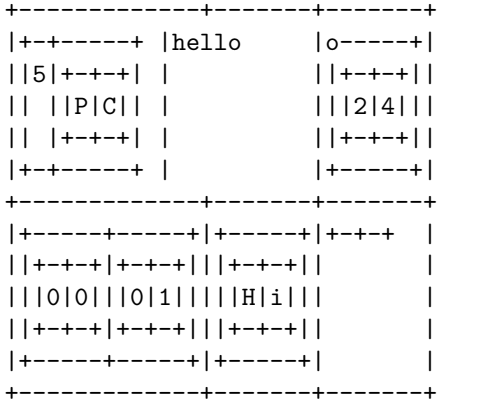
```
set "diagram; Null

+
|
+
```

The display of an empty table is the upper left corner of the frame and if the table has a non zero length, the display includes the frame along one side. The array A, defined below, is displayed in the four mode combinations as follows:

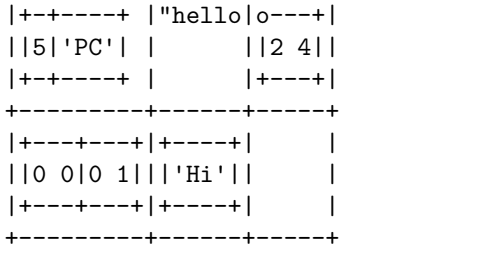
```
A := 2 3 reshape [5 'PC', "hello, single 2 4, tell 1 2, solitary 'Hi', tell 0 2];

set "diagram; set "decor; A
```




```
set"diagram; set "nodecor; A
```




```
set "sketch; set "decor; A
```



```
set "sketch; set "nodecor; A

+---------+-----+-----+
|+-+--+ |"hello |o---+|
||5|PC| |       ||2 4||
|+-+--+ |       |+---+|
+---------+-----+-----+
|+---+---+|+--+ |     |
||0 0|0 1|||Hi| |     |
|+---+---+|+--+ |     |
+---------+-----+-----+
```

Chapter 3 Predefined Data Operations

Programming in Nial is mainly done by creating operations that carry out data transformations from some input data to desired output data and then combining these using the expression mechanisms to find the solution to a given problem. The programming task is accomplished by making use of the predefined operations of Nial.

This chapter gives a brief description of the data operations provided with Q'Nial, grouped by topic area. Detailed descriptions of the operations, organized alphabetically, are given in the *Nial Dictionary*.

Most of the predefined operations are directly implemented in the Q'Nial interpreter. However, a few of them are written in Nial itself and are defined in a file that can be modified by a Q'Nial user. The file *defs.ndf* in the *initial* directory holds these definitions.

Properties of Data

A **binary operation** is one that must have a pair of arrays as its argument. If the argument is not a pair, a fault is returned.

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.

An example of a **unary pervasive** operation is *abs* which returns the absolute value of a numeric atom. When a unary pervasive operation is applied to a non-atomic argument, it returns an array of the same structure as the argument. The atoms of the result are the result of applying the operation to the corresponding atoms of the argument. A unary pervasive operation f applied to a non-atomic array A gives the same result as applying f to each item of A. That is,

```
f A = EACH f A

abs -3.5 27 -8
3.5 27 8
```

An example of a **binary pervasive** operation is *minus*, also denoted by -, which returns the difference of two numbers.

Consider two nonatomic arrays, A and B, of identical structure. When a binary pervasive operation such as *minus* is applied to them as in A *minus* B, the resulting array C has the same structure as A or B. C has atoms that are the result of applying *minus* to the pairs of atoms in corresponding positions of A and B.

```
3 5 7 - 2 8 3
1 -3 4
```

A binary pervasive operation f satisfies the equation

$A\ f\ B = A\ EACHBOTH\ f\ B$

A unary operation which reduces a simple array to an atom is **multi pervasive**.

When a multi pervasive operation, such as *sum*, is applied to an array A with items all of identical structure, it returns an array of that same structure. The atoms of the result are computed by applying *sum* to the simple arrays formed by taking the atoms from positions in the corresponding items of the argument.

Thus, for example, if T is a table, *sum rows* T returns the column totals for T.

```
T := 2 4 reshape count 8

1 2 3 4
5 6 7 8


rows T

+-------+-------+
|1 2 3 4|5 6 7 8|
+-------+-------+


sum rows T

6 8 10 12
```

A multi pervasive operation used on a pair behaves in precisely the same manner as a binary pervasive operation.

A multi pervasive operation f satisfies the equations

$f\ A\ =\ EACHALL\ f\ A$
$f\ A\ =\ REDUCE\ f\ A$

In binary pervasive and multi pervasive operations, the items of the argument must all be of the same shape and structure. If they are not, items that have only one item, (eg. atoms, singles and solitaries), are reshaped to the common shape. If all items with more than one item are not the same shape, the fault *?conform* is returned.

A **predicate** is an operation that tests a condition and returns *true* if the condition holds and *false* otherwise. A predicate is not pervasive.

Logic Operations

A **logic** operation is one that combines arrays of booleans or produces a result of booleans. The predefined logic operations follow:

A = B *true* if A and B are the same array value; *false* otherwise

A ~= B *true* if A and B are not the same array value; *false* otherwise

= A *true* if all the items of A are equal; *false* otherwise,

diverse A *true* if no two items of A are the same array value; *false* otherwise

and A *true* if all the items of A are *true*; *false* otherwise

or A *true* if at least one item of A is *true*; *false* otherwise

not A the opposite boolean from A

The operations *equal* and *unequal* are synonyms for = and ~= respectively. The infix use of *equal* is a special case of the prefix use that tests if all the items of an array are the same. The operation *diverse* is the opposite test. The operations *and* and *or* are multi pervasive; *not* is unary pervasive. They provide the fundamental operations of boolean logic in a very general setting.

Arithmetic Operations

The *arithmetic* operations include elementary and modular arithmetic operations and operations for real number computations in science and engineering.

Arithmetic formulae may be applied to single numbers or to nested collections of numbers. There is also a random number generation operation. The arithmetic operations are given below by class of pervasiveness:

Unary Pervasive	Binary Pervasive	Multi Pervasive
opposite, sin	plus	sum +
reciprocal, cos	minus -	product *
floor, tan	times	
ceiling, arcsin	divide /	
sqrt, arccos	mod	
abs, arctan	quotient	
exp, sinh	power	
ln, cosh		
log, tanh		

All of the above operation names and syonyms represent the corresponding mathematical functions where *sqrt* is square root, *abs* is absolute value, *exp* is exponent, *ln* is natural logarithm, *log* is logarithm to the base 10 and *mod* is modulus.

The standard programming symbols for arithmetic can be used. The symbol + is a synonym for *sum*. It can be used both in a prefix way to sum an array of numbers or in an infix way to add two numbers together. Since *sum* is pervasive, its meaning is extended to do itemwise additions.

The symbol - must be used with care in Nial. It is a synonym for *minus* and can form part of a negative number. When used as *minus* it must be separated from a following number by at least one space to avoid ambiguity.

The operation *opposite* reverses the sign of a number.

Linear Algebra Operations

Q'Nial supports the following linear algebra operations; as well, *inv* is provided as a synonym for *inverse* and *ip* is one for*innerproduct*:

inverse A The inverse of square matrix A

innerproduct A B The inner product of A and B.

solve A B The values of X that satisfy the set of linear equations $AX=B$

Comparison Operations

The **comparison** operations compare numeric or literal atoms according to an implicit ordering. All the comparison operations are pervasive. They are given below by class of pervasiveness:

Binary Pervasive	Multi Pervasive	Non Pervasive
< <=	max	up
> >=	min	
match		
mate		

The operations *gt*, *lt*, *gte* and *lte* are synonyms for >, <, >= and <=, respectively.

The comparison operations are defined for all atomic types. For numeric types, the comparison reflects the usual ordering of the numbers. For characters and phrases, the comparison is made by using a predefined collating sequence. For phrases and faults, the comparison is judged on the first character that differs.

Comparisons between numeric types and nonnumeric types are defined to be *false*, as are comparisons between phrases and single characters.

The operations *<, <=, >, >=,* and *mate* convert numeric arguments to be the same type before doing the comparison. The operation *match* does an equal comparison of atoms without conversion whereas *mate* does an equal comparison with conversion.

The operations *max* and *min* find the upper and lower bounds of the items of the argument, respectively. Special faults are returned to indicate the top (*?I*) and bottom (*?O*) of the pre-lattice of atomic objects if there are no items or if the items are incomparable.

The operation *up* defines a total ordering on the arrays of Nial. It compares valence, then length of axes in order, and then items. Atomic items are compared by type, and if of equal type by *lte*. Non-atomic items are compared by applying *up* recursively.

Type Testing Operations

The operation *type* is unary pervasive and returns a representative atom corresponding to its argument type as defined by the following table.

Representative

Atomic type	Atom	Predicate
boolean	o	isboolean

Atomic type	Atom	Predicate
integer	0	isinteger
real number	0.	isreal
character	<blank>	ischar
phrase	“”	isphrase
fault	??	isfault

The operation *type* can be used to test whether an atom is of a particular type. For convenience a predicate is provided for testing each atomic type directly. For convenience, there is an additional predicate operation, *isstring* to test whether or not an array is a string.

Set-like Operations

There are two set-like operations defined in *defs.ndf*.

allin A B Tests whether or not all items of A are in B.

like A B Tests whether or not all items of A are in B and vice versa.

Conversion Operations

A common requirement in programming is to convert data from one representation to another. For example, numbers are read from a file in literal form as a string but are required in numeric form for arithmetic calculations. Alternately, numbers are converted to a string representation before they are written to a file.

The **conversion** operations return a result of the type specified by the operation name.

Operation	From	To
string	atom	string
phrase	string	phrase
fault	string	fault
quiet_fault	string	fault
char	integer	character
charrep	character	integer
tonumber	string	number
tolower	string	lower case string
toupper	string	upper case string
toraw	simple	bitstring
fromraw	bitstring	simple
gage	array	simple integers

Operations *toupper* and *tolower* take an atomic character or a string argument and return one in which the letters have been changed to upper or lower case respectively. The operations *char* and *charrep* are unary pervasive. The operations *toraw* and *fromraw* are used to convert simple arrays of characters or numbers into bitstrings in order to do low level bit manipulation. The operation *gage* is used to remove unnecessary structure from an array representing a shape or address.

Structure Testing Operations

The following table describes operations that test the structure of an array. The test that an array is simple always holds if an array is atomic or if it is empty.

atomic A A is an atom.

simple A All items of A are atoms.

empty A A has no items.

Measurement Operations

The following table describes operations that measure the structure of an ar ray. They can be used to construct other tests on the structure of an array.

tally A number of items in A

valence A number of axes that A has

shape A list of the lengths of axes of A

axes A list of the axis numbers for A

Array Construction Operations

The following table describes several operations that **construct** an array from one or more components. These include the unary operations, *single* and *solitary*, that add a level and enclose an argument array as an item, the binary operations, *pair*, *hitch* and *append*, that join two arrays into a new structure, and four operations, *link*, *cart*, *laminate*, and *catenate* that combine an arbitrary number of arrays into a new structure. The operations *link* and *cart* are unary, but are often used in infix mode to combine two arrays.

single A single holding A as its item

solitary A solitary holding A as its item

A pair B list of 2 items having items A and B

A hitch B list having A as the first item and the items of B as the remaining items

A append B list with items of A as the beginning items followed by B as the last item

link A list with items of items of A in orde

cart A array of all arrays of the same shape as A taking one item from each of the items of A

I laminate A merged items of A along a new axis before axis I of the items

I catenate A items of A joined along axis I

Reshaping Operations

The following table describes the operations that preserve the level structure and ordering of items of an array, but possibly alter the shape. The operation *reshape* is used both for providing shape and for replicating items if there are not enough to fill out the desired shape. The operation *pass* serves as the identity operation on arrays and is provided for mathematical completeness.

A reshape B array having shape A and items taken from B cyclically

list A list of items of A

pass A A, unchanged

post A table with tally A rows and one column holding the items of A

Array Generation Operations

The operations that are used to generate arrays are described in next.

tell A array of addresses for an array of shape A in 0 origin

count A array of shape A equal to *1 + tell A*

grid A array with the same shape as A holding the addresses of A

random S An array of uniformly distributed random real numbers between 0.0 and 1.0 of shape S. The value returned depends on the current value of the internal seed.

seed A Sets the random number seed to A, a real number between 0. and 1.

The operations *tell* and *grid* generate arrays of addresses. The operation random is used to generate pseudo random numbers. A seed number is initialized when a Nial session is started and subsequent numbers are generated using a linear congruential method. The operation *seed* can be used to reset the generator.

Selection Operations

Many of the operations involve the **selection** of an array from a given array according to some control argument. The selected array may be an item, an array of items of the given array or an array deep in the nesting structure of the array. The selection operations are described below.

The operations *take* and *drop* use a negative value to indicate that the right end or bottom parts of the array are to be taken or dropped. Earlier versions had operations *takeright* and *dropright* that defined in *defs.ndf* for compatibility.

first A first item of A

second A second item of A

third A third item of A

last A last item of A

A pick B item of B at address A

A choose B array of the same shape as A with items selected from B according to addresses A

A reach B part of B that is reached by succes sive *pick* operations using the items of path A

A take B corner of B with shape *abs* A chosen by signs of items of A

A drop B corner of B left after dropping extents of length *abs* A from the ends speci fied by the signs of items of A

A sublist B list of items of B selected according to the boolean pattern A, where *true* indicates selection

A except B list of items of A not in B

front A list of items of A excluding the last

rest A list of items of A excluding the first

cull A list of unique items in A

A cut B list of lists of items of B separated at positionscorresponding to *true* values in A, eliminating items of B inpositions corresponding to *true*

A cutall B list of lists of items of B separated at positions corresponding to *true* values in A, with all items of B retained in the result

Insertion Operations

The operations *pick, choose* and *reach* select an array from a given array using addresses. The corresponding operations that insert items according to addresses are *place, placeall* and *deepplace*.

A B place C array that is the same as C except that the item at address B is replaced by A

A B placeall C array that is the same as C except that items at addresses B are repla ced by items of A

A B deepplace C array the same as C except that the part at path B is replaced by A

The notations described in Chapter 5 provide an alternative way to achieve selection and insertion in arrays.

Searching Operations

The table below describes operations that locate one array in another. The operation *seek* combines the work of *find* and *in* since both can be done with one search. There are two search algorithms used internally. If B has been sorted with *sortup*, a binary search is done; otherwise a linear is search is used.

A find B address of the first position in B that holds A; if A is not an item of B, then *gage shape B*

A in B *true* if A is an item of B; otherwise*false*

A notin B *true* if A is not an item of B; otherwise*false*

A seek B pair with first item A *in* B and second item A *find* B

A findall list of addresses of all positions in B that hold A

Nesting Restructuring Operations

The next table describes operations that add or remove levels to an array by splitting or merging arrays along axes. The operations *rows*, *cols*, *raise* and *lower* are all special cases of *split*. The operation *mix* is a special case of *blend* in which the axes of the items are placed at the end of the axes of the result. The operation *content* removes all nesting structure from the array, returning the atoms at the leaves as a list.

rows A rows of A

cols A columns of A

mix A array formed by merging the first two levels of A, placing the axes of the top level first

A raise B array containing arrays of items of B partitioned along axes so that the result has the first A axes of B

A lower B array containing arrays of items of B partitioned along axes so that the items have the last A axes of B

A split B arrays of items of B where the axes mentioned in A become the axes of the items of the result

A blend B array of valence equal to *valence of* B plus the *valence* of an item of B obtained by merging *axes* of B according to axis numbers in A

content A A list of atoms of A in depth-first row-major order

Data Rearrangement Operations

The table below describes the operations that involve the rearrangement of items within an array. The operations *reverse* and *rotate* work on the list of items of the array, leaving the result in an array of the same shape as originally given.

The operation *transpose* interchanges items in the array to produce the array with the axes in reverse order. It is a special case of the *fuse*, which does a reordering and possible fusion of axes according to the axis numbers in its left argument. If the argument is a nested array, items with 2 or more axis numbers indicate that the corresponding diagonal elements along those axes should be selected.

The operation *pack* interchanges the top two levels of its argument. Thus, if its is given a pair of triples the result is a triple of pairs.

reverse A array of the same shape as A with the items of A in reverse order

N rotate A A with items rotated N places. If $N > 0$, the items are rotated to the left; if $N < 0$, the items are rotated to the right.

transpose A array formed by reversing axes of A

A fuse B array formed from B with the axes reordered according to permutation A. Non-atomic items in A specify axes to be merged by diagonalization.

pack A array formed by interchanging the top two levels of A after replicating items with one item to the same shape as the other items

String Manipulation Operations using Regular Expressions

Regular expressions are string patterns commonly used to describe a search in a string for substrings that have some desired property. The first table below describes the notation for regular expressions used by some of the string manipulation operations described in the second table.

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>	matches itself
\<char>	matches the character

regexp R S Applies the regular expression R to string S and returns a list. The first item is a boolean indicating if the search succeeded. The second item is the first substring matched by R and the remaining items, if any, are the substrings that match the subgroups indicated by parentheses in R.

regexp_match R S [O] Applies the regular expression R to string S and returns a boolean indicating if the search succeeded. If O is ?i then the search is case insensitive.

regexp_substitute R S T [O] Applies the regular expression R to string T doing substitutions based on S. Return the modified string if the search matches; otherwise return T. O provides options for case insensitive search (?i) or for doing all substitutions (?g).

string_split C S [N] Splits the string S whenever one of the character in C occurs in S, eliminating the character. N indicates a limit on the number of substrings obtained.

string_translate C D S [O] Translates characters in string S based on mapping characters in C to the corresponding ones in D. Options include (?d) to delete characters and (?c) to complement characters and (??s) to squeeze many occurrences to one.

The string manipulation operations are based on a package of public domain C++ routines developed by Henry Spencer at the University of Toronto. The regular expressions obey the standard syntax rules for regular expressions used in Perl and Unix software.

Chapter 4 Predefined Transformers

The expressiveness of using operations to achieve a solution to a computational problem is enhanced by the use of transformers or operation modifiers. A **transformer** maps a predefined or user defined operation to a related operation. It is said to be a *second order function* because it has an operation (a first order function) as both its argument and its result. This chapter provides brief descriptions of the transformers or operation modifiers that are built into Q'Nial. More complete descriptions of the predefined transformers, along with examples are found in the *Nial Dictionary*.

Each and Related Transformers

The most important transformer is *EACH*. It modifies an operation so that the operation is applied to the items of an array rather than the array itself. The result is an array of the same shape as the array to which the modified operation is applied. There is a family of transformers related to *EACH* in that they apply the argument operation to arrays formed from parts of the argument. The transformers *EACHLEFT*, *EACHRIGHT* and *EACHBOTH* produce binary operations. They are described in the following table:

EACH f A items of A

A EACHLEFT f B pairs formed from items of A and array B

A EACHRIGHT f B pairs formed from array A and items of B

A EACHBOTH f B pairs formed from items of A paired with corresponding items of B

EACHALL f A arrays of the same shape as A formed from corresponding items of items of A

TWIG f A deeply nested simple arrays of A

LEAF f A deeply nested atoms of A

Partitioning Transformers

The transformers described below apply their argument operation to partitions of an array and then build an array by using the results of the applications as partitions of the result. The transformer *RANK* is primitive and the others are defined from it in *defs.ndf*. The last two transformers do a reduction of the given operation to the partitions.

N RANK f A items of *N lower A; mix*

[A,B] PARTITION f C items of *A split C*; *B blend*

BYROWS f A items of *rows A; mix*

BYCOLS f A items of *cols A; 1 blend*

REDUCEROWS f A reduce *items of rows A; mix*

REDUCECOLS f A reduce *items of cols A; 1 blend*

Applicative Transformers

There are a number of transformers described below that control how an operation or an atlas of operations is applied.

A CONVERSE f B result of *B f A*

N FOLD f A apply *f* to *A*; and then to the result, etc. *N* times

TEAM f A operations in atlas *f* are applied to items in corresponding positions in *A*; if *f* is an operation other than an atlas, it is applied directly to *A*

OUTER f A apply*f* to the items of *cart A*, the array of all combinations of items of items of *A*

A INNER [f,g] B generalized inner product using a reductive operation *f* to reduce each list formed from the outer product of the binary pervasive operation g applied to the
rows of *A* and the columns of *B*

The transformers *OUTER* and *INNER* generalize the outer and inner product concepts from linear algebra. The expression *A OUTER * B*, where *A* and *B* are numeric vectors, produces the matrix formed by multiplying each item of *A* with each item of *B*. *A INNER [+,*] B* does matrix multiplication if *A* and *B* are conformable matrices.

Sorting Transformers

The sorting of data objects depends on a comparison operation that describes the desired ordering relation on the items. The operation *up* provides a total ordering for the universe of arrays; the operation *lte* (<=) provides a partial ordering of the universe of atoms.

The transformer *SORT* expects as its argument a comparator that will partially or totally order the universe of the items to which the resulting operation will be applied. When the modified operation is applied the result is an array of the same shape as the argument with the items in order according to the comparator.

The transformer *GRADE* is similar to *SORT* except applying the modified operation returns a permutation array which, if used to *choose* the argument, will result in the sorted array.

SORT f A array of *shape A* of the items of *A* sorted according to *f*. If *f* is <=, the items are returned in ascending sequence

GRADE f A indices of the items of *A* in the sequence such that choosing the items according to the sequence of the indices returns the items sorted according to *f*

The operations *sortup* and *gradeup* are defined in *defs.ndf* as abbreviations for *SORT up* and *GRADE up* respectively.

Reduction Transformers

The reduction of a list by an operation is the same as placing the operation between each pair of items in a list and then evaluating the resulting expression. The transformer *REDUCE* applied to a binary operation *f* implements this concept. Thus,

REDUCE f [A, B, C] = A f (B f C)

where the parentheses indicate that the grouping is to the right. For an associative operation, i.e. an operation *f* such that

(A f B) f C = A f (B f C)

the result of applying the operation between items is the same regardless of whether the grouping is to the left or to the right. However, for a non-associative operation, the result varies depending on which way the grouping is done. Thus, *REDUCE* is a right grouping reduction.

A related transformer is *ACCUMULATE*, which forms right grouping reductions of the initial lists of an array. For an associative operation*f*, the amount of work needed to compute

ACCUMULATE f [A, B, C]

can be lessened by using the fact that the reductions can be done with left grouping and each item in the result can be formed by one application of f using the preceding reduction value.

The operations *sum, product, and, or, max, min* and *link* are reductive by definition and are not affected by applying a reduction transformer to them. They are all associative operations when applied as a binary operation.

REDUCE f A reduce A applying f in a right-to-left order

ACCUMULATE f A accumulate A using right-to-left reductions

Control Structure Transformers

One of the interesting features of Q'Nial is that it allows the user to choose a style of programming appropriate to the problem. For some purposes, an imperative style using the control structure expres sions is appropriate while for other purposes a more functional approach is simpler. Q'Nial integrates these two styles by providing transformers that have the same result as two of the control structure mechan isms. *ITERATE* is equivalent to the for-loop and *FORK* is equivalent to the if-expression.

ITERATE f A f is applied to the items of A in row major order and the result of the last application is returned

FORK [f₀, f₁ . . . fₙ] A For even values of i less than or equal to n, $f_i\ A$ is applied sequentially until one of the results is *true*; then $f_{i+1}\ A$ is returned. If no application of $f_i\ A$ evaluates to *true*, then, if n is even, $f_n\ A$ is returned; otherwise the fault *?noexpr* is returned

Selection Transformer

There is one additional built in transformer to select items of an array based on a predicate.

FILTER f A f is applied to the items of A and the resulting boolean array is used to select items of A using sublist

Chapter 5 The Formal Description of Nial Programs

Nial is a programming language specifically designed for use in an interactive environment. The formal description of the language given in this chapter

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 inter actions 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 correct ness of English. Nial uses a small number of syntactic constructs that can be grouped into the following classes:

- juxtapositional forms
- list forms
- control structure expressions
- parameterized functional forms
- indexing forms
- assignments
- definitions
- declarations

These are combined using punctuation symbols and spacing to build program fragments that can be processed by the Q'Nial interpreter. The meaning of a program fragment, its **semantics**, is explained in terms of three classes of objects:

arrays: the data objects

operations : functional objects that map arrays to arrays

transformers : functional objects that modify operations

The explanations are given in relationship to the global environment, which is the set of associations between names and objects formed earlier in the session; and in terms of how the objects are combined.

In the remainder of this chapter, the syntax of Nial is described using an extended Backus-Naur Form (BNF) formalism using the following rules:

- Symbols that appear in program fragments are in **bold text**.
- A single optional form is enclosed in brackets [ ].
- A choice between options, where one must be chosen, is written in parentheses ( ) with a vertical bar | separating choices.

- Alternative rules for a construct are separated by the vertical bar |.
- Fragments that can be repeated are enclosed in curly braces {}, where {} means repeated zero or more times, and {}+ means repeated one or more times.

  In the following sections, the meaning given to a rule is given in a description following the rule.

Environment

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 operation has the effect of loading a sequence of program fragments from a file. The effect is as if 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 objects 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.

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 encloses another. A local association is not visible outside the construct in which it is defined; and a name with local scope hides 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 any environment, a name can have only one **role**: either a variable, an expression name, an operation name, a transformer name, or an identifier.

In program text, the scope of all names is determined by the static structure of the program text. The one exception is program text in a string that has the operation *execute* applied to it under program control. Dynamic execution of program text is described in Chapter 8.

Action

An **action** is the construct that is entered in the interactive loop of the Q'Nial interpreter or accepted as a group of lines by the operation *loaddefs*:

<action> ::= *<definition-sequence>*
| *<expression-sequence>*
| *<external-declaration>*
| *<remark>*

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 treated as commentary input and is ignored.

Definition

<definition-sequence> ::= *<definition>* {**;** *<definition>* } [**;**]

<definition> ::= *<identifier>* **IS**
(*<simple-expression>* | *<operation-expression>* | *<transformer-expression>*)
| *<comment>*

Every well formed program fragment that can appear on the right of the keyword **IS** in a definition is interpreted to be one of the three kinds of semantic objects in Nial: an array, an operation or a transformer.

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.

The name is formed according to the rules for specifying an identifier:

<*identifier*> ::= <*letter*> { (<*letter*> | <*digit*>) }

<*letter*> ::= **&** | **_** | **a** | **b** | . . . | **Y** | **Z**

<*digit*> ::= **0** | **1** | **2** | . . . | **9**

In Q'Nial, identifiers are limited to 80 character positions in order to facilitate the reconstruction of program text in the canonical form used in *see* and *defedit*. They may be entered in upper or lower case. Internally, Q'Nial translates the letters of an identifier to upper case so that Data, data and DATA would all be the same identifier. An identifier displayed in canonical form is presented in upper or lower case appropriate to its role. An expression or a variable is displayed with its first letter in upper case and the rest in lower case. An operation is displayed in lower case. A transformer or a reserved word is displayed in upper case.

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.

An operation definition associates a name with an operation expression and results in a *named-operation*. A transformer definition associates a name with a transformer expression and results in a *named-transformer*.

There is no distinction in the way user-defined named objects and predefined objects are used other than the fact that predefined ones cannot be redefined in the global environment.

If the expression on the right of **IS** 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 the expression on the right of **IS** is itself a named object, the name being defined is associated with the right-hand-side name rather than the expression it has as its association.

External Declaration

<*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.

Remark

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

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. The purpose of a remark is to permit an extended textual description within a definition file for documentation purposes.

Array Expression

An *array* is a data element of the Nial language as described in Chapter 2.

An *array-expression* is a well formed fragment of program text that returns an array when interpreted by Q'Nial. It is usually called an expression rather than an array-expression in this manual except in contexts where the abbreviated term might cause confusion.

The following sections describe how expressions are formed and what they mean. In most contexts, an *array-expression* is formed as an *expression-sequence* as described in the next section. In some contexts, however, it is limited to being to be a *simple-expression* as described in the subsequent section.

Expression Sequence

An *expression-sequence* is sequence of one or more expressions separated by semicolons.

<expression-sequence> ::= *<expression>* { **;** *<expression>* } [**;**]

An *expression* is a *simple-expression*, a *comment*, or one of the control structure expressions.

<expression> ::= *<simple-expression>*
| *<assign-expression>*
| *<selection-expression>*
| *<loop-expression>*
| *<comment>*

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.

Simple Expression

A *simple-expression* is defined by a juxtaposition syntax involving *primary-expressions* and *operations*.

<simple-expression> ::= *<primary-sequence>*
| *<operation-sequence>* *<primary-sequence>*
| *<simple-expression>* *<simple-operation>* *<primary-sequence>*
| *<simple-expression>* *<simple-operation>* *<operation-sequence>* *<primary-sequence>*

<primary-sequence> ::= { *<primary-expression>* }+

<primary-expression> ::= *<constant>*
| *<variable>*
| *<indexed-variable>*
| *<named-expression>*
| *<expression-list>*
| (*<expression-sequence>*)
| *<block>*
| *<cast>*

A *primary-expression* is a program fragment that denotes an array. If a *primary-sequence* has exactly one *primary-expression*, its value is the value of the one expression. 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 value of a *simple-expression* formed by a *primary-sequence* is the value of the *primary-sequence*.

The value of a *simple-expression* formed by the construct *operation-sequence primary-sequence* is determined as follows: the *primary-sequence* to the right of the *operation-sequence* is evaluated, then the operations in the *operation-sequence* are applied to the result in a right-to-left order.

The construct *simple-expression simple-operation primary-sequence* corresponds to infix usage of a *simple-operation*. The value of a *simple-expression* formed by this construct is determined as follows: the *simple-expression* to the left is evaluated, then the *primary-sequence* to the right is evaluated; the two values are made into a pair and the result is obtained by applying the *simple-operation* to the pair. A *simple-expression* can be formed by the infix use of two or more *simple-operations* separating *primary-sequences*. The interpretation given for a single infix use just explained implies that the infix operations are used in a left-to-right order.

The last construct for forming a simple expression describes infix usage in the case where the program fragment after the *simple-operation* involves applying an *operation-sequence* to the *primary-sequence*. It is evaluated in a fashion similar

to the rule governing the previous construct except that the *operation-sequence* is applied to the value of the *primary-sequence* prior to forming the pair.

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.

Primary Expressions

There are several forms of primary-expressions that can form a *simple-expression*. They are described in the next few sections.

Constant

A *constant* is a *primary-expression* that denotes an explicit data value. It is an array that has a fixed specific value throughout the computation and forms one syntactic unit in the formation of a strand. Constants provide a notation for constructing each of the six atomic types as well as a boolean list (bitstring) and a character list (string):

<constant> ::= *<boolean-constant>*
| *<bitstring-constant>*
| *<integer-constant>*
| *<real-constant>*
| *<character-constant>*
| *<string-constant>*
| *<phrase-constant>*
| *<fault-constant>*

A **boolean-constant** is a numeric atom that repre sents a logical value.

<boolean-constant> ::= **1** | **0**

<bitstring> ::= *<boolean-constant>* { *<boolean-constant>* }+

The two letters *l* and *o* are used to denote the abstract boolean values **true** and **false**, respectively. When used as numbers, they are treated as the integers 1 and 0. The names *True* and *False* are predefined named-expressions denoting the values.

A *bitstring-constant* denotes a list of two or more booleans, called a **bitstring**. A bitstring of length one is denoted by *[l]* or *[o]*.

An *integer-constant* is a numeric atom representing a whole number.

<integer-constant> ::= [**-**] { *<digit>* }+

The limits of the range of an integer number depend on the word size of the host computer. If an *integer-constant* exceeds the limit a real constant is used in its place.

A *real-constant* is a numeric atom that denotes a real number. A number that has a decimal point or an exponent part is a *real-constant*.

<real-constant> ::= [**-**] { *<digit>* }+ **.** { *<digit>* } [*<exponent>*]
| [**-**] { *<digit>* } **.** { *<digit>* }+ [*<exponent>*]
| [**-**] { *<digit>* }+ [*<exponent>*]

<exponent> ::= (**e** | **E**) [**+** | **-**] { *<digit>* }+

A *real-constant* is represented by a double precision floating-point number within the computer system. It is converted to the nearest one in the floating-point number system. Because the computer has a fixed length for its representation of floating-point numbers, the precision of the number stored may be less than that specified in the constant.

There is one predefined *real-constant*, **Pi**, which is defined in *defs.ndf*.

A *character-constant* denotes a single atomic character; a *string* denotes a list of characters.

<character-constant> ::= **‘** *<symbol>*

<string-constant> ::= **’** { *<symbol>* }+ **’**

where *symbol* is a member of the set of characters for the host system.

A *character-constant* denotes a literal atom that can be represented on the keyboard, on the screen or on the printer of the computer by a single symbol. The set of characters for Q'Nial is determined by the host system. On most systems, the 128 characters of the ASCII character set are represented in the stan dard way. On some systems, an additional 128 characters are available for special use. The ordering sequence of the characters for sorting purposes is fixed for each Q'Nial version.

There is one character-constant predefined in *defs.ndf* called **Separator**. It has the value / on UNIX and OSX systems. It is useful for building path names for host files on these systems.

A *string-constant* denotes a list of zero or more characters. A string-constant is denoted by a sequence of zero or more symbols bounded by single quote marks. A pair of adjacent single quote marks within the bounding ones denotes a single quote mark in the resulting list.

The remaining two constant constructs denote literal atoms that are phrases or faults.

<phrase-constant> ::= **"** { *<non-terminating-symbol>* }

<fault-constant> ::= **?** { *<non-terminating-symbol>* }

where *non-terminating-symbol* is a member of the set *symbols* excluding the characters:

<blank> **( ) [ ] { } # , ;**

The excluded symbols are omitted in *phrase-constants* and *fault-constants* because they tend to be punctuation symbols which naturally end a phrase. Omitting them avoids easily created errors. A phrase or fault containing the excluded symbols can be created using the operations *phrase* or *fault*, respectively, each taking a string as argument.

A *phrase-constant* denotes an atomic literal that is treated as a unit. Phrases are stored uniquely in an internal table and represented by an internal pointer. Comparisons of phrases for equality are as efficient as integer comparisons since only the internal pointers are compared.

A *fault-constant* denotes an atomic literal that is used to represent an erroneous or extraordinary result. Faults are very similar to phrases but are treated differently by the non-structural operations in order to propagate error reporting information. A fault indicates the type of error or condition detected. All faults produced by Q'Nial begin with a question mark.

Not all faults indicate errors; some signal special values. The fault *?eof*, for example, indicates the end of an input file.

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 in Chapter 9.

Variable

A *variable* is a name associated with an array value. Its syntactic form is that of an identifier.

<variable> ::= *<identifier>*
| *<identifier>* **:** *<identifier>*

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:*.

The second form of variable is used to refer to a variable in a scope that has invoked the scope in which the form appears. It is useful in debugging situations

where the value of variables in the calling environment need to be examined. The first identifier refers to the name of the definition and the second to the variable in scope of the definition.

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 *reserved word*. (A table of the reserved words is given in the *Nial Dictionary* under **reserved words**.) 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 environment, 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 global identifier, i.e. the class of syntactic object it names, cannot be changed.

Scope of a Variable

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.

Indexed Variable

An *indexed variable* is a variable for which a part of the associated array value is referenced:

<indexed-variable> ::= *<variable>* **@** *<primary-expression>*
| *<variable>* **@@** *<primary-expression>*

| *<variable>* # *<primary-expression>*
| *<variable>* | *<primary-expression>*

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.

If an indexed-variable is used as a primary-expression, its value is the value of the selected part of the variable. There are four ways to index an array in Nial as described in the following table:

Symbol Type	Name	Index	Result
@	at	address	item at the address
@@	at-path	path	part at the path
#	at-all	array of addresses	the array of items at the addresses
|	slice	positions on axes	cross sections of items

Nial uses zero-origin indexing, so that the first item in a list is at address 0 and the first item in a table is at address (0 0). When the indexed-variable notation is interpreted, the primary-expression is evaluated. The result of the evaluation of the primary-expression and the form of indexing together determine the locations specified by the particular indexed-variable notation. If the primary-expression does not evaluate to a valid index for the array and the type of indexing, a fault is returned.

In the *at* form of indexing, the result is the item at the given address of the array associated with the variable.

In the *at-path* form of indexing, the value of the index is a list of addresses for subparts of the array on a path descending into the nesting structure of the array. The notation refers to the value of an item of an item etc. for as many levels as the length of the path.

In the *at-all* form of indexing, the index is an array of addresses. An array of the same shape as the index is selected from the array associated with the variable, with items chosen using the index items as addresses.

In the *slice* form of indexing, the index is a list of arrays, each array representing the positions to be chosen for one of the axes of the array being indexed. There must be as many items in the index as there are axes. If an item of the index is the fault *?noexpr* (usually generated using an expression-list with a missing position) that entry is expanded to be the list of all positions on that axis. The cartesian product of the expanded index is then used to form an array of addresses, which are used to select the corresponding items as in *at-all* indexing. The **cartesian product** of a list of arrays is an array of lists containing all the

combinations formed by taking one item from each of the arrays in turn. It is implemented by the same algorithm used for the operation *cart*.

The shape of the result of slice indexing is found by linking the shapes of the items of the expanded index.

Named Expression

<named-expression> ::= *<identifier>*

A *named-expression** is an array-expression that is associated with a name by being predefined or through a definition. The use of a named-expression in a primary-expression causes the associated expression to be executed when the primary-sequence in which it is embedded is evaluated. Each mention of the name causes the expression to be evaluated.

Expression List

<expression-list> ::= [*<expression-comma-sequence>*]
| [*<expression-comma-sequence>* *<simple-expression>*]

<expression-comma-sequence> ::= *<expression-comma-sequence>* **,**
| *<expression-comma-sequence>* *<simple-expression>* **,**
| *<empty>*

The value of an expression-list is a list of the length of the number of simple-expressions in the expression list, counting expressions omitted before or after a comma. The items of the result are the values of the simple-expressions evaluated and sequenced in left-to-right order. If there are no items in the expression list, the result is *Null*, the predefined empty list.

An expression can be missing before or after a comma. The value corresponding to a missing expression is the no-expression fault: *?noexpr*. An expression-list containing omitted expressions is useful in providing an index for the *at-slice* form of indexing.

A list may be constructed directly by using brackets notation or strand notation. In **brackets-comma notation**, the elements of a list are separated by commas and the list is bounded by square brackets. In **strand notation**, the elements of a list are separated by one or more blank spaces. Parentheses can be used with strand notation for grouping two or more items as a strand that forms a sublist of the list denoted by the strand notation. Extra parentheses are ignored in strands.

A list may contain arrays of any type in any sequence desired. Both strand and brackets notation may occur in the same list. A list in brackets notation is one unit in the construction of a list using strand notation. Strand notation can be

used to create a list with two or more items. Brackets notation can be used to create a list with no items, one item, or many items.

Parenthesized Expression Sequence

The value of a *parenthesized expression-sequence* is the value of the expression-sequence. The parenthe ses indicate that the expression-sequence is evaluated before inclusion in a further computation. This use of parentheses is consistent with their use in mathematical notation. An expression-sequence can be made into a primary-expression by enclosing it in parenthe ses.

Block

<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.

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.

The value of a block is the value of the expression-sequence within the block, evaluated within the local context formed by the declarations and definitions within the block, if any.

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.

The determination of the scope of a variable is done when the program fragment in which it is located is analysed. The same scope is then used each time the fragment is evaluated.

Nested Definition

A *nested definition* is one that appears in a definition sequence within a block. The defined name is local to the block. 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 supporting 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.

Cast

A **cast** is an array 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 represen tation 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 internal representation is a nested array forming the parse tree of the construct.

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

The description of the syntactic construct *simple-expression* is now complete.

Assign Expression

<*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 analysed (parsed) and when it is executed.

During the parse of an 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 and the association is with this new variable. 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 expres sion indicates and the corresponding locations are updated with the items of the array value.

The keyword **GETS** is a synonym for the assignment symbol *:=* .

Selection Expression

Selection-expressions are used to describe computations in which one of many actions or values is to be evaluated.

<*selection-expression*> ::= <*if-expression*>
| <*case-expression*>

<*if-expression*> ::= **IF** <*simple-expression*> **THEN**
<*expression-sequence*>
{ **ELSEIF** <*simple-expression*> **THEN** <*expression-sequence*>
}
[**ELSE** <*expression-sequence*>]
ENDIF

<*case-expression*> ::= **CASE** <*simple-expression*> **FROM**
{ <*constant*> **:** <*expression-sequence*> **END** }+
[**ELSE** <*expression-sequence*>]
ENDCASE

In the **if-expression** the selection is made by evaluating the simple-expressions after the *if* and *elseif* keywords as boolean expressions. As soon as one of these evaluates to *true*, the expression-sequence following the corresponding *then* is evaluated and its value returned as the result of the if-expression.

All the simple-expressions after the *if* and *elseif* may evaluate to *false*. If they do and an *else* is pre sent, the result is the value of the expression-sequence following the *else*. If all the simple-expressions after the *if* and *elseif* evaluate to *false* and there is no *else* present, the result is the fault *?noexpr*. If one of the simple-expres sions evaluates to a non-boolean value, the result is the fault *?L*.

In the **case-expression**, the selection is made by evaluating the simple-expression following the *case* and comparing it to the constants preceding the colons. If the value matches a constant, the expression-sequence following the corresponding colon is returned as the result of the case-expression. If the value does not match one of the constants, the result returned by the case-expression is the value of the expression-sequence following the *else* if it is present; otherwise the result is the fault *?noexpr*.

The case-expression form of selection is useful when the choice depends on a single atomic value and there are a large number of possible values the simple-expression can take on.

Both of these forms can be used in a value oriented way or as an imperative. If the desired effect is to return a value, each expression-sequence must omit the final semicolon so that if it is selected, its value rather than the fault *?noexpr* is returned.

Loop Expression

Loop expressions are used to repeat a computation a number of times. In the three forms of loops, the computation is in the form of a loop-body. This is an expression-sequence as described above with the additional property that it, or expression-sequences embedded within it, can use the exit-expression as an additional form of expression.

<loop-expression> ::= *<for-expression>*
| *<while-expression>*
| *<repeat-expression>*

<for-expression> ::= **FOR** *<variable>* **WITH** *<simple-expression>* **DO**
<loop-body>
ENDFOR

<while-expression> ::= **WHILE** *<simple-expression>* **DO**
<loop-body>
ENDWHILE

<repeat-expression> ::= **REPEAT**
<loop-body>
UNTIL
<simple-expression>
ENDREPEAT

<exit-expression> ::= **EXIT** *<simple-expression>*

In the **for-expression**, the simple-expression evalua tes to an array value used to control the loop. The loop-body is evaluated repeatedly with the variable taking on the items of the control array in turn. If the control array is empty, the loop-body is not evaluated.

A **while-expression** implements a loop with a pretest. The simple-expression and the loop-body are evaluated alternately as long as the simple-expression evaluates to a boolean value and is *true*. If the simple-expression is *false* on the first evaluation, the loop-body is not executed.

A **repeat-expression** implements a loop with a post test. The loop-body and the simple-expression are evaluated alternately until the simple-expression evaluates to a boolean value and is *true*. If the simple-expression is *true* on the first evaluation, the loop body is executed once.

In all three loop forms, if an embedded exit-expression is evaluated, the loop terminates early and returns with the value of the simple-expression following the reserved word *exit*. The exit-expression cannot be used outside a loop-expression.

The value of a loop expression in all three cases is the value of the expression-sequence on the last evaluation of the loop body. If the loop body is never evaluated, the value is the fault: *?noexpr*. In the last two forms, if the simple-expression does not return a boolean, the value of the loop is the logical fault *?L*.

Comment

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

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 definition 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*.

The description of the construct *expression* is now complete. The next three sections describe the constructs for *operation-expression*.

Operation Expression

Q'Nial has a large number of predefined operations chosen both for their generality and for their practical utility in a wide variety of applications. Programming in Nial is achieved by defining new operations and using them to carry out a required computation. The following sections describe the program fragments that can be used as operation-expressions.

<operation-expression> ::= *<operation-sequence>*
| *<operation-form>*
| *<curried-operation>*

<operation-sequence> ::= { *<simple-operation>* }+

<curried-operation> ::= *<simple-expression>* *<simple-operation>*

The result of applying an **operation-sequence** to an argument is determined by applying the simple-opera tions 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.

Thus, the effect of applying an operation-sequence of two or more simple-operations to an argument is equivalent to the effect of applying the **functional composition** of the operations to the argument. (The word *effect* is used in this explanation because the application of an operation to an array will always produce a *result*; but some operations produce a *side-effect* such as, for example, writing a record to a file.)

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

Simple Operation

<simple-operation> ::= *<named-operation>*
| *<atlas>*
| *<transform>*
| (*<operation>*)

<named-operation> ::= *<identifier>*

A **named-operation** is a predefined operation or an operation that is associated with a name through a definition.

An operation does not need to be named. It can be created and used in place without being named. However, except for very brief operation-expressions, the normal procedure is to give an operation-expres sion a name using the definition mechanism.

When a predefined operation is applied to an argument, the algorithm implied by the description of the operation (given briefly in Part 2, or in more detail in the *Nial Dictionary*) is followed.

The effect of applying a user-defined operation is equivalent to the effect of evaluating the program text fragment with the following change: the occurrence of the name of the user-defined operation is replaced by the operation-expression (possibly in parentheses if context requires it) which is associated with the user-defined operation name.

An operation can be viewed as a black box with input and output. The argument is the input to the operation and the result is the output. The application of the operation may also produce side effects if the operation does input or output operations or if assignments are made to variables that are declared to be nonlocal.

<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.

<transform> ::= *<named-transformer>* *<simple-operation>*

A **transform** is a simple-operation formed by modifying a given simple-operation. This is done by placing a transformer name before the given simple-operation. An operation-expression that is not a simple-operation can be modified by placing it in parentheses.

The result of the modification denotes a new operation. This new operation is called a transform. When the transform is applied to an array, the operation that is part of the transform is used in the evaluation of the result in the manner determined by the transformer.

A transformer usually specifies a general algorithm which can have one or more operations as parameters. For example, the *EACH* related transformers generalize 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.

If a transform is formed using a named-operation, it is the *name* of the operation to which the transform is bound. The transform is not bound to any specific definition of the operation. Thus, a subsequent redefinition of the named-operation will change the meaning of the transform.

Any operation-expression is made into a simple-operation by enclosing it within parentheses. A simple-operation formed by parenthesizing an operation-sequence returns the same result as the operation-sequence. Parenthesized operations permit curried operations to appear within an operation-sequence and allow compositions of operations to be an argument of a transformer.

Operation Form

<operation-form> ::= **OPERATION** { *<identifiers>* }+ *<block>*
| **OPERATION** { *<identifiers>* }+ **(** *<expression-sequence>* **)**

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 in parentheses without automatic localization.

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-form 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 translated to internal form (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 description of the construct *operation-expression* is now complete.

Transformer Expression

A **transformer** is a functional object in Nial which, when applied to an operation, forms a new operation called a transform. Transformers provide a systematic way of modifying or generalizing operations. Q'Nial provides a number of predefined transformers that have proven useful in extending the expressive power of the language. A mechanism is provided to extend Nial with user defined transformers.

Applying a transformer has no action associated with it other than capturing the environment where it is applied to an argument operation. When the resulting transform is applied, the captured environment is used as the environment in which to invoke the argument operation when it is used within the algorithm of the transformer.

<transformer-expression> ::= *<named-transformer>*
| *<transformer-form>*

<named-transformer> ::= *<identifier>*

A **named-transformer** is one associated with a name by a definition or is a predefined transformer. A transformer cannot be used without being given a name. A predefined transformer modifies its operation argument according to the algorithm given in its description.

A user defined transformer has the effect of the named-transformer expression with which it is associated. A transformer definition associates a name with a transformer-form or with another transformer name. In the latter case, it is the name of the other transformer that is bound not the trans former-expression defined by the other transformer.

Transformer Form

<transformer-form> ::= **TRANSFORMER** { *<identifier>* }+ *<operation-form>*
| **TRANSFORMER** { *<identifier>* }+ **(** *<operation-expression>* **)**

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 construct requires that the operation-expression be an operation-form; the second allows any operation-expression enclosed in parentheses to be used.

The effect of applying a transformer-form to an operation-expression is the effect of the operation formed from the body of the transformer, such that wherever one of the formal operation parameters occurs, it is replaced with the corresponding argument operation-expression. If there is only one parameter then its occurrences are replaced by the argument. If there is more than one parameter then the operation-expression must be an atlas of the same length and the formal parameters are replaced by the corresponding operations of the argument atlas. 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*.

The associations are made with the argument opera tion-expression in the environment where the trans former is applied. Thus, if the transformer is recursive, the formal parameter may have a different association on each recurrence.

Summary of Juxtapositional Syntax

The following table summarizes 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 Class
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

An informal description of the rules for parsing the juxtapositional forms is given in Chapter 3 of Part 1 of the Manual.

Synonyms

The **synonyms** given below abbreviate typing for interactive use or avoid difficulties with some termi nals on older compu ter systems that do not support the full character set used in the syntax rules. The following synonyms are available for keywords or delimiters used in the syntax rules:

Usual	Alternate
[	<<
]	>>
:=	GETS
{	BEGIN
}	END
OPERATION	OP
TRANSFORMER	TR

Chapter 6 File Input and Output Operations

Q'Nial includes operations to create and manipulate files of textual information, to communicate with devices as read/write files and to provide direct access to component files holding arrays or strings. The files are stored externally to the workspace using the mechanisms of the host system.

Sequential Files

A sequential file corresponds to a sequence of lines of text with lines separated by an end-of-line indicator. The files are read and written in units of lines which are converted to and from Nial strings. A sequential file is opened for read, write, append, communications or as a pipe.

The sequential file capabilities are:

open Fn Mode Open file *Fn* in mode *Mode* and return an integer file number *F*.

close F Close the file designated by *F*

readfile F [N] If N is missing, return a string holding one record from the file designated by *F*, omitting the end of line indication; otherwise, return *N* characters from the file *F* including end of line characters; or return a fault *?eof* indicating end of file.

writefile F A [M] Write character array *A* to the file designated by *F*. If *M* is *true* or missing, write the end of line indica tion; if *M* is *false*, do not write the end of line indication.

getfile Fn Obtain all the records of file *Fn* as a list of strings.

putfile Fn S Write strings *S* to the file *Fn*.

appendfile Fn S Append strings *S* to end of file *Fn*.

Filestatus Return a list of triples giving the file number, filename and mode of use of each open file.

A filename is specified by a phrase or a string. The name is given to the operation *open* along with a mode indication given by the following table

Indicator	Open Mode
"r	read
"w	write
"a	append
?c	communications
?pr	pipe read
?pw	pipe write
?d	direct access

A file number, used in all subsequent operations on the file, is returned. A file opened in read mode can be used only for input with the operation *readfile*. A file opened in write or append mode can be used only for output with the operation *writefile*. Write mode creates a new file; append mode opens an existing file at the end, so that additional records can be written to it. If the file does not exist when *open* is used, append mode is equi valent to write mode.

A file must be open to be used by *readfile* or *writefile*. Fault *?eof* is returned if an attempt is made to read beyond the end of the file. If the second argument to *writefile* is a table, one record is written for each row of the table.

Communication mode assumes that the file is being used for both reading and writing. The version of *readfile* in which a number of characters can be specified, provides the low level control of the reading process. The version of *writefile* in which no end of line control is added, allows control over the charac ters transmitted to a communications device.

Pipe read and pipe write modes are used for executing a host command that either reads the result of the execution using *readfile* or provides input to the execution using *writefile*. The command is given as a string in place of the file name. See the discussion of the operation host in Chapter 9.

The operations *getfile* and *putfile* are intended for use on relatively small files in which all the data can be held in the workspace. *Getfile* is equivalent to opening the file in read mode, reading all its records and then closing the file. *Putfile* is the similar composite opera tion for the writing process.

The first three file numbers are used as follows:

0	stdin	the standard input stream
1	stdout	the standard output stream
2	stderr	the error output stream

Thus, *readfile 0* accepts input from the keyboard, and *writefile 1 Data* sends the string *Data* to the display screen. These are useful for testing out a file-oriented program during debugging.

Q'Nial Specific Direct Access File Operations

Q'Nial supports a component style of direct access to binary files in two forms: files in which the components are alphanumeric records that are treated as Nial character strings; and files in which the components are representations of arbitrary array values.

The same underlying mechanism is used to support both forms of direct access files. The direct access files are Q'Nial specific but use the host system files for their representation. Two files are used: a record file given the extension *.rec* and an index file given the extension *.ndx*.

It is possible to treat an externally created file as a *record* direct file provided it is named appropriately and a suitable index file is created.

A direct access file is prepared for use by calling the operation *open* with a file name without an extension and the mode indication *“d*. If the two corresponding host files exist, they are opened. If they do not exist they are created and initialized.

The kind of direct access file is determined by the first write operation applied to it. The use of *writearray* creates a file of array components, whereas the use of *writerecord* creates a file of record components. All subsequent read and write accesses to the direct access file must be of the same kind.

The operations that support direct access files are described as follows

open Fn "d Open the file *Fn* for direct access and return an integer.

filetally F Return the highest component number in the file designated by integer *F*.

readrecord F N Return string component *N* of the file designated by integer *F*.

readarray F N Return array component *N* from the file designated by integer *F*.

writerecord F N B Write the string *B* to component *N* of the file designated by integer *F*.

writearray F N B Write the array *B* to component *N* of the file designated by integer *F*.

eraserecord F N Erase component *N* in the file designated by integer *F*.

In the above operations, *N* can be a list of component numbers. For a read, the corresponding list of components is obtained. For a write, *B* must be a corresponding number of components to be written. The components can be read or written in any order. Note that using an integer with the read operation returns the component as the result, and using a solitary integer, a list with one integer, returns a solitary list with the component as the item.

There is no requirement that all the component numbers below the number returned by *filetally F* be in use. If there is no component at a position specified in *N* in a *readarray*, the result is the fault *?missing*. For *readrecord*, the result is the empty string. The result in both operations is the fault *?eof* if *N* is greater than the result of *filetally F*. In a *writearray* of *writerecord* operation, if *N* is greater than or equal to *filetally F*, then the record is written and the *filetally* is increased to one higher than *N*. Note that the index file used to implement direct access files has an entry for each possible record and hence leaving huge gaps in the record numbers can waste a substantial amount of space.

Eraserecord is used for both kinds of files. If the component removed is the last one, any components immediately preceding it that are not in use are also removed and the filetally is adjusted accordingly. For a file of records it is not possible to distinguish between an unused record and a record consisting of an empty string.

In the write operations, if a component already exists, its value is replaced. The physical host file used to hold the components is not necessarily in the same order as the component numbers. As components are overwritten, their previous space is used if possible, otherwise the information is appended to the end of the *.rec* file. For fixed length components, the order of the components is not changed because of an update.

As components of a direct access file are rewritten out of place because of increased size, or as they are erased, unused space accumulates in the *.rec* file. A record is maintained on the amount of unused space in the file and, after every write or erase, if the unused space has become significant relative to the size of the file, an automatic compression process is executed which rebuilds the file so that the components are in index order.

It is possible to use Q'Nial to access files created by another process. To do this, the file must be renamed to have the extension *.rec* and an index file that is appropriate to the data structure within the file must be built. If the file has a logical structure of fixed size that repeats, the index file need have only as many records as make up one unit of the logical structure. If the record number used in a *readrecord* is above *filetally*, the record file is checked to see if the file extends beyond the length indicated by the *filetally*. If so, the file is assumed to consist of a sequence of blocks of records of the same structure as indicated in the index file and the record selected is determined by *N mod filetally F*.

Direct Access Operations for Host Files

Q'Nial also provides operations that can access the raw byte data of files on the host system. In these operations the name of the file is used as the argument and file open and closes are done implicitly.

readfield Fnm P N Return N bytes of the file *Fnm* starting at position P as a string.

writefield Fnm P S Write the string S to the file named *Fnm* at position P.

filelength Fnm Return the number of bytes in the file named *Fnm*.

The direct access operations for host files are useful in applications where pre-existing files need to be accessed or modified under program control. The *readfield* operation can be used to read in any portion of a file as a string without interpretation of newline characters. If the position plus the length requested is greater than the filelength the fault *?eof* is returned.

The operation *writefield* can write a string of any length to any position in a file. If it is written in a position beyond the current file length, the intervening space may be filled with arbitrary data. If the file does not exist, *writefield* will create it.

Chapter 7 Operations for the Interpreter Mechanisms

Q'Nial has operations that provide direct access by the user to most of the underlying mechanisms that support the evaluation of Nial constructs. These operations behave like most other operations in Q'Nial but they are dependent on the internal representations used in the Q'Nial implementation and may not produce identical results from one version to another. They should not be considered part of the Nial language; rather, they are extensions made specifically in the Q'Nial implementation.

Top Level Loop

Q'Nial7 is implemented as an interactive program running on a console or in a terminal emulation window. The direct input to the interpreter mechanism is a string representing Nial program text. The execution of the text is carried out in a three stage process. First, the string is scanned to produce a list of tokens that represent the component parts of the text. The result is a list beginning with a Token Stream Tag (the number 99) and followed by an alternating sequence of integer codes and phrases.

There is no limitation on the size of tokens corresponding to the literal types. That is, a constant token for a literal string, phrase or fault can be as long as required for an application. However, identifiers are stored only up to a maximum length of 80 characters. The following segments from a log of a session illustrate the top level loop mechanism:

```
set "decor;
Token_list := scan 'A := 3*5.2'
99 2 "A 1 ":= 16 "3 2 "* 18 "5.2
```

The codes for tokens used by scan are:

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

To select the token pairs from the result of *scan*, the following expression is used:

```
Token_pairs := lo cutall rest Token_list
+----+-----+-----+----+-------+
|2 "A|1 ":=|16 "3|2 "*|18 "5.2|
+----+-----+-----+----+-------+
```

To test which tokens are constants, the comparison is:

```
EACH first Token_pairs > 2
oolol
```

The last token pair represents a real number:

```
Realtoken := last Token_pairs
18 "5.2
```

The value of a constant token pair can be obtained by executing the string of the token or by using parse as described below:

```
tonumber second Realtoken
5.2
```

The second stage of executing the string of text is called the parsing stage. The token list is processed and a nested array that represents the structure of the expression denoted by the text is returned. The resulting data structure is a pair: the first item is a Parse Tree tag (the number 100), indicating that the data structure represents a parse data structure; the second item is a nested list of lists where each list is a node of the parse tree for the expression. Each node of the parse tree has an integer tag as the first item to indicate the type of construct that the node represents. The remaining items of a node are themselves nodes of the corresponding components of the construct or represent constants or names:

```
Parse_tree := parse Token_list
```

```
+---+-------------------------------------------------------------+
|100|+-+-----------------------------------------------------------+|
|   ||9|+--+-----------------+-----------------------------------+||
|   || ||13|+--+------------+|+--+----------+------+----------+|||
|   || ||  ||22|2 7670 43248|||49|3 1 3 82 1|1 3 "3|1 5.2 "5.2||||
|   || ||  |+--+------------+|+--+----------+------+----------+|||
|   || |+--+-----------------+-----------------------------------+||
|   |+-+-----------------------------------------------------------+|
+---+-------------------------------------------------------------+
```

In the above example, the tags have the meanings described in the following table:

Tag	Meaning
100	parse tree
9	expression sequence
13	assignment expression
22	variable list
2	variable
49	binary operation application
3	basic operation
1	constant

The parse trees are not intended for modification or for examination by program control. They provide an effective representation for expressions that can be used for evaluation or for recovery of the textual form as described below.

However, they can be used in simple ways such as converting a constant token to its value. For example, to get the value corresponding to *Realtoken* we can

form the corresponding parse tree by:

```
parse (99 hitch Realtoken)

+---+--------------+
|100|+-+----------+|
|   ||9|1 5.2 "5.2||
|   |+-+----------+|
+---+--------------+
```

The real number can be selected by

```
R := second second parse (99 hitch Realtoken)
5.2
```

The third stage in the execution of the string of text is the evaluation stage. The evaluation is done by recursively "walking" the parse tree and evaluating each construct according to the semantic rules of Q'Nial:

```
eval Parse_tree
15.6
```

We have illustrated that a string containing a Nial array expression can be evaluated by the composition of *eval**, parse* and *scan*, which is how the operation *execute* is defined:

```
execute IS eval parse scan
execute 'A := 3 \* 5.2'
15.6
```

It is not intended that users understand or manipulate the internals of parse tree representations of con structs. Q'Nial has few safeguards if *eval* is applied to data structures that are not created directly by the interpreter itself.

In order to reduce the need for users to deal with parse trees, Q'Nial has a method to obtain the internal representation of a valid construct. This is the **cast** array-expression, designated by placing an exclamation mark before the construct.

Q'Nial also has operations to recover the textual form from the internal representation. The operations *deparse* and *descan* accomplish this in two steps: *deparse* converts a parse tree to a token stream that has a few extra token symbols to indicate indentation and line breaks and *descan* converts a token stream into a list of strings. For example:

```
New_token_list := deparse Parse_tree
99 2 "A 1 ":= 16 "3 2 "\* 18 "5.2
Link descan New_token_list
'A := 3 \* 5.2 '
```

When *descan* and *deparse* are applied to the parse trees for array expressions with control constructs such as *if-expressions* and *for-expressions*, the result is a list of lines which when written provide the construct in an indented form. In addition, all identifiers appear with their use encoded as follows:

Type	Canonical Form
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

Figure 8-3 Canonical Form for Nial Identifiers

The operation:

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

provides a canonical string representation for any valid Nial action, in the sense that for such a string S, the two following identities hold:

```
canonical canonical S = canonical S
execute canonical S = execute S
```

The first identity states that the *canonical* operation has no effect on a string already in canonical form. The second states that the canonical form of an action has the same meaning as the action itself.

The *deparse* and *descan* operations are used by *defedit* to transform the parse tree of a definition to textual form. They are also accessed implicitly in the operation *see*.

The evaluation operations:

scan A The token stream corresponding to the string A as a program fragment.

parse A The parse tree corresponding to token stream A.

eval A The value obtained by evaluating a parse tree, cast or a named array expression A.

deparse A Canonical token stream corresponding to parse tree A.

descan A The list of strings corresponding to token stream A.

execute A The value derived by executing the string A as a program fragment.

For console versions the top level loop of Q'Nial is informally described by:

```
EndSignal := false ;
Prompt := ' ' ;
WHILE not EndSignal DO
X := eval parse scan readscreen Prompt ;
IF X \~= ??noexpr THEN
writescreen picture X ;
ENDIF
ENDWHILE
```

where it is assumed that the expressions *Bye* and *Continue* will set the variable *EndSignal* to *True*. Also this version of a top level loop does not implement the use of ?]? to capture the last value computed. A loop is easily written for an application to use a different prompt or a different convention with respect to output of the results of computations. In this way, the Q'Nial interpreter can be tailored for different styles of use.

Picture Operations

The five operations described below give access to the picture drawing capabilities.

picture A The character table that is the picture of A with the current mode settings. In the picture of A, all non-atomic arrays are left justified and numeric atoms are right justified.

sketch A The picture of A in the form that it takes when the mode switches are set to diagram and nodecor.

diagram A The picture of A in the form it takes when diagram mode is set and the decor mode uses its current setting.

display A A string which, when executed, returns A.

paste Sw A A character table constructed by pasting an array A of character tables together according to the settings in switch Sw for vertical edge spacing, horizontal edge spacing, vertical line switch, horizontal line switch, vertical justification indicator and horizontal justification indicator.

positions A The array of addresses in the picture of A where each item of A is placed.

The *paste* operation is used internally by the *picture* operation. *Paste* allows a programmer to produce non-standard pictures of arrays.

The justification indicator is an array of the shape of A, with integer codes for each item; or a single integer code applicable to all items. The justification codes are 0, 1 and 2 for left or top, centred and right or bottom respectively.

The vertical edge spacing is an integer indicating the number of blank lines between rows of items. The horizontal edge spacing is an integer indicating the number of blank spaces between columns of items. The line switch is 1 to draw lines, or 0 to suppress lines.

Phrases, Names and Casts

The names used in Q'Nial in forming associations are represented internally as phrases. The associations are stored in arrays called **symbol tables** that record the environment in which the name is used and its role in that environment.

Q'Nial uses two different mechanisms to refer to names at the scan and parse levels. At the scan level a name is a phrase, which has to be looked up in the correct environment by examining the symbol tables in an appropriate order while parsing the tokens. After the parse is done, a name is denoted by a parse tree that points to its symbol table entry. The cast notation *!Name* is used to denote the parse tree that represents the name. Note that at the top level , 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.

Both representations of names are useful. The phrase is readable. It is looked up in the symbol tables in the context where it is used and hence refers to the object in a **dynamic** way. 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).

value A Return the value of a variable nam ed 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; re turn the result of the operation.

getdef A Return the parse tree associated with the definition name by string or phrase A.

see A Display the definition named by the stri ng or phrase A.

update P A B Put array B at address A in the array associated with the vari able named by the phrase, string or cast P; return the new value of the array.

updateall P A B Put items of B at addresses A in the array associated with the variable named by the phrase, string or cas P; return the new value of the array.

deepupdate P A B Put array B at path A in the array associated with the variable named by the string, phrase or cast P; return new value of the array.

getname A Converts a parse tree symbol table reference A (a triple) to the corresponding name.

getsyms Nm Gets the parameters and local variables of definition Nm.

Assign mimics the behaviour of *gets* or *:=* but does it dynamically. The left argument to *assign* is a phrase, string or a cast naming the variable to be changed or crea ted. The major difference between the operation *assign* and assigning using the *:=* construct is that the former occurs entirely during program execution, whereas the latter has a translation stage in which the scope of the variable name is determined and then an execution phase when the assignment is done. One effect of the difference is that a *new* variable created by the operation *assign* is always placed in the global environment. Another difference is that the operation *assign* does not have the restriction that the phrase or string obey the lexical rule for an identifier and hence it is possible to build associations that do not interfere with variables in the program text.

An important use of *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; whereas in the body of the operation the formal parameter is assigned using *assign* and evaluated using *value*.

The operation *apply* mimics the application of an operation to an array. The operation to be applied is provided as a phrase or a cast. One use of apply is to provide a dynamic switch, where the operation to be applied is selected from a list using *pick* and then applied with *apply*.

The operation *see* writes the canonical form of a definition to the terminal screen. The character table is displayed as if:

```
ITERATE writescreen descan deparse getdef Defnm
```

were executed with *sketch* and *nodecor* display modes. It does not affect the settings of the display modes.

The operations *update, updateall* and *deepupdate* mimic the indexed assignment notations for *at**, at-all* and *at-path* indexing. They are provided to allow selective updating of global variables with no copying.

The operations *getname* and *getsyms* are provided for detailed parse tree analysis and are not intended for general use.

Operations that give user access to the internal mechanisms should be used with due recognition that they may make programs implementation dependent.

Dynamic Execution of Name Associations

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*.

Chapter 8 Management of the Programming Environment

The topics in this chapter describe aspects of the programming environment for Q'Nial that are common to all versions.

The Q'Nial Programming Environment

When Q'Nial is invoked, an area of memory is set aside for use during the interactive session. This area is called the **active workspace**. It holds all the definitions and variables that are created during a session. The size of the workspace can grow as more space is required as long as system space is available.

After initialization, Q'Nial begins interaction, by entering a top level loop. It accepts program text interactively and executes it. Definitions and variables are created by entering actions interactively or by reading in a program text file using *loaddefs*.

The active workspace can be saved and restarted at a later time. It can be cleared or reset to its status at the beginning of the session. A Q'Nial session ends when either *Bye* or *Continue* is executed.

Invoking Q'Nial

There are a number of command line options in invoking the console version of Q'Nial which provide flexibility for its use in applications. A Q'Nial session is invoked by the command:

```
Nial
```

It has the following syntax:

```
Nial [(+\-)size Wssize] [-defs Filename] [-lws WSFilename] [-i] [-h]
```

The parameters are shown within brackets [ ] to indicate that they are all optional. The order of the parameters does not matter. The meanings of the parameters are as follows:

Wsname The named workspace is loaded instead of the *clearws.nws* file that is normally entered or created on invocation. The workspace can contain a *Latent* expres sion and hence can trigger an applica tion. This feature gives direct control over where to start the work. If no name is given, Q'Nial looks for file *co tinue.nws* in the current directory. If it exists, the session is started with it rather than the clear workspace.

***** (+|-)size Wssize***** This option begins Q'Nial with a workspace of *Wssize* words. *Wssize* is an integer (.>= 50000) with a possible suffix of M for megawords or K for kilowords. If *+size Wssize* is used, the workspace size is fixed at the specified size. If an operation cannot complete with the given size, a jump to the top level occurs with the message:

Warning: workspace full Returning to top level.

-defs Filename After loading the starting workspace and executing its *Latent* expression if any, the definition file *Filename.ndf* is loaded using the *loaddefs* operation. The effect is equivalent to executing the expression *loaddefs Filename* when the first prompt is given. The definitions are not displayed when loaded. If the *Latent* expression enters an interaction loop, the file is not loaded until the loop terminates.

-i Run Q'Nial in interactive mode

-h display the help information for invoking Q?Nial.

The options for invoking Q'Nial can be used in the following ways:

1. During application development, the part of an application that is stable could be stored in a workspace, say *appl.nws*, and the current defini tions being debugged could be in *newdefs.ndf*. A system script file (.bat under MS-DOS) having the following line could be used to enter Nial with everything loaded:

   nial -defs newdefs appl

2. An application could be tailored for particular use by loading a predefined set of definitions using the *Wsname* or the *-defs* Filename options.

3. Q'Nial could be used as a filter that executes a program silently and then quits could be invoked with the following linewhere the expression *Bye* is placed at the end of file *Filename.ndf*. The actions in the file are executed but the top level loop is not entered.

   nial -q -defs Filename

If the -q option is not used, a banner message such as the following is displayed:

```
Q'Nial V7.0 Open Source Edition Intel x86 64bit Linux Feb 21 2016
Copyright (c) NIAL Systems Limited
clear workspace created
```

If none of the *-defs, Filename* or *Wsname* options is used, 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. Once the initial workspace has been made, it can be saved using:**

```
save "clearws
```

When the clear workspace is loaded, the default prompt of five blank characters is displayed and the cursor is placed at the sixth position.

Q'Nial views its input as a stream from the standard input file *stdin*. This means that a file of actions can be piped to Q'Nial to use Q'Nial as a filter. One side effect of this design is that one can end a Q'Nial session in the console versions by giving it the end of file signal interactively. On Unix systems by <Ctrl d>. Terminating a Q'Nial session in this manner is equivalent to using *Bye* and hence the current workspace is not saved.

Naming the Latest Result

Q'Nial retains the result of the most recent interaction. To give the last value computed a name, eg. *Var*, the right bracket symbol ( ]) is followed by the variable name.

```
]Var
```

Session Related Expressions and Operations

There are a number of operations that assist in workspace management, managing program development, interfacing with host facilities and debugging.

Restarting a Q'Nial Session

During a session, to start over with the workspace restored to its original contents, one of the expressions *Restart* or *Clearws* is used.

Restart Reset the active workspace to have the same content as that provided when Q'Nial was invoked and retun to top level.

Clearws Reset the active workspace to be cleared of all user defined variables and definitions.

Ending a Q'Nial Session

A session on Q'Nial is ended by the execution of one of the following expressions:

Bye End the session of Q'Nial and return to the host operating system. Information in the current workspace is lost.

Continue Save the workspace as *continue.nws* in the current directory; end the session of Q'Nial and return control to the host operating system. The workspace is restored in the next session started from the same directory using the invocation of *nial* without the *Ws name* option from the same direc tory.

The continue workspace is a convenience for short term saving of current work. If a session that was started from a continue workspace is ended by using *Bye*, the file *continue.nws* is deleted.

Interrupts and Error Recovery

There are certain invalid computations or resource limitations that prevent Q'Nial from continuing with a computation. A message is displayed when such situations arise. Q'Nial can normally return to the top level loop when errors of this type occur. It calls the user-defined operation *recover* (if it exists) after cleaning up all the temporary values existing when the error occurred. The recovery may take a few seconds if there is extensive cleanup of temporary values to be done.

In the definition of an operation or expression, a situation may occur which is best handled by interrupting the computation and returning control to the interaction loop. There are also situations when recovery is needed to prevent returning to the interaction loop. The expression Toplevel causes a return to top level.

Q'Nial permits interruption of a computation by using an interrupt signal that is host system dependent. The interrupt signal is <Ctrl c> on most console versions of Q'Nial.

A user initiated interrupt is equivalent to execution of *Toplevel*. When such an interrupt occurs and *recover* is present, *recover* is executed. Interrupts can be inhibited using the operation *setinterrupts*. It should be used for control of a process which must be completed, such as database updating or user identification.

Toplevel Return control to the user interaction loop after applying the operation *recover* if it is present in the workspace.

recover Msg The occurrence of any condition that forces a return to the interaction loop. The argument is the warning message which can be used to determine the cause of the jump.

setinterrupts A Permit or prevent the interruption of computation. If *A* is *true*, interrupts are permitted; if it is *false*, interrupts are blocked. The default is to permit interrupts.

Fault Triggering

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:

- a trigger mechanism is executed that causes an interruption when a fault is created during execution of a defined operation, expression or transformer, or
- the fault is treated as a normal atomic array.

When Q'Nial is invoked, the fault triggering mecha nism 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*, defined in *defs.ndf*,

can be used to create a fault without causing fault triggering. Fault triggering can be controlled using the following operations.

Operation Action

settrigger A Permit or prevent the interruption of computation by the fault triggering mechanism. If *A* is *true*, a fault triggers an interrupt.

quiet_fault A Return the result of *fault A* 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 mes sage 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 top level loop without an attempt to recover.

Workspace Management

To assist in the management of the workspace, Q'Nial has expressions and operations that are used in the following ways:

Saving and Loading the Workspace

Q'Nial includes mechanisms to save the contents of the current workspace in an external file and to load a previously saved workspace as the current one. During program development, workspaces can be used to avoid reloading a large definition file at the beginning of each session. Also, a workspace can encode an application that is shared with other users without providing the source to them.

The storage scheme used to store the workspace, designed to permit rapid saving and loading, is not portable between versions of Q'Nial on different systems or between revised versions on the same system. Hence, it is always wise to retain definition files in order to be able to reconstruct a workspace. A library program *wsdump.ndf* is available to convert the contents of a workspace into a definition file.

When a *save* or *load* is executed, the current computation is ended and the *Checkpoint* or *Latent* expression is done in the interaction loop 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 interaction loop. It can be used to restart a computational process after an intermediate dump of the workspace.

If a workspace being loaded contains an expression named *Latent*, the expression is executed following the load of the workspace and prior to restarting the interaction loop. *Latent* can be used to initiate a comput ational process or to set internal system variables prior to entering the interaction loop.

save Wsname Save the current workspace under the name *Wsname* with extension *.nws*.

load Wsname Load the workspace in the file named *Wsname* having exten sion *.nws*.

Checkpoint User defined expression executed after a *save*.

Latent User defined expression executed after a *load*.

With *load* or *save*, the extension *.nws* may be omitted in the filename. The filename can be specified as a phrase or a string. If the file naming convention on the host computer is sensitive to upper and lower case, care must be taken to spell the filename correctly.

Loading a Definition File

A definition file is a text file containing Q'Nial actions. It is viewed as a sequence of actions supplied to the system when the file is loaded. A definition file has the extension *.ndf* to distinguish it from other text files. The major purpose of definition files is to collect definitions of expressions, operations and transformers that form a module of code in an application. They may also provide a script of inputs for computations done without user interaction.

loaddefs Fn Switch Read and execute the actions in definition file *Fn*. If *Switch* is *o* or is omitted, do not display the definitions; if *Switch* is *l*, display them.

library Fn Switch Read and execute the actions in definition file *Fn* from directory *niallib*. If *Switch* is *o* or is omitted, do not display the definitions; if *Switch* is *l*, display them.

In *loaddefs, Fn* may be any path that leads to a file with a *.ndf* extension. The assumption is that the path starts from the current directory unless a complete path is given. In *library, Fn* can be any path that starts in the *niallib* directory. *Fn* can be specified as a phrase or a string. With *loaddefs* or *library*, the extension *.ndf* may be omitted in the filename.

The operation *library* uses the environment variable *nialroot* to find the directory that holds the subdirectory *niallib*and the other default libraries. The predefined variable *Libpath* can be set under program control to indicate other directories to be searched. The operation *library* is defined in the file *defs.ndf* and can be modified to provide an alternative library strategy.

Setting Workspace Switches

Q'Nial has a number of optional behaviours that depend on the value of internal switches. These are set by the mode setting operation *set*.

set Switch Set an internal switch that controls how execution is displayed, controls tracing, or controls the output to log files. Return the previous setting. The settings are *sketch, diagram, decor, nodecor, trace, notrace, log*and*nolog*.

The switch setting can be specified as a phrase or a string in upper or lower case.

Workspace Status

The expression *Status* provides information about the workspace. Its results are explained as follows:

Position Value	Value Given	Meaning
0	number of free words	amount of space available in workspace
1	largest free block size	determines largest object that can be built
2	number of free blocks	indicates the fragmentation of memory
3	workspace size	gives the current size of the workspace
4	size of internal stack	used to hold intermediate values in evaluation
5	size of phrase/fault table	area used for hash table for phrases and faults
6	internal buffer size	used for temporary space by many operations

The last three items indicate the size of three internal areas that grow if required, but never shrink in size. After some large computations these components may be larger than needed. They are reset if the workspace saved and reloaded.

Symbol Table

Q'Nial maintains an internal **symbol table** holding the name and role of every object in the workspace. The following operations provide access to the symbol table information:

symbols Sw If *Sw* is *o*, return a list of pairs of names and roles of all user-defined objects in the global environment. If*Sw* is *l*, also include the names and roles of predefined objects.

erase A Erase the variable or definition specified by the phrase *A*.
Leave the phrase *A* in the symbol table with its role unchanged but change its value to one of *?no_value, No_expr, no_op* or *NO_TR* depending on the role of *A*. This operation is used at top level to remove a user defined object from the current workspace. It permits the use of the identifier for a different purpose but not a differ ent role. It cannot be used to erase a name in a local scope.

Vars The list of global variables in the workspace.

Exprs The list of user-defined expressions in the workspace.

Ops The list of user-defined operations in the workspace.

Trs The list of user-defined transformers in the workspace.

The operation *symbols* returns a list of the items in the table. The operation *erase* removes the definition or variable from the workspace although the name remains in the symbol table with the same role. The expressions *Vars*, *Exprs*, *Ops* and *Trs* return the names of user-defined objects in each of these roles.

Logging the Work of the Session

Q'Nial has a number of switches that can be set to permit logging the work of the session. The log file can also be renamed. The log file is opened and closed for each line that is written in order to ensure that it is retained if a session is terminated by an external process.

set Logsw Set a switch that controls how logging is to be handled. Return the previous setting. The switch setting can be a phrase or a string in upper or lower case. The possible settings are *log, nolog* and *nolog.*

setlogname Fn Set the name of the file used to log screen output to *Fn*, which can be a string or a phrase. The name is case sensitive if the host file system names are case sensitive.

The information displayed on the screen during a Q'Nial session is recorded in a log file if logging is activated by the *set "log* action. The screen displays that occur during the host interface operations are not saved in the log file. The log file can be edited to make a working program from the series of trials made while working interactively. The default log file name is *auto.nlg.*

Time Related Expressions

Q'Nial provides two expressions which access the internal clock of the computer.

Time System dependent timing information. On Unix, it is the processor time used for the user and system tasks for the process since the beginning of the session measured in seconds.

Timestamp The current date and time in the form of a string.

Display Related Settings

Q'Nial has a few settings that affect the way in which information is displayed on the screen.

setprompt A Set the Q'Nial prompt to *A*, where *A* is a string or a phrase. The maxi mum length prompt is 40 characters.

setwidth N Set the width of output displayed on the screen and sent to the log to *N* characters. The result is the previous screen width setting. Setting the output width to 0 allows the output to be of arbitrary length. Used by default in CGI-Nial.

setformat Str *Str* is a format specification for real numbers using the conventions for the *C* library routine *printf*. The result is the previous format setting.

setscroll Mode Set the scroll setting for window mode to *Mode*. (Console versions only)

Screensize Returns the height and width of the actual screen or window in use.

The default prompt displayed by Q'Nial in the interaction loop is a string of five blank characters. If a more visible prompt is preferred, *setprompt* can be used to change it.

Array pictures are written to the screen and to log files so that they fold as a unit; that is, if the array picture is too wide, all of its lines are displayed in the available space and then after a blank line the remain ing portion of the picture is displayed. The width of the display field defaults to the screen size but can be increased or decreased using *setwidth*.

Real Number Formatting

The three styles of real number formats that can be set using *setformat* are given below:

'%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, and

'%g' displays the number in *f* format if possible but defaults to *e* format if the number is not within a suitable range.

Finer detail format specification is achieved by placing two numbers separated by a period between the '%' and the letter. For example '%15.5f' uses a field of width 15 to display a number in *f* format with 5 decimal places. In general, the first digit refers to 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 left out

Because a real number display needs to be distinguishable from an integer, there are some cases where the field is one space wider than predicted by the format string in order to accommodate the decimal point added to the end. Also, if an *f* format is not wide enough for the number, it is widened so that the number is displayed.

The format string *'%.17g'* is used by the operation *display* in depicting real numbers. This format accurately reproduces the same number when executed on most platforms.

The default format is *'%g'*, which displays the number in a compact format displaying 5 significant digits. For numbers in the range 1e-5 < x < 1e6, it omits the exponent, for larger or smaller numbers the exponent is included. In both cases trailing zeros are removed.

System Related Expressions

There are four expressions which hold information about Q'Nial:

Copyright The copyright message.

Version The release and version of Q'Nial being used.

System The operating system in use.

Nialroot A string showing the path to the *nialroot* directory.

Host Interface Operations

Q'Nial provides access to the operating system and to an editor of choice.

host Cmd Pass string or phrase *Cmd* to the host command processor for execution. The details of the use of *host* are specific to the operating system of the com puter being used. At the top level loop, an input line beginning with ! is treated as a host command. For most versions of Q?Nial, the host command causes Q'Nialto block until the command is completed.

edit Filename Edit the text file named by the phrase or string *Filename* using the host default editor. Definition files can be prepared and modified from within Q'Nial so that sessions of Q'Nial are not interrupted to handle editing.

defedit Def Place the definition named by phrase or string *Def* in a window in canonical form and invoke *editwindow* with the lines of the definition as strings in the window. After editing, execute the strings to load the definition. This is only available in the console versions of Q?Nial.

The operation *host* is platform dependent. Its argument is a command line that is executed by the command processor for the host system. If the command processor supports output redirection for the command in use then the results of host command can be sent to a file and obtained with *getfile*. An alternative is to use the pipe read mode of the operation *open* to have the lines of output of the result available using *readfile*. The library operation *newhost*implements a version of *host*that uses pipes.

The operation *edit* is used to access a standard text editor on the host system in console versions of Q'Nial. The editor to be used is determined by the environment variable *EDITOR* which the user can set in the operating system environment. If the variable is not set a default editor is chosen.

Defedit is convenient for quick modification of an operation. Its definition is in file *defs.ndf*. It can be modified to suit individual tastes.

Session and Workspace Variables

During execution of Q'Nial, a number of global variables within the interpreter determine how the environment behaves. These are classified as session or workspace variables depending on whether or not they are saved with the workspace and cause the environment to change when a workspace is loaded. The two tables below contain some internal variables that are only used with console versions.

Session Variable	Default Setting	Setting Operation
log file setting	nolog	set ?log; set ?nolog
log file name	auto.nlg	setlogname Fnm
screen display width	80	setwidth N
trace setting	notrace	set ?trace; set ?notrace
interrupts enabled	true	setinterrupts M
fault triggering	on	settrigger M
messages	on	setmessages M
prompt string	5 blanks	setprompt Str
workspace size	200,000 words	command line option
stack size	3,000	N/A
phrase/fault table size	4,000	N/A
Internal buffer	1,000	N/A

The following table describes the workspace variables.

Workspace Variable	Default Setting	Setting Operation
sketch/diagram switch	sketch	set ?sketch; set ?diagram
decor/nodecor switch	nodecor	set ?decor; set ?nodecor
real format	'%g'	setformat Str
definition trace	*o* for all definitions setdeftrace Defnm [M]	

The sizes of the workspace, the stack and the phrase and fault table are automatically increased as required during the session while space is available. Some of the session variables can be set using start up options that differ from version to version.

One approach to setting the programmable settings is to initialize them as desired in a *Latent* expression in a workspace that is loaded when a Q'Nial session is started. They session variables will then be set as Q'Nial is initialized or when the *Restart* expression is executed.

Workspace variables are internal settings that are preserved with the workspace. The above figure describes them. For example, if a workspace is saved with *diagram* and *nodecor* modes set, the reloaded workspace will retain those settings. However, if the *log* setting is in effect when the workspace is saved, nothing is recorded in the saved workspace to restore that setting on a reload of the workspace.

Chapter 9 Debugging and Profiling Nial Definitions

Debugging Definitions

The Q'Nialsystem provides an optional debugging facility that aids interactive debugging. 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 to monitor all uses 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 an operation or expression.

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 reset an internal break flag for the definiton 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.

For console versions supporting the alphanumeric windows package, break mode can also be entered when Q'Nial is awaiting keyboard entry in window mode (except in *editwindow*) by typing <Ctrl b>. When normal operation is resumed, the previous screen is displayed and the input request is still pending.

When initially entering the "debug mode" because of a break caused by any of the above methods, a short banner is printed that indicates that the debug evaluation loop has been entered and it is followed by the expression to be executed and a prompt with a default debugging command.

```
------------------------------------------------------------
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]
```

Whenever you initially enter *debug mode*, the current callstack is printed to indicate where the break occurred. The debugging banner also reminds you that you can resume execution by typing resume and that hitting return will cause the debug action, indicated in square brackets at the prompt, to occur. The

debugging action defaults to *stepv* but changes to the last debug command used if you execute another one.

At the debug prompt, the user can explore the values of variables by entering the variables name, can evaluate any Nial expression, say to display the definition with *see* or to get the *shape* of a variable, or can execute one of the debug commands described below that moves the computation forward.

A breakpoint set by the operation *breakin* sets an internal breakpoint flag. If the user reloads the definition (maybe as part of a *loaddefs*), then a breakpoint flag remains set if it was set. The breakpoint flag is also preserved if the workspace is saved and then subsequently loaded. Using *breakin* again or *erase* on the definition name clears the internal breakpoint flag. To clear all internal breakpoint flags execute

EACH breakin Breaklist

Access to Intermediate Scopes

Nial has lexical scoping, that is a name is only directly visible in a definition if it is defined in the definition or in a surrounding one. For debugging, we often wish to look at the value of a variable in an operation that called the operation we are currently in. The scoped variable reference, one of the forms of a primary expression, is used to observe the value of local variables in named expressions that are blocks and in named operations that are operation forms. The use of a scoped variable reference produces the value of the variable for the most recent use of the definition in the call stack. The syntax is

funname:varname

This is general syntax that can be used under program control anywhere a variable reference is made, but can only be used to reference a variable, not to change it.

Debugging Commands

The debugging commands permit resumption of execution in several forms, with or without the display of data computed during execution. The

step(v) [N] *step* executes the next executable expression (displayed after *?..*) in either the current definition or in a definition called within the current expression. It *steps into* definitions called within
an expression. If *stepv* is used, the value of the executed expression is displayed. If N is provided (N>=1) then the command is executed N times before the user is prompted again.

next(v) [N] *next* is similar to *step* except that it does not go into definition calls, but rather, executes a definition within a the current expression quietly. It is used to trace the statements within one definition, without showing the detailed execution of definitions that are called by the current expression. If *nextv* is used, the value of the executed expression is displayed. If N is provided then the command is executed N times before the user is prompted again.

stepin [N] *stepin* is the same as *step* except that it traces the evaluation of the next expression to be executed. It is useful for seeing the details of how an expression is computing its value.

toend(v) *toend* executes to the end of a loop or of a definition. It is useful if you want to skip all of the remaining expressions within a loop or an expression sequence in a definition and stop on the first expression after the loop or definition call.

resume *resume* continues program execution until either another *break* or *fault* is encountered, or until the expression being executed completes.

The debugging facilities can be used in a number of different ways. For example, you can step forward from a breakpoint using *nextv*, *stepv*, or *stepin*, depending on the details you wish to observe. If a *breakin* is used to break in a definition then you may want to use next to move quickly to the area where the problem is. Alternatively, you can edit the definition and insert a *Break* expression at the point of concern and reload the definition. Then execution will be interrupted at that point and you can enter variable references to observe the value of variables that affect the computation, or try out parts of the next expression to be executed to see why it is not working as expected.

Another way to get debugging started at a convenient place is to place a watch on a variable with its corresponding action being a break. See the section below on the Watch Mechanism.

The debugging commands *step* and *next* have special behaviour for the control structure expressions such as if-then-else and while-do expressions. When the next expression is a control structure expression only two lines of it are displayed. If *next* or *step* are given as the command, then the next expression becomes one of the components in the control structure expression. For example, in an if-expression, the next expression becomes the boolean expression following *IF*. Nothing is executed, just the focus of debugging has moved into the control construct. If the value versions of the command are used, then on the execution of the last expression in the control construct, both the value for the last expression and the value for the entire control construct are displayed.

The *toend* command serves a dual purpose: it is used to terminate a loop, or to terminate a definition. If execution is in a loop and *toend* is given as the command, then all the remaining iterations of the loop are executed and

execution stops after completion of the loop. If the current loop is nested within another, only the inner loop is executed to completion.

If the *toend* command is given at the debug prompt and the current expression is not within a loop, then the execution continues to the end of the definition. If the definition had been called during the execution of an expression by using *step* or *stepv* then the execution will stop on the expression following the definition. If the break that started debug mode was in the definition, then *toend* behaves like *resume*.

In the rare case where there is a conflict between a variable name and a debugging command you can precede an expression given in the debugging loop with a backslash (\) to indicate that it is an expression to be evaluated.

Repeating Debug Commands

All versions of the debug commands also allow the addition of an integer argument. This argument is interpreted as the number of times to execute the given debug command, before returning to the debug prompt. For example, using the debug command *nextv 10* will show all the results of *nextv* as if the command was issued ten times. After ten executions, the debug prompt is printed

Monitoring Execution Flow

In trying to understand the behaviour of a program it is sometimes necessary to trace the flow of execution to see how the flow has gone. The operations described in the next table assist in this endeavour.

Debugging is an art. There is no best way to use the debugging facilities to solve a problem. For complex situations where the problem involves deeply nested definition calls, it may be convenient to issue *Callstack* to find out what definition is executing, and at what level of nesting it is at. Also using *see* on the current definition helps to see the context in which the debugging is occurring.

Callstack Display the sequence of active definitions at the point of the break.

seeusercalls M Set the flag that controls display of entry to and exit from user definitions to boolean value M. If M is True then display the call information.

seeprimcalls M Set the flag that controls display of the execution of primitive functions to boolean value M. If M is True then display the call information.

Watch Mechanism

The debugging system also includes an ability to place a *watch* on a global or local variable. The capabilities are:

watch Var Expr Set a watch on the variable given by cast *Var* if *Expr* is a non-empty string. If *Expr* is empty, remove the watch from the variable.

Watchlist Return the list of watch variables and expressions that are currently set.

The approach is to associate an expression to be executed when the value of the variable changes. The expression can display the value that has been assigned, or execute a *Break* expression to interrupt the computation. The variable is referenced by 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 action to be performed is given as a string of program text that denotes an expression.

Trace Mechanism

The trace facility has two capabilities: to trace the execution of expressions at the top level and to trace the evaluation of the application of a defined opera tion.

set Tracemode The action *set "trace* causes the execution of expressions at the top level to be traced. It presents the result of the application of a defined operation but does not trace the evaluation of the defined operation. Trace mode is turned off by the action *set "notrace*.

setdeftrace Def [Sw] Change the trace mode setting for definition *Def*. If *Sw* is *true*, turn on tracing; if it is *false*, turn it off. If *Sw* is omitted, reverse the setting.

The trace mechanism provides a lot of output for the evaluation of an expression. The intermediate expressions within the expression being traced are displayed and their values printed. This is appropriate for small values, but becomes unwieldy for large data objects. Thus, debugging with tracing is best done with small arrays. The output produced by tracing is captured by the log facility and hence it possible to study the details of the execution *after the fact*.

Profiling Capability

Q'Nial has a profiling capability that can be used to gather relative execution times for defined operations, transformers and expressions. The data on execution

times is collected relative to the tree of function calls, so that both the total time a routine is used and the time it is used from a specific other routine can be computed. The profiling capability makes it easier to find inefficiencies in Nial programs. The following summarizes the capabilities.

setprofile M sets the internal flag that turns on or off the collection of data on execution time to *M*. If *M* is True the data is gathered.

profile Fnm displays the profile data to the screen if *Fnm* is the empty string or writes it to the file named by *Fnm*.

Clearprofile clears the current profile information and reinitializes the profiling system.

profiletable provides the profile information as a table

profiletree provides detailed profile information in terms of the tree of calls.

The profiling capabilities are described in more detail in the *Nial Dictionary*.