An Introduction to Nial

Michael Jenkins

2015-08-01

Preface

This Introduction to Q'Nial is a quick guide on how to use Nial. It is intended for people with some experience in programming languages such as Pascal, C, Fortran or Turing. Concepts similar to one in those languages are described, but the main focus of the text is to explain the novel aspects of Nial programming. The guide provides suggestions on how to use the features of Q'Nial effectively.

The information is presented in six chapters.

Chapter 1 shows a first session, explains some differences with other languages and shows how to define an operation.

Chapter 2 provides an overview of Nial regarding data objects and introduces operations.

Chapter 3 describes data manipulation by functional objects, operations and transformers. It shows how various rules in Nial are used in the derivation of a result of an expression and explains brackets notation and indexing.

Chapter 4 describes several types of expressions and expression sequences and explains how to define a parameterized operation and a transformer.

Chapter 5 is an extended session showing how to solve a problem from graph theory using Nial.

Chapter 6 provides an overview of the Host System interface and describes some of the mechanisms of the Q'Nial interpreter that are available for direct use.

It is a good idea to experiment with Q'Nial as you read the material in the Introduction to Q'Nial.

Chapter 1 Getting Started

Conventional high level languages, such as Fortran, C or Pascal, are designed for the creation of complete programs using a text editor. The program is translated (compiled) to machine language and finally executed (run) to carry out its task. Conventional program development consists of an edit, compile, and run cycle that is either managed manually or controlled through a programming environment.

Q'Nial is designed for the creation of an evolving set of program fragments developed interactively. Q'Nial comes as a console version where a session is started by typing

```
nial -i
```

Q'Nial responds by presenting a banner of information and then awaits input at the keyboard. You interact with Q'Nial by typing a program fragment on the keyboard and reviewing the result displayed on the screen. Each keyboard entry is completed by pressing *Enter*. To end the session, you type *Bye* or *Continue*.

A First Session

The following session uses the Linux Console Version of Q'Nial:

```
> nial -i
Q'Nial V7.0 Open Source Edition Intel x86 64bit Linux Jul 29 2015
Copyright (c) NIAL Systems Limited
clear workspace created

     A := 'hello'
hello

     B := 'world';

     A link B
helloworld

     link A ' ' B
hello world

     Bye
```

The messages indicate that the session starts with the initial workspace, which provides access to predefined Nial objects.

The session begins with an assignment of the string ‘hello’ to the variable A. A is the variable; *gets* is the assign action; the string ‘hello’ (a list of characters) is the expression. The string is displayed as the output of the interaction. The entry typed at the keyboard is shown indented because the default prompt for Q’Nial, the indication that Q’Nial is awaiting input, is five blank spaces.

```
     A gets 'hello'

hello
```

The default behaviour is that every input action in Q’Nial returns a value and that the value is displayed immediately after the input. The value of an assignment action is the value of the expression to the right of *gets*. In the example above, the expression is the string ‘hello’.

```
    B gets 'world' ;
```

An input action that ends with a semicolon (;), however, does not display a value. This occurs because the value returned in this case is a special value *?noexpr* which, by convention, is not displayed in the top level loop.

```
     A link B

helloworld
```

The next input is an infix use of the predefined Nial operation *link* which joins strings together. The result is the joined string with no separation between the strings held by A and B.

```
     link A ' ' B

hello world
```

The above input uses *link* in a prefix manner to join three strings: the string held by A, a string holding a single blank character and the string held by B. The result is a string with the words separated by a blank.

```
     Bye
```

The final input action, *Bye*, ends the session. If *Continue* were used instead of *Bye*, the workspace information would be saved as the file *continue.nws* in the current directory and the next invocation of Q’Nial from within the same directory would restart the session with variables A and B defined.

Some Differences with Other Languages

The above session illustrates some differences between Nial and conventional compiled languages. First, the use of a console version of Q'Nial is driven by a top level loop which consists of:

- Issue the prompt.
- Read an input action typed after the prompt.
- Execute the action.
- Display the value of the action if it is not the fault *?noexpr*.

This is similar to the edit-compile-run cycle of compiled languages but differs in that the edit-compile-run cycle is done on a program fragment consisting of one line and the output is automatically displayed. This paradigm is common in other high level languages that have an interactive interface such as Lisp, Prolog, APL and Smalltalk.

A second difference is the lack of declarations for the variables A and B. Nial is an untyped language in the sense that variables and operation parameters do not have a fixed object type associated with them. In this case, both A and B are assigned objects that are strings. The type information is associated with the objects rather than with the variables.

A third difference is illustrated by the operation *link*. In Nial, all data objects have both structure and content. Operations, which correspond to value-return ing functions in compiled languages, map data objects to data objects.

Nial can use display settings to provide more meaning ful displays of values. The examples in "A First Session" were shown in sketch, nodecor modes. The following session shows a similar set of examples in diagram, decor modes. In the examples, the double quote mark (") before diagram is a phrase designator indicating to Q'Nial that diagram is a word or phrase, which is data, and not a variable name.

```
     set "diagram
sketch

     set "decor
"nodecor


     A gets 'Hi'

+-----+
|`H|`i|
+-----+
```

```
     B gets 'Mom'
+--------+
|`M|`o|`m|
+--------+


      C gets A ' ' B
+-----------------------+
|+-----+|+--+|+--------+|
||`H|`i|||` |||`M|`o|`m||
|+-----+|+--+|+--------+|
+-----------------------+


     link C
+-----------------+
|`H|`i|` |`M|`o|`m|
+-----------------+
```

The operation *set* is used to control a number of internal switches. Its argument is a phrase or string indicating the setting desired. Here it is used twice to place Q'Nial into the diagram-decor mode of output display. In this mode, the full structure of data objects is displayed (diagram mode) and the contents are decorated to distinguish among atomic types (decor mode). With the display modes set, it can be seen that the argument to *link*, stored here in variable C, is a triple (a list of length three) of lists of characters of lengths 2, 1 and 3 respectively. The diagrams illustrate that *link* joins the lists that are items in its argument to form a list of characters of length 6.

The next example illustrates that data objects such as C can be built up by combining other data objects without any detailed description of their structure. The data objects are dynamic in that their size and struc ture do not have to be described before they are computed.

Returning to the discussion of *link*, the first example used it as an infix operation on A and B and the second as a prefix operation on the triple assigned to C. The dual usage is possible because Nial views all operations as unary operations. That is, they take a single array as an argument and return a single array as a result. Nial translates an infix use to a prefix use applied to the pair formed from the two arguments. Thus, the action A *link* B is equivalent to *link* A B. As well, *link* can be used to join lists of lists of any type or even of mixed type.

The work done by *link* in forming its result involves looping over the items of the argument and, for each item, looping over its items to move them to the result. Thus, for a task of joining m lists each of length n, *link* does m times n elementary data move ments. Most of the predefined operations of Nial have the property of doing a substantial computation in terms of the elementary steps done by a conventional compiled language. This higher level view of data and operations on data gives Nial an expressive power that supports rapid problem solving.

Defining Operations

The actions presented in the two previous sessions use built-in capabilities of Nial. In order to program, new functional objects that capture a number of computa tional steps under one name must be created. This process is called abstraction and is central to all programming. Consider the following:

```
     Numbers gets 45.2 3.7 -35.3 87.14
+--------------------+
|45.2|3.7|-35.3|87.14|
+--------------------+

     sum Numbers
100.74

     tally Numbers
4

     sum Numbers / tally Numbers
25.185
```

The four actions above assign a list of numbers to the variable Numbers, sums the list of numbers, measures the length of the list of numbers and computes the average of the list of numbers. These actions compute array values and are called array-expressions.

```
     average IS OPERATION A {sum A / tally A}
     average Numbers
23.788

     see "average
average IS OPERATION A {
sum A / tally A }
```

The next action is a definition of a new operation that computes the average of a list of numbers. An action that is a definition does not return a value. The definition consists of a name for the object being defined *average*, the reserved word IS followed by reserved word OPERATION followed by an identifier A as its one formal parame ter.

The body of the function is a block that has just one expression, namely the expression that computes the average of A. The action following the definition uses *average* to compute the average of the list held by Numbers. The final action uses the operation *see* to display the definition of average in canonical form. When a definition is processed it is stored in the workspace for later use. It can be viewed by see (and edited using defedit in console versions of Q'Nial).

The session just described shows that a short definition can be typed as an action in the top level loop. However, this technique is impractical for operation definitions that span several lines. To prepare longer definitions, it is preferable to use a text editor and then to load the definitions as one action. The operation *loaddefs* provides the capability to load one or more definitions that are stored in a text file.

There are several ways to prepare a longer definition. The simplest way on a multiwindow system is to invoke a text editor in a separate window, create the definition and store it as a file, say *mydef.ndf*. A second way is to open the default system editor using the action *edit "mydef.ndf*, create the definition and store it. In either case, the action *loaddefs"mydef* is used to bring the definition into the workspace. A definition file name is required to have the extension *.ndf*, but the extension may be omitted in the argument to *loaddefs*.

Definitions created as actions or by using an editor can be retained by using the operation save to store the workspace as a binary file. The workspace can be reactivated later using *load*.

Chapter 2 An Overview of Nial

Nial is based on a mathematical treatment of array data structures, informally called array theory, and on standard features borrowed from other programming languages. Thus, it combines a mathematical approach to the treatment of data with well understood programming concepts. These capabilities are presented in an interactive environment driven by the Q'Nial interpreter. Nial is ideal for exploratory or experi mental computing, but can also be used effectively for doing intensive computations involving large data sets.

The concept of program development in a conventional compiled language does not apply to Nial. In Nial, the unit of translation is an action which either defines a new functional object to be held in the workspace or describes a data object to be computed in the form of an array expression. The latter may assign the data object to a variable retaining the data object in the workspace. A session consists of a sequence of interactions with the pro grammer or user in which actions are entered. As the session proceeds, definitions and variables are collected in the active workspace. The meaning of an action is determined by definitions and vari ables already processed.

The operation *loaddefs* can be used as an action that processes several other actions stored in a file. A program in the conventional sense can be created as one large definition in a file and then loaded and invoked using *loaddefs*; but such an approach ignores the advantages of being able to work in a combination of textual definition and direct interaction. Loaddefs may be used to bring in a single definition or a set of definitions prepared using an editor and can call itself recursively.

The Data Objects

All data objects in Nial are considered to be array data structures. An array is a collection of data objects organized along axes. The most frequently used structure is the list which has only one axis. A string is a list of characters.

Any collection of data objects can be stored in a list; there is no requirement that all the items have the same type. Thus, record types in Pascal or structs in C can be represented in Nial as lists.

Consider the following examples of lists displayed in diagram mode. A list of characters:

```
     'hello world'

+---------------------+
|h|e|l|l|o| |w|o|r|l|d|
+---------------------+
```

A list of numbers created by the operation count which returns the list of integers starting at 1:

```
     count 12

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

A list of two items of different type:

```
     "age 23

+------+
|age|23|
+------+
```

A list of two items, each of which is a list of items:

```
     (count 5) 'hello'

+----------------------+
|+---------+|+---------+|
||1|2|3|4|5|||h|e|l|l|o||
|+---------+|+---------+|
+----------------------+
```

Arrays can nest within other arrays. The above example is easier to comprehend when the boxes of the innermost lists are omitted as in the example below:

```
     set "sketch; (count 5) 'hello'

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

An array with two dimensions is called a table. Tables are convenient data structures in which to store character information that is to be displayed on a computer screen; or numeric data that represents a matrix in a scientific computation:

```
     set "diagram; T gets 3 4 reshape count 12

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


     set "sketch; T

1  2  3  4
5  6  7  8
9 10 11 12
```

A table is built using the operation *reshape* with first argument giving the number of rows and columns and the second the list of items. T is a 3 by 4 table. In diagram mode, a table is boxed. In sketch mode, if all the items are atomic, the boxing is omitted. An array having all atomic items is said to be simple.

There are many operations that manipulate tables in Nial.

```
     transpose T

1 5  9
2 6 10
3 7 11
4 8 12


     rows T

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


     cols T

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

The rows of T and the columns of T are related by the operation *pack*, which interchanges the levels of a list of lists. The operation *mix* can be used to convert a list of lists to a table:

```
     pack rows T

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


     mix rows T

1  2  3  4
5  6  7  8
9 10 11 12


     mix cols T

1 5  9
2 6 10
3 7 11
4 8 12
```

Higher dimensional arrays are sometimes used in three dimensional modelling. A three dimensional array can be created using *reshape*:

```
     A gets 2 3 4 reshape count 24

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

The array A is a 2 by 3 by 4 array of the first 24 integers counting from one.

The operation *raise* can be used to partition a higher dimensional array into an array of lower dimensional ones:

```
     List_of_tables gets 1 raise A

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


     Table_of_lists gets 2 raise A

+------------------------------------+
|1  2   3  4| 5  6  7  8|9 10 11 12 |
+------------+-----------+-----------|
|13 14 15 16|17 18 19 20|21 22 23 24|
+------------------------------------+
```

List_of_tables is a list of two 3 by 4 tables. Table_of_lists is a 2 by 3 table of lists of length four. In the following example, the operation *reverse* is used directly on List_of_tables giving Rev_tables. The items of List_of_tables can be reversed by modifying the operation *reverse* using *EACH*. In these examples, *gets* is replaced by the symbol *:=* which is an alternative notation for assignment.

```
     Rev_tables := reverse List_of_tables

 +-----------------------+
 |13 14 15 16|1  2  3  4|
 |17 18 19 20|5  6  7  8|
 |21 22 23 24|9 10 11 12|
 +-----------------------+
```

```
     Rev_tables_items := EACH reverse List_of_tables

+---------------------+
|12 11 10 9|24 23 22 21|
| 8  7  6 5|20 19 18 17|
| 4  3  2 1|16 15 14 13|
+---------------------+
```

The effect of a *raise* can be undone using *mix*:

```
     mix List_of_tables

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

For mathematical completeness, atomic data objects (also called atomic arrays) such as booleans, integers, characters and real numbers are treated as arrays with no axes and are viewed as self containing. The effect of the latter choice is that the operation *first* used on an atom returns the atom.

Nial has two additional atomic types not found in conventional languages: phrases and faults. A phrase is an atomic unit of literal data. Every phrase cor responds to a string but the internal structure is concealed so that the character data cannot be directly accessed. The string content can be retrieved using the conversion operation *string*. Q'Nial stores phrases uniquely, that is, only one copy of each phrase in use is kept in the workspace. This choice allows a phrase to be represented as an integer that points to the content internally. The use of a unique representation results in efficient equality comparisons and copying of phrase data.

Faults are similar to phrases but are used to denote special values or error conditions. Faults that report errors cause an interrupt of the current execution when they are created unless the default behaviour of Q'Nial for fault triggering has been modified. The special value faults such as *?noexpr* and *?eof* behave like phrases except that they can be tested using the test *isfault*.

As an illustration of the use of Nial data objects and some of the data manipulation capabilities in Nial, consider the requirement to display a table of data with appropriate labels on the rows and columns and an overall label. Given the data table:

```
     Data
```

```
23 34 12
38 18 26
37 13 42
```

produce the following display as a table for a report:

```
Crop Yields Oats Rice Corn
None          23   34   12
Arsenic       38   18   26
Old Lace      37   13   42
```

The operation *labeltable*, defined in the Nial library, can be used for this purpose.

```
     library "labeltab; see "labeltable

labeltable IS OPERATION Corner Rowlabel Columnlabel Table {
  % Combine the labels ;
  Firstrow := Corner hitch Columnlabel ;
  % Hitch labels to the rows ;
  Labelledrows := Rowlabel EACHBOTH hitch rows Table ;
  % Hitch labels to the rows and mix ;
  mix ( Firstrow hitch Labelledrows ) }
```

First the data objects required for applying the opera tion are prepared:

```
     Data := 3 3 reshape 23 34 12 38 18 26 37 13 42 ;
     Corner := phrase 'Crop Yields' ;
     Collab := "Oats "Rice "Corn ;
     Rowlab := EACH phrase 'None' 'Ar senic' 'Old Lace' ;
```

To do this, the table data is entered as a list and reshaped. The label information is specified as phrases, one for each row and column of the data and one for the corner. The operation *phrase* is used to create a phrase with embedded blanks for the corner. The transformer *EACH* is used with *phrase* to build the row labels that also include blanks. Then the operation *labeltable* is applied to the list of arguments yielding the desired result:

```
    labeltable Corner Rowlab Collab Data

Crop Yields Oats Rice Corn
None          23   34   12
Arsenic       38   18   26
Old Lace      37   13   42
```

Labeltable is defined as an operation form, consisting of a header

```
OPERATION Corner Rowlabels Column labels Table
```

and a body which is a block marked off by { and }. The block is a sequence of six expressions separated by semicolons. The expressions beginning with % are comments. The operation is called by the action:

```
labeltable Corner Rowlab Collab Data
```

The action is evaluated in two steps. First, the actual argument provided in the call (in this case, a list of 4 arrays) is assigned to the formal parameters in the header. Then, the block forming the body of the operation form is evaluated. This second step involves executing the expressions in the block in order. The result of the execution of the operation is the value of the last expression.

The details of the execution of the block are as follows. The expression

```
Firstrow := Corner hitch Columnlabels
```

attaches the value in *Corner* to the list in *Collab* to give the local variable *Firstrow* the value

```
Crop Yields Oats Rice Corn
```

The expression

```
Labelledrows := Rowlabels EACHBOTH hitch rows Table ;
```

uses *hitch* modified by *EACHBOTH* to place the row labels on the corresponding rows of the data table to give the local variable *Labelledrows* the value

```
+-------------------------------------------------+
|None 23 34 12|Arsenic 38 18 26|Old Lace 37 13 42|
+-------------------------------------------------+
```

The subexpression

```
     Firstrow hitch Labelledrows
+----------------------------------------------------------------------------+
|Crop Yields Oats Rice Corn|None 23 34 12|Arsenic 38 18 26|Old Lace 37 13 42|
+----------------------------------------------------------------------------+
```

extends the list of labelled rows by placing the first row on the front.

The final expression

```
mix ( Firstrow hitch Labelledrows )
```

uses operation *mix* to convert the list of lists holding the rows of the result to the desired table:

```
Crop Yields Oats Rice Corn
None          23   34   12
Arsenic       38   18   26
Old Lace      37   13   42
```

Notice that there is no programming required to get the output displayed in a tidy rectangular arrangement. The array display mechanism automatically lines up columns using left justification for textual fields and right justification for numeric fields.

The example illustrates the use of data manipulation operations. The local variables (local by default) *Firstrow* and *Labelledrows* hold inter mediate values that are used only once in the subsequent lines. They serve as useful names for intermediate results and make the operation more easily understood. An equivalent operation could be defined as

```
lab IS OP Co Rl Cl T { mix ((Co hitch Cl) hitch (Rl EACHBOTH hitch rows T)) }
```

However, such a cryptic definition is more difficult to understand since the use of abbreviated names for the parameters and the defined operation make the purpose of the definition obscure, and the long single expression requires careful analysis to comprehend.

The illustration of how to display a table of data demonstrates the two major programming components of Nial: data objects are manipulated by an expres sion sublanguage that uses many predefined high level functions, and expressions in the sublanguage are combined using control mechanisms such as assignment, sequencing and function definition to provide a high level mechanism for rapid programming.

Chapter 3 Data Manipulation Language

All data objects in Nial are treated as rectangular nested arrays with dimen sionality, extent and depth. The array

```
X := 2 3 reshape (8 9) 23 'hello world' "goodbye (tell 3 2)([sin,cos]0.5)

+-----------------------------------+
|8 9    |       23|hello world      |
+-------+---------+-----------------|
|goodbye|+-------+|0.479426 0.877583|
|       ||0 0|0 1||                 |
|       |+---+---||                 |
|       ||1 0|1 1||                 |
|       |+---+---||                 |
|       ||2 0|2 1||                 |
|       |+-------+|                 |
+-----------------------------------+
```

is a 2 by 3 array of other arrays. The upper left entry (at address 0 0) is the pair 8 9 and the item at address 1 1 is the table of addresses for a 3 by 2 array. X holds a character string at address 0 2 and a phrase at address 1 0.

There are three measurements of the structure of an array:

valence	the number of dimensions (or axes)
shape	the list of lengths of the array along each dimension
tally	the number of items in the array.

The measurements are related. For every array A, the following equations hold:

```
valence A = tally shape A
tally A = product shape A
```

The definitions along with the second equation indicate that the term item refers to top level objects in an array and not the most deeply nested ones. The meaning of these measure ments for the array X is as follows:

- the valence is 2 (X has two dimen sions),
- the shape is the list 2 3 (X has two rows and three columns), and
- the tally is 6 (X has six items).

The expression for the table assigned to X involves the infix use of operation *reshape* between two lists, the first giving the shape of the table and the second being a list of the arrays to be used as the items of the table.

The syntax for a list is a sequence of two or more array expressions placed side-by-side. This construct is called a strand. A strand evaluates to the list that has as its items the values of the expressions in the strand. An item of a strand can be an atomic array, string or an expression in parentheses.

Functional Objects

Nial has two classes of functional objects: operations and transformers. Operations are the functional objects that map arrays to arrays, such as *rows*, *reshape* and *link*. Operations are either predefined (implemented by the Q'Nial interpreter) or defined by the user during a session.

Operations in Nial are said to be first order functions because they act directly on an argument array to produce a result array. Since a list of arrays is itself an array, some operations require that the argument array have a specific number of items. As well, many operations return a result that is a list of arrays computed by the operation.

The syntax for the use of an operation allows it to be used in both a prefix and an infix manner. For example, operation *link* can be used between two arguments that are lists in order to join the two lists together, or it can be used in front of an argument that is a list of an arbitrary number of lists in order to join all the lists together.

An operation is applied to an array and results in an array value. For example, applying the operation *sum* to the list 3 4 5 results in the value 12.

The notation for applying an operation in Nial is based on juxtaposition (side-by-side placement of objects). The operation is placed before the argument. The following example shows the juxtaposition syntax for applying an operation.

```
     sum (3 4 5)
12
```

The parentheses used in the above example are not necessary. Nial would give the same result if they were omitted.

The notation for infix use of an operation like *reshape* places the operation between two arguments. It is interpreted as the operation applied to the pair formed by the two arguments.

```
     2 3 reshape 4 5 6 7 8 9
4 5 6
7 8 9
```

```
     reshape ((2 3) (4 5 6 7 8 9))
4 5 6
7 8 9
```

A transformer is a functional object that maps an operation to a modified operation. It is said to be second order function, because it is a functional object that operates on one or more first order functions and produces a function. The effect of applying the transformer *EACH* to the operation *first*, for example, results in an modified operation (called the EACH transform of first) that applies *first* to each item of its argument. Juxtaposition is used to denote the application of a transformer to an operation. The application of the modified operation in

```
     (EACH first) ((2 3) (4 5 6 7 8 9))
2 4
```

is equivalent to applying first to the lists in the pair

```
    (first (2 3)) (first (4 5 6 7 8 9))
2 4
```

Juxtaposition of objects can be used in three additional ways besides its use in forming strands. Juxtaposition can occur in prefix use of an operation, infix use of an operation and in application of a transformer. To avoid ambiguity, specific rules have been adopted:

1. Strand formation takes precedence over either infix or prefix operation application.
2. Transformer application takes precedence over operation application.
3. If two or more prefix uses of operations occur in a row, as in *sum link A*, the rightmost operation is done first and its result is used as the argument of the operation to the left.
4. If an infix use precedes a prefix use, as in *A reshape link B*, the prefix application is done first.

The first rule means that *sum 3 4 5* is the same as *sum (3 4 5)* and that *2 3 link 4 5* is the same as *(2 3) link (4 5)*.

The second rule means that the expression *EACH first (2 3 4) (5 6 7 8 9)* applies the modified operation *EACH first* to the pair formed by the strand.

The third and fourth rules exist because there is no other sensible interpretation of these juxtapositions.

A modified operation can also be used in an infix manner. For example, in the operation labeltable described in the previous chapter, the expression

```
Rowlabels EACHBOTH hitch rows Table
```

uses *EACHBOTH hitch* in an infix manner between Rowlabels and the prefix expression rows Tables.

The reading of the above expression is helped by the spelling convention in Nial for names associated with the different classes of objects. An operation is spelled in lower case, a transformer in upper case and an array variable or named expression begins with a capital with the rest in lower case. These rules make it possible to recognize the kind of object associated with the name. The user of Q'Nial is not required to spell names according to these rules. When definitions are displayed by *see* or *defedit* they appear in canonical form with this prescribed spelling format. Nial is case insensitive; *first* and *First* are both legal spellings, but the former is the canonical one.

The interpretation of *sum link A* to mean *sum (link A)* given above suggests that the juxtaposition sum link could be used to denote the composition of operations *sum* and *link*, resulting in the following equation:

```
(sum link) A = sum (link A)
```

As well, it is useful to have this notation for composition in order to specify the composition of two operations as the argument of a transformer. For example:

```
EACH (link second) A
```

The interpretation of the above rules is illustrated by the following example:

```
2 3 4 EACH first sum 4 5 6
= (2 3 4) EACH first sum (4 5 6)          (strand rule)
= (2 3 4)(EACH first) sum (4 5 6)         (transformer precedence)
= (2 3 4)(EACH first)(sum (4 5 6))        (prefix operations rule)
= (2 3 4)(EACH first) 15                  (meaning of sum)
= (EACH first) ((2 3 4) 15)               (infix rule)
= (first (2 3 4)) (first 15)              (meaning of each)
= 2 15                                    (meaning of first)
```

A feature of Nial syntax is that all operations are implicitly unary and their interpretation when used in an infix way is determined by the above rules rather than by an operation precedence table. Thus,

```
2 + 3 * 4
= + (2 3) * 4
= 5 * 4
= * (5 4)
= 20
```

This example demonstrates that infix uses of operations are evaluated left to right without precedence. The same expression in Pascal or C would be evaluated by doing the multiplication first.

Bracket Notation

Nial syntax has a second way to construct a list. It is called bracket-comma notation. The expression

```
[34,275,86,-52]
```

has the same meaning as the strand

```
34 275 86 -52
```

The bracket-comma notation has the advantage that it can represent lists of length one and length zero. Thus,

```
[235]
```

denotes the list of length one holding the atom 235 as its item and the empty list Null is denoted by

```
[]
```

There is a corresponding syntactic object for operations. It is called an atlas (a list of maps). An atlas applies each component operation to the argument producing a list of results. Thus,

```
[sin,cos,sqrt] 3.14
```

is equivalent to

```
[sin 3.14,cos 3.14,sqrt 3.14]
```

An elegant example of atlas notation is to rewrite the definition of average as:

```
average IS / [sum,tally]
```

The equivalence with the earlier definition can be demonstrated as follows:

```
average A
= (/ [sum,tally]) A
= / ([sum,tally] A)                (prefix operations rule)
= / [sum A,tally A]                (atlas rule)
= / (sum A) (tally A)              (strand equivalence)
= (sum A) / ( tally A)             (infix rule)
```

The atlas notation is mainly used as a shorthand for describing operations without the need to explicitly name the argument. It is also used to form an operation argument for a transformer that uses two or more operations.

Addresses and Indexing

In procedural programming languages, arrays are treated as subscripted variables. A name declared to be an array is said to denote a collection of variables, each of which can be assigned a value independently. In such languages, there is a notation for denoting one variable from the collection, similar to the idea of using a subscript in mathematical notation. In such languages, array computations use looping constructs to do a computation that accesses all the items of an array.

In Nial, arrays are implicitly given an addressing scheme. For the table T, the addresses are given by the operation *grid* as an array of the same shape as T, with each location holding its own address.

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


grid T
+---------------+
|0 0|0 1|0 2|0 3|
+---+---+---+---|
|1 0|1 1|1 2|1 3|
+---+---+---+---|
|2 0|2 1|2 2|2 3|
+---------------+
```

The addresses of an array are integers for a 1-dimensional array (a list) or lists of integers for other arrays. The numbering scheme uses zero-origin counting. For the table T above, the addresses are lists of length two. The list A below has addresses that are integers.

```
     set "diagram;
     A := count 5
+---------+
|1|2|3|4|5|
+---------+


     grid A
+---------+
|0|1|2|3|4|
+---------+
```

The operation *pick* is the fundamental selection operation in Nial based on addressing. The expression

```
     1 2 pick T
7
```

selects the item of T at address 1 2. In picking from the list A, either an integer or a list of one integer corresponding to an address can be used:

```
     4 pick A
5
```

```
     [4] pick A
5
```

The operation *choose* can be used to select multiple items from an array. For example, in the expression below, *choose* returns the items of T at the three addresses in the left argument:

```
     [1 2,2 3,0 0] choose T
7 12 1
```

Nial also has an indexing notation similar to subscript notation in other languages. The following expression selects the item of T at address 1 2.

```
     T@(1 2)
7
```

This form of selection is called at-indexing. One difference between it and using *pick* for selection is that *pick* can be modified using a transformer because it is an operation but the @ symbol is a syntactic construct and cannot be modified by a transformer. Also the at-notation can only be used with a variable, whereas pick can select from the result of any expression.

Chapter 4 Language Mechanisms

Nial has many linguistic mechanisms that are similar to those used in other programming languages. These can be group ed under a small number of headings: assignments, selections, iterations and function construction.

Assignment Expressions

Assignment has been illustrated in many of the previous examples. An assign-expression associates a name or a list of names to a value. There are three forms involving a single variable, a sequence of variables, or an indexed variable as the target:

```
X gets count 20;
A B C := 10 20 30;
Y@3 := tell X@2;
```

In the first example, the keyword *gets* is used between the target and the value being assigned. In the second two examples, the alternative symbol *:=* is used. In the second form, using a sequence of variables as the target, the value on the right must have the same tally as the number of target variables on the left. The example showing this is valid because three variables are assigned with a list of length three. In the third example, the at-indexing notation is being used on the right to select a value from X and on the left to insert a value into Y.

For the first two forms, the result value is the value on the right. For the third form, the value of the result of the assign-expression is the value of the updated variable.

Selector Expressions

There are two selector mechanisms: if-expressions and case-expressions. The if-expression can have a single conditional expression or a sequence of conditional expressions. An optional else clause can be provided. Examples are:

```
X gets IF A = 0 THEN
  "Infinity
ELSE
  Y / abs A
ENDIF;
```

in which an if-expression is used to select between values, and

```
IF A=0 THEN
  X:=?undefined ;
ELSEIF A < 0 THEN
  X:=Y / opposite A;
ELSE
  X:=Y / A;
ENDIF ;
write link 'X is ' (string X);
```

in which a selection is made among three expression sequences. In the first example, semicolons are not placed after the expressions in the THEN and ELSE clauses so that the values of the expressions will be passed as the value of the entire if-expression, which is assigned to variable X. In the second example, each clause is an expression sequence that makes an assignment to variable X. In this case, no overall result is expected and semicolons appear after each clause.

A case-expression is also used to choose among values or actions. A simple example is:

```
X := CASE N FROM
  1: sin Y END
  2: cos Y END
  3: tan Y END
  ELSE
    fault 'unexpected value of N
  ENDCASE;
```

The value of the expression after the keyword CASE is used to select one of the clauses following the FROM. The values preceding the colons are unique constants that are compared with the selection value. Only one of the expressions is evaluated.

Iterations

There are three forms of iteration in Nial: while-expressions, repeat-until expressions and for-expressions. Each implements an explicit looping construct.

The while-loop and repeat-loop are similar to the same constructs in other languages. They are useful when there is an iteration in which it cannot be predicted ahead of time as to how many times the loop needs to be traversed.

This happens, for example, when reading from a file of unknown length, determining the value of a converging series, or examining a data structure of unknown structure. A loop to process the records of a file using sequential file operations can be imple mented in Nial as

```
File := open "myfile "r;
Record := readfile File;
WHILE Record ~= ??eof DO
  <process record>
  Record := readfile File;
ENDWHILE;
close File;
```

The for-loop in Nial differs from the for-loop concept used in most compiled languages in that the set of values used in the iteration is determined at the beginning of the loop, fixing the number of times the body will be evaluated. In the following example, the loop evaluates count 20 to the list of the first 20 integers. The body is then evaluated 20 times, first with one, then with two, etc.

```
FOR I WITH count 20 DO
  X@I := I*I;
ENDFOR
```

There is no restriction on the values used in the control array. The above loop to process the records of a file could also be programmed as

```
Records := getfile "myfile;
FOR Record with Records DO
  <process record>
ENDFOR;
```

In this case the operation *getfile* returns a list of character strings containing the records of the file. Each is processed in turn. The advantage of the first version using the while-expression is that it requires less internal storage since only one record of the file is in the workspace at a time.

Expression Sequences

All the control constructs for iterations and selections have two kinds of expressions in them: expressions that are used for control and expressions that represent the value to be returned or the action to be taken. The latter expressions can, in general, be a sequence of expressions separated by semi-colons. The expressions used for control are restricted to being simple expressions that compute a single array value in one step. However, any assignment expression or expression sequence can be made into a simple expression by enclosing it in parentheses. Thus, the while-expression version of the read loop can be written as:

```
File := open "myfile "r;
WHILE (Record := readfile File) ~= ??eof DO
  <process record>
ENDWHILE;
close File;
```

Although the embedding of assignments is permitted in Nial, it is a practice that should be used with restraint. Its overuse can make programs difficult to read.

Blocks

The block concept in Nial is borrowed directly from Algol 60 and its descendants. It is essentially the same as that in Pascal except for the treatment of new variables assigned within the block. The symbols that delimit the block are { and }. A block can declare variables to be LOCAL or NONLOCAL. Without explicit declaration that it is NONLOCAL, a variable used as the target in an assignment is automatically treated as a local variable.

```
loaddefs "block 1
average IS OPERATION A {sum A / tally A}
{ NONLOCAL X;
  X := count 25;
  Y := 5 5 reshape X;
  EACH average cols Y }
11. 12. 13. 14. 15.

X
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

Y ?undefined identifier: Y <***>

The block above declares X as a nonlocal variable. This creates a global variable X. The use of X after the block shows that the value assigned in the block has been passed to the global variable. Y is declared as a local implicitly by its use as the target in the second assignment. The attempt to use Y outside the block results in an *?undefined identifier*: fault. The value of the block, which is the value of the last expression in the block, is computed using the global operation *average* defined before the block. The actions in the above session were stored in a file *block.ndf* and brought into the workspace using *loaddefs*.

A block can also hold a definition. The following block illustrates the same example as the one above with the operation *average* made a local definition.

```
{ NONLOCAL X;
  average IS OP A { sum A / tally A }
  X := count 25;
  Y := 5 5 reshape X;
  EACH average cols Y }

11. 12. 13. 14. 15.
```

Defining a Parameterized Operation

The operation *average* defined in the previous section illustrates the use of an operation-form in a definition. An operation-form begins with the keyword

OPERATION (which may be abbreviated to OP), is followed by a list of one or more formal parameter names and then is followed by a block or an expres sion-sequence in parentheses.

The formal parameters are local to the operation form and are treated as local variables that are initialized when the operation-form is used. For example:

```
compare IS OPERATION X Y {
  Maxs := EACH max X Y;
  Mins := EACH min X Y;
  >= Maxs and <= Mins }

2 5 3 9 compare 6 4 5
l

compare [3 4 5 6, 4 5 6 7]
o
```

The name *compare* is associated with the operation-form by the keyword IS. The operation-form has two formal parameters X and Y, and its body is the block with a sequence of three expressions. The parameters, X and Y, and the variables, Maxs and Mins, are all local to the operation-form. The two actions after the definition illustrate the use of compare in infix and prefix style respectively.

When compare is used, the two arrays that are provided are called the actual arguments. The evaluation of a use of *compare* is in two steps:

1. Assign the values of the actual arguments to the formal parameters.
2. Evaluate the body of the operation-form with the parameters initialized.

The result of the application is the value of the body, which is the value of the last expression in the expression sequence in the body.

Nial's interpretation of parameter passing is equivalent to call-by-value in Pascal. There is no equivalent in Nial to Pascal's call-by-variable form, although it can be simulated using operations and variable names.

An operation-form that has only one formal parameter assigns the entire actual argument to the parameter. In Chapter 1, the operation *average* is applied to a list of numbers which are assigned to the single formal parameter A.

If *compare* is used in infix style, the actual argument is the pair formed from the two surrounding expressions. In general, an operation-form with two or more formal parameters splits the actual argument, placing the items of the argument in the parameters in a left to right order. In the latter case, the length of the parameter list and the tally of the actual argument must agree.

An operation-form in Nial corresponds to a first-order lambda-form in Lisp. As in Lisp, an operation-form can be written in line and used without being given a name. For example:

```
(OPERATION X Y { Maxs:=EACH max X Y;
    Mins := EACH min X Y;
    >= Maxs and <= Mins }) [3 4 5 6, 4 5 6 7]

 o
```

In such a use, the operation-form must be enclosed in parentheses.

Defining a Transformer

A transformer is used to modify an operation or an atlas of operations in a systematic way. Most of the general transformers needed for programming in a functional style are built into Nial. Nial provides a mechanism for defining additional transformers.

The mechanism for defining a transformer is called a transformer-form. It consists of the keyword TRANSFORMER followed by a formal operation parameter list followed by an operation-expression. TR is a short form for the keyword.

A simple example of a transformer definition using a transformer-form is

```
TWICE is TRANSFORMER f OPERATION A { f f A}
TWICE (5+) 20
30
```

The transformer modifies the operation parameter f by applying it twice to the argument. In the example, the operation (5+) is applied twice to 20 to give 30.

A transformer can also be used to describe an algorithm for a class of problems. Such an algorithm is called a schema. A typical use for this purpose is given by the following example of a transformer to do a general depth recursion on a labelled tree data structure, where at each node the first item is the label and the rest of the items form a list of nodes holding the "children". A leaf of the tree is a node with a single item.

```
RECUR is TR process_leaf process_label combine
OP Tree {
  IF tally Tree = 1 THEN
    process_leaf first Tree
  ELSE
    New_label := process_label first Tree;
```

```
    Children := rest Tree;
    New_children := EACH (RECUR [process_leaf, process_label, combine]) Children;
    New_label combine New_children
  ENDIF }
```

```
Tree gets ["Root,
            ["L10,
              ["L20, 25 ,13 ],
              ["L21, -20, 45, 15 ]],
            ["L11, 100],
            ["L12, 1000] ]

+----------------------------------------------------+
|Root|+---------------------------+|L11 100|L12 1000|
|    ||L10|L20 25 13|L21 -20 45 15||       |        |
|    |+---------------------------+|       |        |
+----------------------------------------------------+
```

The schema does a depth-first recursion on the tree data structure testing for a leaf. The operation parameter *process_leaf* is applied to the leaf if one is found. Otherwise, the label of the node is processed using *process_label*, the schema is applied recursively to each child of the node and the two results are combined using operation parameter *combine*. In a use of *RECUR*, the actual operation arguments are provided as an atlas.

```
leaves is RECUR [pass, pass, link sec ond]
leaves Tree

25 13 -20 45 15 100 1000
```

In leaves, operation *pass* is used for the first two arguments indicating that no processing is done to the labels and leaves. The composed operation *link second* is passed as the third argument and has the effect of linking the result of applying the schema to the children. The effect is to gather all the leaves of the tree into a list.

```
sum_leaves IS RECUR[pass,pass,sum second]
sum_leaves Tree
1178
```

In *sum_leaves*, the effect is similar except the child nodes are combined with the composed operation *sum second*. The effect is to form the sum of the leaves.

```
add_one IS RECUR [1+,pass,hitch]
add_one Tree
+-------------------------------------------------+
|Root|+--------------------------+|L11 101|L12 1002|
|    ||L10|L20 26 14|L21 -19 46 16||       |        |
|    |+--------------------------+|       |        |
+-------------------------------------------------+
```

In *add_one*, the effect is to form the array of the same structure as Tree with one added to each of the leaves. This is achieved by using (1+) to process the leaves, using pass on the labels and using *hitch* to rebuild the tree at each level.

```
change_labels IS RECUR [pass,?LAB first,hitch]
change_labels Tree

+------------------------------------------------+
|LAB|+--------------------------+|LAB 100|LAB 1000|
|   ||LAB|LAB 25 13|LAB -20 45 15||       |        |
|   |+--------------------------+|       |        |
+------------------------------------------------+
```

In *change_labels* the effect is to form the array of the same structure as Tree with each label replace with the phrase LAB. This is achieved by using *pass* to process the leaves, using the curried operation *LAB* first on the labels and using hitch to rebuild the tree at each level.

Chapter 5 Problem Solving with Nial

Q'Nial provides an interactive environment for experimenting while trying to solve a problem. The interactive approach allows thinking about the problem by trying steps towards a solution.

This chapter consists of a log of an interactive session used to explore a problem in graph theory. The recording of the log is started using

```
set "log ;
```

The Problem

Consider a directed graph consisting of nodes a, b, … and edges that indicate a directed link between some of the nodes. For example, consider the following graph:

```
                 +---+
   +-------->-----| b |
   |              +---+
   |
 +---+     +---+     +---+     +---+
 | a |--->--| c |-->--| g |-->--| k |
 +---+     +---+     +---+     +---+
                                 |
 +---+               +---+       |
 | d |----------->---| e |--->---+
 +---+               +---+
```

One way to represent the graph is to provide a list of edges in which each edge is a pair of nodes indicating a link from the first node to the second.

```
Edges := ("a "b) ("a "c) ("c "g) ("g "k) ("d "e) ("e "k)

+-----------------------+
|a b|a c|c g|g k|d e|e k|
+-----------------------+
```

This array describes the graph completely.

Suppose the requirement is to write an operation that, given an edgelist for a graph and two nodes Node1 and Node2, determines whether or not there exists a path from Node1 to Node2 along the links in the graph. Visual inspection of the example graph reveals that there exists a path from a to k but not one from b to d. The task is to determine how to test this finding computationally.

Testing for a Path

We begin experimenting by looking for the path from a to k. First, look for all the links that start at a.

```
     EACH first Edge

a a c g d e

     EACH first Edges = "a
o
```

Using *equal* returns only a single result. Try *match*.

```
     EACH first Edges match "a

lloooo
```

This gives a bitstring, a string of booleans, that can be used with *sublist* to extract the desired links.

```
    Lnksfrm_a := EACH first Edges match "a sublist Edges

+-------+
|a b|a c|
+-------+
```

Then, test whether or not a direct path to k has been found.

```
     "k in EACH second Lnksfrm_a

o
```

The answer is negative. Now repeat the process to see if there is a link through b or c.

```
     Lnksfrm_b := EACH first Edges match "b sublist Edges

     Lnksfrm_c := EACH first Edges match "c sublist Edges

+---+
|c g|
+---+
```

There are no links found from b and only one from c. Test to see if the link from c completes a path.

```
     "k in EACH second Lnksfrm_c

o
```

The result is negative so the process is repeated on the node to which c links.

```
     Lnksfrm_g := EACH first Edges match "g sublist Edges

+---+
|g k|
```

```
+---+

     "k in EACH second Lnksfrm_g

l
```

A path has been found. This exploration indicates that a path test can be written in this style. It will have two parts: a test to see if a direct link occurs or the use of a loop to find whether or not one of the links leads to a path.

```
pathtest is OP Edges Node1 Node2 {
  Lnksfrm_node1 := EACH first Edges match Node1 sublist Edges;
  IF Node2 in EACH second Lnksfrm_Node1 THEN
    % direct path Node1 to Node2;
    True
  ELSE
    % loop through linked nodes to see if a path to Node2 exists;
    Linkednodes := EACH second Lnksfrm_node1;
    Found := False;
    I := 0;
    WHILE not Found and (I < tally Linked nodes) DO
      Node := Linkednodes@I;
      Lnksfrm_node := EACH first Edges match Node sublist Edges;
      Found := Node2 in EACH second Lnksfrm_Node;
      I := I + 1;
    ENDWHILE;
    Found
  ENDIF }


     pathtest Edges "a "k
o
```

The test of this version shows that the algorithm fails.

```
     pathtest Edges "a "b
l


     pathtest Edges "a "c
l


     pathtest Edges "a "g
l
```

Further testing shows that the algorithm is close to working. It finds a path over two links but not over three. The problem is apparently in the loop that looks for a path from a node in the list of linked nodes.

Instead of trying to test for a path directly, it is necessary to call the routine recursively to get the path test to go down an arbitrary number of links.

```
pathtest is OP Edges Node1 Node2 {
  Lnksfrm_node1 := EACH first Edges match Node1 sublist Edges;
  IF Node2 in EACH second Lnksfrm_Node1 THEN
    % direct path from Node1 to Node2;
    True
  ELSE
    % loop through linked nodes to see if a path to Node2 exists;
    Linkednodes := EACH second Lnksfrm_node1;
    Found := False;
    I := 0;
    WHILE not Found and (I < tally Linked nodes) DO
      Found := pathtest Edges Linkednodes@I Node2;
      I := I + 1;
    ENDWHILE;
    Found
  ENDIF }

     pathtest Edges "a "k
l


     pathtest Edges "b "d
o
```

The operation has succeeded.

Returning the Path Found

An alternative solution would be to return the path if one is found. This is only a minor change in the algorithm. Instead of returning a boolean value in the direct test, the link would be returned. In the loop, an empty path would be set and it would be replaced by a found path in the loop. The first attempt is:

```
findpath is OP Edges Node1 Node2 {
  Lnksfrm_node1 := EACH first Edges match Node1 sublist Edges;
  IF Node2 in EACH second Lnksfrm_node1 THEN
    % direct path from Node1 to Node2;
    Node1 Node2
```

```
  ELSE
    % loop through linked nodes to see if a path to Node2 exists;
    Linkednodes := EACH second Lnksfrm_node1;
    Foundpath := Null;
    I := 0;
    WHILE empty Foundpath and (I < tally Linkednodes) DO
      Foundpath := findpath Edges Linkednodes@I Node2;
      IF not empty Foundpath THEN
         Foundpath := Linkednodes@I hitch Foundpath;
      ENDIF;
      I := I + 1;
    ENDWHILE;
    Foundpath
  ENDIF }
```

```
     findpath Edges "a "c
 a c
     findpath Edges "a "k
 c g g k
```

The operation works on a direct link but has a missing node and a repetition in a path of length four. The problem is caused by the path computation after the recursion has returned a path. It should be Node1 rather than the linked node that is added to the path.

```
findpath is OP Edges Node1 Node2 {
  Lnksfrm_node1 := EACH first Edges match Node1 sublist Edges;
  IF Node2 in EACH second Lnksfrm_Node1 THEN
    % direct path from Node1 to Node2;
    Node1 Node2
  ELSE
    % loop through linked nodes to see if a path to Node2 exists;
    Linkednodes := EACH second Lnksfrm_node1;
    Foundpath := Null;
    I := 0;
    WHILE empty Foundpath and (I < tally Linkednodes) DO
      Foundpath := findpath Edges Linkednodes@I Node2;
      I := I + 1;
    ENDWHILE;
    IF not empty Foundpath THEN
      Foundpath := Node1 hitch Foundpath;
    ENDIF;
    Foundpath
```

```
  ENDIF }
```

```
     findpath Edges "a "k
a c g k
```

Finding Paths in a Cyclic Graph

Now consider the situation where a graph has more than one path between two nodes. It may be desirable to compute all the paths between nodes. Consider the graph which is the same as the previous one except that it adds a link from a to d.

```
     Edges:=("a "b)("a "c)("c "g)("g "k)("d "e)("e "k)("a "d)
+---------------------------+
|a b|a c|c g|g k|d e|e k|a d|
+---------------------------+
```

The modification needed is twofold. On a direct link, the result must be returned as a solitary list holding the direct link to indicate one existing path. For the linked nodes, the loop has to be changed to check all of the nodes for paths and these have to be combined to give the list of all paths.

```
findpaths is OP Edges Node1 Node2 {
  Lnksfrm_node1 := EACH first Edges match Node1 sublist Edges;
  IF Node2 in EACH second Lnksfrm_Node1 THEN
    % there is a direct path from Node1 to Node2;
    Paths := [ Node1 Node2 ];
  ELSE
    Paths := Null;
  ENDIF;
  % loop through linked nodes to add any additional paths;
  Linkednodes := EACH second Lnksfrm_node1;
  FOR Node WITH Linkednodes except [Node2] DO
    Newpaths := findpaths Edges Node Node2;
    Paths := Paths link (Node1 EACHRIGHT hitch Newpaths);
  ENDFOR;
  Paths }
```

```
     findpaths Edges "a "k
+---------------+
|a c g k|a d e k|
+---------------+
```

The algorithm worked on the first attempt.

The limits of the algorithm can be explored. Consider the test:

```
     Cyclic_graph := ("a "b)("b "c)("b "d)("c "a)
+---------------+
|a b|b c|b d|c a|
+---------------+
```

```
     findpath Cyclic_graph "a "d
system warning: C stack overflow
```

The algorithm fails with a stack overflow. The problem is caused by the recursion going around the cycle in the graph until the internal stack in Q'Nial runs out of space. The algorithm has an implicit assumption that the chains of links will all terminate. The assumption would be valid if it is known that the directed graph does not have cycles (i.e. it is acyclic). To make the algorithm work in the general case, it needs to be altered to avoid cycles. The approach is to add a fourth parameter that records the nodes that have been encountered in pursuing a path and to use it to avoid a recursion that cycles.

The expression calling *findpaths* uses Null as the initial value for the new parameter Encountered. Encountered is a list which is extended when a new node is considered. The list is used to remove can didates from the list of linked nodes that have already been processed.

```
findpaths is OPERATION Edges Node1 Node2 Encountered {
  Lnksfrm_node1 := EACH first Edges match Node1 sublist Edges;
  IF Node2 in EACH second Lnksfrm_Node1 THEN
    % direct path from Node1 to Node2;
    Paths := [ Node1 Node2 ];
  ELSE
    Paths := Null;
  ENDIF;
  Encountered := Encountered append Node1;
  % loop through linked nodes not encountered to add paths ;
  Linkednodes :=EACH second Lnksfrm_node1 except Encountered;
  FOR Node WITH Linkednodes except [Node2] DO
    Newpaths := findpaths Edges Node Node2 Encountered;
    Paths := Paths link (Node1 EACHRIGHT hitch Newpaths)
  ENDFOR }
```

```
     findpaths Cyclic_graph "a "d Null
+-----+
```

```
|a b d|
+-----+
```

Exploring directed graph computations could continue in other ways. For example, the use of Edges is only one way to represent a graph. Another is to use a boolean table where a true in row i and column j indicates that there is a link between nodes i and j. With this representation, a correspondence between the node names and the indices is main tained.

Chapter 6 Host System interface

Q'Nial is designed as a portable language processor that runs in a very similar way on different operating systems. This design goal leads to an abstract ap proach to accessing system functions. A second design goal is to make many of the internal capabilities of the interpreter accessible through operations which permit the user to exploit the underlying mechanisms. Some of the latter operations are dependent on the specific implementation details.

Use of Files

In Q'Nial there are four ways that files can be used:

1. conventional sequential files accessed by line
2. direct access files accessed by byte position,
3. Nial direct access files where components are Nial strings,
4. Nial direct access files where components are arbitrary arrays.

The first method can be used to access and create text files that are used by other applications. The actions on the file equate a Nial string with a record of text in the file. The end-of-line indication is stripped off on input and restored on output, allowing Nial programs to work compatibly across different host systems. Q'Nial supports input/output operations to a console or terminal as accessing the sequential system files *stdin* and *stdout*. This means that *readscreen* is implemented by writing the argument to *stdout* and then doing a read of a record of data from *stdin*.

The second method is used for direct manipulation of a file using the standard Posix systems functions. A file is viewed as a sequence of bytes and two operations *readfield* and *writefield* are provided to read bytes from a file or write bytes to one.

The Nial direct access methods use two host system binary files to represent a conceptual file, one to hold the index and the other to hold the data records. An example of the use of direct files can be seen in the library file *simpledb.ndf*.

The console versions of Q'Nial also supports a window capability for character-based windows. This is integrated with the normal sequential keyboard input, screen output so that programs that use simple input and output can work with or without windows. An extensive editor capability is provided with the windowing capability.

Interpreter Mechanisms

The internal features of the Q'Nial interpreter made available to the user include:

scan	scan a string of program text into a token stream
parse	parse a token stream into a parse tree.
eval	evaluate a parse tree
execute	evaluate a string of program text
descan	convert a token stream to a list of strings
deparse	convert a parse tree to an annotated token stream
picture	create data pictures as character tables
paste.	paste a collection of character tables together
positions	give coordinates of the position of items in a picture
assign	assign a value to a variable name or cast
value	look up the data value associated with a variable
apply	apply an operation to a value the operation being given by its name or as a parse tree
update	do an indexed assignment

These capabilities permit an application to manipulate a program as data and to control its execution. For example, in a rule-based artificial intelligence system, the rules can be stored as text while the system is being designed. The operation *execute* can be used to evaluate the rules during this stage. Later, when the application has stabilized, a more efficient version can be built using *parse scan* to process the rules into parse trees and using *eval* to evaluate them. The equation:

```
execute Str = eval parse scan Str
```

which holds for every string that denotes a Nial array expression, drives this approach.

The operation *apply* can be used in combination with a selection process to permit dynamic application of one operation from a set of operations. This can be used to simulate method application in a object-oriented system.

The operations associated with the *picture* operation allow the user to build output screens and reports by pasting together component parts using the operation *paste*.