The Design of the Q'Nial
Michael Jenkins
June 2017

Q'Nial is an interpreter that implements Nial, the Nested Interactive Array Language, designed by Michael Jenkins and Trenchard More in the early 1980s. This document gives an overview of the design and implementation of Version 7 of Q'Nial. This version is intended primarily for the execution of scripts that are written in Nial in a Unix environment. A simple interactive interface is provided for debugging and testing purposes. The document describes the overall design at a high level, provides notes on individual modules and explains how the implementation can be extended by adding interfaces to other software. The software is being release on GitHub as open source software.

1. Overview

Nial is a very high level language intended for solving problems in both numeric and symbolic domains. It is based on nested array data structures as the sole data handling mechanism. The interpreter supports a workspace concept in which Nial data and function definitions are stored.

The interpreter analyzes program text written in Nial notation producing an internal form (a parse tree) that is either executed immediately or stored for later execution. Thus, the interpreter naturally divides into a front end that does the analysis and an evaluator that executes the internal form.

The evaluator itself is further subdivided into a layer that implements an abstract array machine, and a layer that implements the semantics of the language constructs. The abstract machine provides support for the nested array objects, with a heap mechanism for storage of the objects, a stack for holding references to the objects during evaluation, an object table for uniquely storing symbolic data types and a buffer mechanism used to store C values during object creation.

The "instructions" of the abstract machine are stack based. The top of the stack holds the array that is the argument to a primitive operation and the result of the operation is left on the stack. The main task of the evaluator is to walk the parse tree of the internal form of Nial code, invoking primitive operations to achieve the work of the program.

The following sections discuss the structure of the intepreter in terms of the above partition of its functionality with more detail provided on each piece of the system.

2. The Abstract Machine

The Nial abstract machine is implemented as five main components:

- memory management of array objects in the heap
- array creation with routines to fetch and store items
- a stack of array values
- an object table for unique storage of phrases and faults
- routines to manage a heap object as a buffer of C values

The predefined functional and data objects of the language are constructed using the capabilities of these components. The files *absmach.c* and *absmach.h* implement the components.

2.1 The Heap

The Nial heap is a C array *mem[]*, that is dynamically allocated on startup. Its use as a heap implemented by the following memory management routines:

- *reserve(n)*, which reserves space for an array object needing *n* words of storage, and
- *release(x)*, which releases the heap block at offset *x*.

The heap blocks are allocated on a first fit basis, with the free list maintained using a doubly linked list. Tags are kept at both ends of a free area so that adjacent free blocks can be merged on the freeing of a block without searching the free list.

The heap is allocated as an array of words of defined type *nialint*, set as 32 or 64 bit integers depending on a compiler switch. References to arrays within the heap are given as an index into the heap array stored as a *nialint*. Cross-references from one array to another also use *nialint* indices.

A heap entry never moves when viewed as an index value; however, during heap expansion, the actual address of an array can change because of entire heap moves. Thus, care has to be taken not to hold an actual C address across a call that can result in heap expansion.

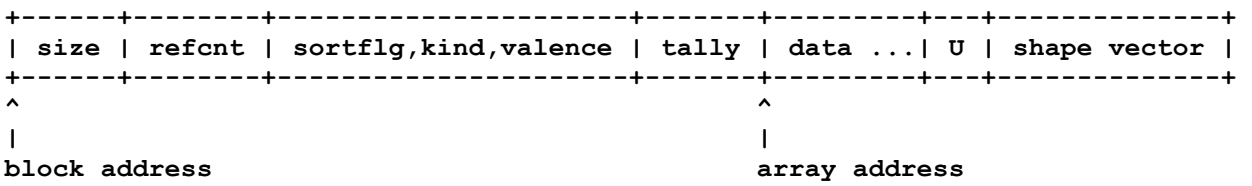
Because of Nial's scope rules, whereby most variables are automatically made local, the heap gets used largely in a stack like fashion. As a result the number of free areas remains small and most of the free space is in one block. Some operations can result in temporary fragmentation, but once the results are consumed, fragmentation reduces dramatically. No compaction algorithm is needed to maintain the heap because of this property of Nial usage.

2.2 Array Representation

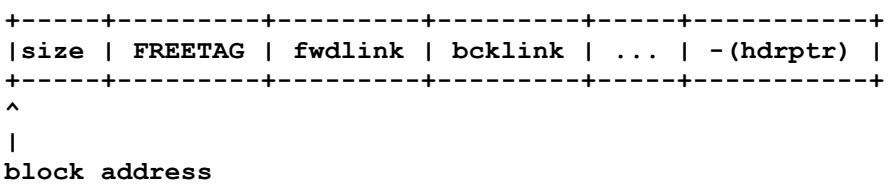
All data objects in Nial are arrays and are stored in the heap. An array has the following properties:

- valence, the number of axes (equivalent to rank in APL or dimensionality in linear algebra)
- shape, an array of *nialints* giving the lengths along each axis
- tally, a nialint that is the number of items in the array, equal to the product of the shape
- kind, the storage kind for the items of the array
- data, the list of items of the array.

The heap is viewed as a sequence of blocks that are either allocated or free. An allocated block has the following format:



where `U` represents some unused space if the block is larger than needed. When the block is free it has the form:



Memory management uses the **`block address`** in its tasks, whereas the evaluator use the **`array address`** to reference the array the block contains. The two addresses relate by

`block address + hdrsize = array address`

Using different values for these two purposes reduces the amount of index arithmetic needed on each reference to an item of an array.

The same heap design is used for both 32-bit and 64-bit implementations of Q'Nial but is paramterized to allow for the different word size. Both versions use 64-bit double precision numbers for real number values. In the 32-bit version the block header size is an even number of *nialints* to ensure that the array reference is an even integer. This avoids alignment problems on architectures that required double precision numbers to be aligned on double word boundaries.

The tally field in the header is redundant, it could be computed from the shape. It is included because it reduces overhead in many routines and can take advantage of the space needed to ensure alignment of the array pointer.

Both the block and array addresses are even integer indices for the heap array. Within the header, the third word packs in the valence, the storage kind and a flag indicating whether the array is in lexical order. The latter permits a speedup on certain array theory operations.

In the data field, the data is packed, either consisting of actual data stored as bits, bytes, words, double words, or as integer indices to other heap entries. The storage kinds correspond to the six atomic types plus "atype" which corresponds to a simple array of mixed type, an array of depth two or more, or an empty array.

The shape is attached at the end to avoid having to step over a varying sized field to get to the array address from the block address.

Any array, atom or otherwise, can be represented in this layout. For an atomic array, the storage kind is its atomic type and the valence is zero. A homogeneous simple array has storage kind corresponding to the type of atoms that it holds, but the data is packed. Simple arrays containing just phrases or just faults are not stored as packed because the items vary in representation size. Instead they are stored as "atype".

The macros that refer to array properties use the array address, the ones that refer to memory management use the block address.

The heap management is done with a doubly linked list of free areas. A free block can be detected from either end. From the front of a block, the refcnt field of the header is -1 for a free block or >=0 for an allocated array. From the end of a block, the last word is a negative number whose absolute value is the index for the block (its block address) for a free block, or the last word of the shape for an allocated array. For an array with no axes (a single, all atoms), the last word must be a zero to mark that it is allocated.

All arrays are created using the routine:

new_create_array(k,t,n,&sh)

where

k	storage kind
t	tally
n	length (for a phrase or fault only)
&sh	the address of a C array holding the shape

Array creation is also supported using the routines:

creatbool(b)
createint(i)
createreal(r)

createchar(c)
makephrase(s)
makefault(s)
mkstring(Cstr)
mknstring(Cstr,n)

The routine *freeit(x)* is used to free an array block when it is no longer used. The macros *freeup(x)* checks if the reference count is zero and if so calls *freeit*.

The array blocks are managed with reference counts. When *new_create_array* creates a new array block the reference count is set to zero. The reference count is incremented if

- the array is pushed on the stack
- the array is associated with a Nial variable
- the array becomes the item of another array
- the array is recorded in the atom table (phrase or fault)
- the array is permanently associated with a global C variable

It is decremented if

- the array is popped from the stack
- the array is replaced as the value of a Nial variable
- the array is replaced or removed as the item of an array
- the array is removed from the atom table.

An array is considered temporary if its reference count is zero. In order to protect an array needed temporarily, it is pushed on the stack. This approach, rather than incrementing the reference count directly, allows a cleanup if the execution long jumps to top level before decreasing the reference count since such a jump will trigger a cleanup of the stack. If this approach was not used, arrays allocated in intermediate routines called between the set point and the jumped point would not get cleaned up.

2.3 The Stack

The Nial stack is used only to hold references to arrays held in the heap store in the C array *mem[ ].* It is used for arguments and results of both primitive and user defined operations and to hold values temporarily. Each array reference is an integer that is an index into the heap.

The stack is stored in a heap array. It is accessed by the macros:

apush(x)	pushes *x* onto the stack
x = apop()	pops *x* from the stack
swap()	swaps top two items of the stack
top	top element on the stack
topstack	the index to the top stack item, -1 if the stack is empty
growstack()	used to grow the stack when it is full.

The stack automatically expands if an *apush* occurs when it is full.

In debug mode, which can be set up for testing purposes using the build process, the *apush* and *apop* routines are replaced by routines that check the validity of items being pushed or popped.

2.4 The Atom Table

The atom table is used as an "object" table to ensure that phrases and faults are stored uniquely. This means that comparisons between phrases can be done by integer comparison, speeding up computations involving symbolic searches. A hash function is used to compute the location for a phrase or fault, with rehashing used if collisions occur.

The atom table is an array of integers, *atomtbl[ ]* that holds either references to the storage of a phrase or fault atom in the heap or tags indicating empty or held positions in the hash table. Its length is chosen so that it is relatively prime to 239, the value used in the rehashing computation.

Since an atom has a reference count if it is held in the atom table, the free up routine removes phrases or faults from *atomtbl* if their reference count is one.

The combination of sharing of phrase values via the atom table along with the above approach to reference counting has one subtle problem. If there is a situation where two temporary values can exist that point to the same phrase, then the only reference count will be caused by the atom table. If an attempt is made to free both temporary values, then the second one will fail. This problem has been circumvented in *eval_fun.c* and *eval.c* in calls to binary and curried operations by pushing the arguments. It is believed this is the only circumstance where the above situation can occur. In all other cases, phrases being passed will be items of array structures and will have an extra reference count.

The routines that support the atom able are:

- *createatom(k,s)* create an atom of kind *k* from string *s*
- *remove_atom(x)* remove an entry from the atom table
- *hash(s)* hash the string *s*
- *reshash(n)* rebuild the expanded hash table of size *n*

The routines *makephrase* and *makefault* use *createatom*. The *makefault* routine is used to trigger an interrupt on fault creation if the triggering switch is on.

2.5 The C Buffer

Many of the implementation routines for primitives need to hold data as a set of C values before constructing a Nial array in the heap. The variable *Cbuffer* is a character array

allocated in the Nial heap used to hold temporary C values during array construction. Some routines use this in a stack like way, while others use it to accumulate a string of indefinite length. In order to use it efficiently both ways there is a reserve protocol by which enough space is allocated so that items can be pushed without the need to check for available space.

The routines to support the *Cbuffer* are:

- *allocate_Cbuffer()* allocates it
- *extendCbuffer()* extends *Cbuffer* if reserve needs it.
- *reservechars(n)* reserve space for *n* chars
- *copytoCbuffer()* copies a string to *Cbuffer*
- *copyfromCbuffer()* copies a string from *Cbuffer*

3. The Layer of Primitives

The majority of predefined functionality in the interpreter is provided by expressions, operations and transformers that are implemented directly in terms of the abstract machine. These are grouped into a number of files of related implementations.

The details of the abstract machine capabilities are abstracted by macros so that the programming does not explicitly manipulate the array representation. The use of macros allows changes in the array representation to be made with only minor impact on the rest of the code.

Each routine that implements a primitive is called a "basic" routine and can be considered to be an instruction to the abstract machine. Each such routine is named by preceding its Nial name by *i*, e.g. the basic routine for the Nial operation *sum* is *isum*. (Since these routines are all visible to the linker, a more distinctive naming convention such as NC_sum, would reduce the likelihood of name collision in uses of the code that require linking in other software.)

For operations that are defined to be "binary", e.g. expect a pair as the argument, there is also a routine beginning with *b_* , e.g. *b_plus*, to provide efficient calling of binary operations.

The general form of a basic routine is:

```
void i<name>()
 {
    < code to compute the result >
    apush( <result> )
   < cleanup code >
 }
```

For a basic expression, there is no argument on the stack; for a basic operation the stack contains the argument; for a basic transformer routine the stack contains both the function argument and the array argument.

The following code is the code for the Nial operation *first* from the file *atops.c.*

```
/* implements first
   Rules:
     atomic A => first A = A
     nonempty A => first A = 0 pick list A
     otherwise, first A = ??address
*/

void
ifirst()
{
  nialptr     x,
              z;
  x = apop();
  if (atomic(x))
    z = x;
  else if (tally(x) > 0)
    z = fetchasarray(x, 0);
  else
    z = makefault("?address");
  apush(z);
  freeup(x);
}
```

The first action of the routine is to pop the argument into variable *x*, and then to use it for the atomic test and the nonempty test. In both cases the result is placed in variable *z* and pushed on the stack. Then cleanup is done by calling *freeup* on the argument.

If the argument is a temporary array (*refcnt = 0*) then it is freed immediately; otherwise, the call to *freeup* has no effect. The call to *freeup* must be done after the call to *apush* because the result may be the same array as the argument.

Operations that are more complex are often programmed in two layers: the basic routine, beginning with *i*, is used to pop the argument into a C variable. Checks on the validity of the argument may be done at this level. Then a C routine taking an argument is called. Usually, the latter routine will push the result and clean up the argument and temporaries. The support routine may also be called from other places in the code.

See *ipack* in *atops.c* for an example.

For operations that are binary, there are two basic routines: an "*i*" routine that assumes the argument is a pair (and checks that this is true) and a "*b_*" routine that takes two arguments off the stack. Both of these call the same support routine to do the work.

The basic routines are in a number of files:

arith.c	- arithmetic operations, random number generator
atops.c	- array theory operations

compare.c - comparison primitives, max, min
linalg.c - the linear algebra operations
logicops.c - the boolean operations
picture.c - the array diagramming operations
wsmanage.c - the workspace management operations

Routines that support basic Nial expressions, such as *Null*, *Pi*, Readchar, etc. are very similar to the basic Nial operation routines except that there is no argument to pop off the stack.

These routines are not gathered in one file, but appear in the modules to which they are most closely related. For example, *ireadchar()* is in *fileio.c* with the code for reading a single character from the standard input stream.

Routines that support basic Nial transformers are also similar to operation routines except that there is also an operation argument passed on the stack. For example, the following code implements the transformer *FOLD*.

```
/* implements the transformer FOLD as
    n FOLD f x   ==   f f f...f x  (n applications) */

void
ifold()
{
  nialptr     f,
              z,
              x,
              y;
  nialint     n,
              i;

  f = apop();
  if (kind(top) == faulttype && top != Nullexpr &&
      top != Eoffault && top != Zenith && top != Nadir)
    return;
  z = apop();
  if (tally(z) != 2)
  {
    apush(makefault("?argument to a FOLD transform must be a pair"));
  }
  else
  {
    splitfb(z, &x, &y);
    if (kind(x) != inttype || valence(x) != 0)
    {
      apush(makefault("?first argument of FOLD must be an integer"));
      freeup(x);
      freeup(y);                  /* in case they are temporary */
    }
    else
```

```
    {
      n = intval(x);
      freeup(x);
      apush(y);
      for (i = 0; i < n; i++)
        apply(f);
    }
  }
  freeup(z);
}
```

The routine begins by popping the operation to be applied into *f*. This is a reference to the array holding the parse tree associated with the actual operation being transformed.

Then the array argument is popped after testing whether it is a fault. It is checked to have two items which are split into *x* and *y*. The variable *x* is tested to be an atomic integer array and its integer value placed in *n*.

The work of the routine is done by pushing *y* and then using a for-loop that applies the operation f *n* times, leaving the final result on the stack.

Before exit, the array argument is freed. The function argument is not freed because in most cases it will be permanent. If the function argument is temporary it is freed by *apply_transform()* in *eval.c*.

4. The Translator to Internal Form

The front end of the Nial interpreter consists of a hand-built scanner and a hybrid parser. Because Nial definitions are usually fairly small, the decision was made to decouple the scanner and parser into separate passes; thus the scanner is given a string to scan and produces a token stream. The latter is a flat array of alternating numbers and phrases where the numbers indicate the type of token and the phrase contains the text of the token. The parser takes a token stream as its argument and produces a nested array representing a reduced parse tree as the result.

4.1 The Scanner

The module *scan.c* implements the scanner. The scanner is driven by a finite state automaton consisting of a character class table and a state transition table. The scanner is quite efficient; the only drawback to this approach is that adding new "special symbols" such as ":=" requires carefully addition to the transition table layout.

The scanning process works as follows. The scanner starts in *Start* state. The *class* of the next character is found and the transition table is used to select a new state as indicated by the *[Start,class]* entry. This process is repeated until the state becomes *Accept* state. Then the

string isolated between the *Start* and *Accept* state is selected and the routine *mktoken()* is called to construct the token and add it to the stream. The macro definitions for the state names, the character classes and the token numbers are defined in the file *states.h*.

The basic Nial operation *scan* can call the scanner directly within Nial.

4.2 The Parser

The module *parse.c* implements the parser as a hybrid of a top-down recursive-descent parser for the linguistic aspects of the grammar (control constructs, assignment, operation and transformer forms, etc.) and a bottom up shift-reduce parser for the juxtapositional syntax of array theory expressions. This design was chosen because of the ease of adding reasonable error messages for the linguistic parts. The use of the hand built shift-reduce component allows very general use of parentheses, which were difficult to achieve in a pure recursive-descent version.

The parser is also unusual because it does back up to undo some parsing steps in situations where the construct cannot be recognized from the next token. This backup has to be done carefully, otherwise a situation can arise where n! attempts are made to parse a strand of length n before it fails. (We learned about this the hard way!).

The parser is called in two modes: one in which it is expecting an action and one when it can parse a construct corresponding to an array expression, an operation expression or a transformer expression. The second mode is only used when it is being called from Nial using the basic operation *parse*.

Each recursive-descent routine is passed a pointer to a variable in which it stores the parse tree that it constructs. The calling routine then uses the variable to access the tree and embed it in the result the caller is creating.

The shift-reduce part is driven by the routine *formfinder()*. The task of this routine is to look for one of the three kinds of expressions and return with the kind it has found. The parser uses a stack to hold intermediate results as it is parsing in shift-reduce mode.

All the parser routines return a *SUCCESS*, *FAIL*, or *ERROR* code. On *ERROR*, the routine that found the error places an error fault on the Nial stack and all routines higher up in the calling sequence return with *ERROR*. On *FAIL*, the called routine cleans up to its point of call, restoring tokens if necessary by backing up the nexttoken indicator. The calling routine decides whether to try an alternative construct, to return an *ERROR*, or to *FAIL* itself.

The token stream is global to the parse routines. The routine *accept1()* is used to accept a token and move the token indicator to the next token.

The parse trees are constructed as tagged nodes, with each form of language construct having its corresponding node form. The details of the nodes are hidden in node builders (in *blders.c*,

blders.h) and field selectors (in *getters.h*). New nodes are easily added to accommodate an extension to the language.

5. The Evaluator

Every Nial expression denotes either:
- an array value,
- an operation, or
- a transformer.

The array expressions include:
constants
variables
basic named-expressions
user-defined named-expressions
list constructs
control constructs
blocks
assignments
operation applications
definitions

The operation expressions include:
basic named operations
user-defined named operations
operation compositions
atlases e.g. *[f,g,h]*
opforms e.g. *(op a (a+1))*
curried operations e.g. *1+*
transforms e.g. *EACH rest*

The transformer expressions include:
basic named transformers
user renamings of transformers
user-defined named trforms e.g. *TWICE is tr f (f f)*

For each class of semantic object there is one routine in the evaluator that handles it:

array expressions	- *eval(exp)*
operation expressions	- *apply(op)*
transformer expressions	- *apply_transform(tr)*

These are the three major routines of the evaluator. They take as their argument a parameter corresponding to the code they are to evaluate. They assume that a code argument is "permanent" ,i.e. it is owned in some component of a program text. Hence these routines do

not need to free up such an argument. Accordingly, routines that create temporary code objects (some cases in apply_transform and some primitive transformers) are responsible for cleaning up such objects.

The evaluator is stack-based. The stack handling is as follows:

eval(exp)	leaves its result on the stack.
apply(op)	takes the array argument on the stack and leaves the result on the stack. It frees up the argumentif it is temporary.
applytr(tr)	takes the array argument on the stack (topm1) and the operation argument on the stack (top). It leaves the result on the stack. It frees up the argument if it is temporary. It does not free up the operation argument since it is assumed to be permanent code; if it is temporary, then the routine that built it must clean it up.

Each of the routines is driven by a case switch with the semantics of evaluation for each node encoded in the section of code selected by the node's tag.

The stack is used to hold other information as well. Blocks, opforms and trforms all define local scopes. When one of these is entered an activation record is placed on the stack which holds the backpointer to the previous one for this scope (-1 if there isn't one) and the value cells for the local names. The routine *prologue* sets up an activation record and *epilogue* discards it.

In order to get the proper semantics for trforms, their operation argument must be "closed with its environment". This means that the pointers to all local scopes must be saved. When a closure is applied the stack is used to hold the current stack pointers while the ones bundled with the operation are installed. The routines *setup_env* and *restore_env* are used to set up for a closure application and to clean up after it.

The evaluator is complicated by the fact that the user debugging capability is intertwined with the semantic actions. In order to make the interpreter more efficient for production use, two versions of *eval()* are created. When user debugging is enabled the slower version is run; when user debugging is off the version with the debugging code removed is run. To support this double construction of the *eval()* routine, it is stored in the file *eval_fun.c* and included twice with different switch settings.

The interface between the evaluator and the basic routines, which behave as abstract machine instructions, is done through a pair of dispatch tables and an initialization routine stored in *basics.c*. The initialization routine adds symbols and dispatch table indices to the global symbol table for each basic routine. During evaluation, when a parse tree node corresponding to a basic routine is encountered its index is selected and the dispatch table is used to do a call on the corresponding routine. The initalizer and the dispatch tables are kept in synchronization by generating them using the Nial package builder program.

6. The Symbol Table Mechanism

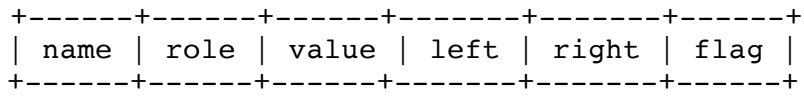
The symbol table mechanism supports a collection of binary trees, one for each separate naming scope created in the workspace. The initial environment consists of a single binary tree containing the reserved words and the names of the predefined arrays, operations and transformers. This binary tree is called the *global_symtab* and every environment has access to it. As the user defines objects and does assignments the *global_symtab* grows.

A separate binary tree is created for every naming scope as they are encountered by the parser. At all times *current_env* contains the list of symbol tables that form the static nesting of environments. The global and system symbol tables are held separately to reduce overhead. The current environment for each construct that defines a new symbol table is saved in the parse tree entry for the construct so that it can be reestablished at runtime in order to support dynamic symbol lookup as required by the semantics of *execute* and other evaluation operations.

A symbol table is a triple consisting of a root, a current stack pointer and a property. It is referenced by the index of where it is stored in the list of symbol tables. The property of a symbol table is either *global*, *parameter*, *open* or *closed*. The local symbol table of an operation form whose body is a block is *closed* otherwise it is *open*. An opform with its body in "(" and ")" is open, but if it uses "{" and "}" then it is *closed*. The symbol table of a block is *closed*. The open vs closed information is used to enforce the static nesting rules. The symbol table of a transformer form is of type *parameter*. The global symbol table is of type *global*.

The stack pointer of a symbol table points to the current activation record associated with that symbol table if any. The global symbol table does not have an activation record since its value cells are in the symbol table entries. For other symbol tables the corresponding field in an entry holds the offset to the value cell in the activation record.

Each symbol table entry has 6 fields:



name	The print value of the symbol stored as a phrase (in upper case).
role	The semantic meaning of the name. Either reserved, identifier, variable, expression, operation, or transformer.
value	The value of the object if the system or global symtab. For a variable it is the array value, for an expression, operation or transformer it is the parse tree that represents it. For a local symtab this contains the offset for the symbol in a local area on the stack.

left	The left child of the node in the binary tree.
right	The right child of the node in the binary tree.
flag	Used for global symbols to indicate whether the symbol is system or user defined.

In addition two debugging flags are inserted into the role field for symbol table entries corresponding to user defined program objects.

trflag	This is patched into the role field to indicate that the object should be traced during evaluation.
brflag	This is also patched into the role field to indicate that a break should be raised when the object starts evaluation.

The symbol table entry could be compressed to 5 fields by combining the role, the system flag and the debugging flags into one word, but no space would be saved due to heap blocks being of even size.

A symbol table has three fields:

- the root of the binary tree,
- a pointer to the current activation record in the stack for local environments, and
- a property field that indicates whether it is global or local and if local whether it is open, closed or a parameter table.

The main routines in the symbol table module are:

MkSymtabEntry	creates a symbol table entry
erase	removes a symbol table entry
lookup	returns the symbol table index and the entry address.

The lookup routine searches the symbol tables in an order to support local scope rules. It has a parameter *searchtype* that was used to control the order of search.

When a symbol table entry is stored in a parse tree node, the component is stored as an integer and not an array reference. This is necessary to break the circularity of array references for recursive definitions. The array operations of Nial assume that all arrays are represented as trees. The inclusion of a circular reference would make some internal routines loop indefinitely.

7. Workspace Management

The file *wsmanage.c* has the routines that load and store workspaces to the file system and also the routine, loaddefs, which loads and executes a Nial definition file (.ndf).

The workspace is dumped in 3 segments:

- a struct of C values that reference the workspace
- the atom table
- the allocated blocks in the heap.

The Nial stack and the C buffer parts of the abstract machine do not need to be saved. The allocated blocks are grouped into contiguous chunks, each of which is written with one writeblock call.

Since all internal heap references are offsets to the beginning of the array mem[], there is no need to scan the heap blocks on either writing or reading them. The decision not to store hard C pointers in the heap is a good one in that it has made workspace file management easier.

A workspace *save* or *load* is always done at the top level. This is necessitated by the fact that computation cannot be resumed at an arbitrary point in the C program since the C stack cannot be reinitialized and control given to the point of interruption.

8. File Access

The file *fileio.c* supports the file access mechanisms of Nial. These include i/o to stdin and stdout, sequential file access using the Nial operations *readfile* and *writefile*, the random access mechanisms of the Nial component file system for direct access (2 kinds), and routines for direct access to POSIX style files.

Most of the file routines assume that the file is open and that a file handle has been assigned. The file handles are managed in a C table of fixed size. (This is a limitation we may want to remove. For some OS's it is safe as long as the default size is larger than the number of allowable open files.)

For Nial direct access files, two host files are needed: one for the data file and one for an index file. The Nial name is given without a suffix, and the host names have *.rec* and *.ndx* as the suffices.

The *iopen()* routine is used to open files in one of the following modes:

- *"r* read only mode for sequential files
- *"w* write mode for sequential files
- *"a* write mode for sequential files starting at current end
- *"d* direct access mode
- *"c* communications mode (allows read and write)

The POSIX style file access routines take a file name argument and do the open and close internally.

The Nial direct access files are in two flavours: ones with items that are Nial strings treated as arbitrary byte arrays, and ones with items that are representations of Nial arrays.

A direct access file has an index file (.ndx) that contains some global information on the file and a 2 word field for each record up to the highest index used. The 2 words are byte position and byte length.

The data file for a direct access file (.rec) is an uninterpreted sequence of bytes. The data is stored in the order written. Updates are put in place if they will fit; otherwise they are placed at the end of the file. A count of unused space is kept in the global information. If the space wastage becomes too high the .rec file is compressed automatically.

While care has been taken in writing the direct access package, it is not as robust as professional database packages in handling error conditions. It has proven adequate for prototyping and small applications, but it is not intended for a large-scal application that depends on the direct access code as a central feature.

9. The Operating System Interface

In Version 7 of Q'Nial the interface to the operation system has been simplified since we are assuming that the operating system is a variant of Unix. Currently, we are building the sysem under Linux and Mac OSX. The operating system interface routine is *unixif.c*. It provide support for the following functionality:

- signal handling for floating point exceptions, Ctrl C, etc.
- routines for cpu time and date handling
- file handling support (open, close, seek)
- routines for sequential i/o of an arbitrary size
- routines for block i/o of an arbitrary size
- routine to call the command interface for UNIX
- routine to call an editor that can be specified
- routine to initialize the NIALROOT path

The file handling approach is based on the buffered versions of the UNIX runtime routines.

10. The Top Level Routine in V7 Q'Nial

The file *main_stu,c* is the main control routine for the interpreter. It's job is to:

- set the initial values for global variables
- process the command line input
- initialize the interpreter
- provides support for the longjmp behaviour during startup
- if starting with a named workspace, load it otherwise create a clear workspace.
- provides support for the longjmp behaviour during Nial computations
- initialize the file system
- execute a *Latent* expression if found in the workspace
- load a definitions file if requested using *loaddefs()*
- if interactive execution requested enter the top level loop
- after execution via Latent or interaction do cleanup and exit

The top level loop interacts with the user with the sequence:

- prompt
- read a line of input
- execute it, using *scan()*, *parse()*, *eval()*
- compute the picture of the result using *picture()*
- display the result using *show()*

The loop is interrupted if *Bye* is executed or the user types Ctrl-C.

All interrupts caused by the user, due to faults, or programmed ones result in a long jump to the main loop routine to code that cleans up intermediate data and restarts the loop.

Errors or interruptions that arise during execution are handled in two ways: Either they call the routine *exit_cover()*, which does the amount of cleanup necessary for the class of error, or they long jump directly following the *set_jmp()* call in *main_stu.c*. The routine *exit_cover()* also long jumps to *main_stu.c* after doing its cleanup work. If the interruption occurs in initialization it goes to the first *set_jmp()* call. If during the interactive loop it goes to the second one.

The return code provided to *set_jmp()* indicates what kind of situation caused the interruption. The details of this process are quite subtle. We do not permit *exit_cover()* to recur since it could easily result in an infinite loop.

Another complicating factor is that *load* and *save* have to be executed at top level. They result in an interruption that in the case of *load* looks for a *Latent* expression that can be used to restart the computation.

11. Coding Conventions

A standard style has been used for the modules of the Q'Nial interpreter. The interpreter is viewed as a collection of modules in the software engineering sense. C does not directly support modules; however, by following the organizational guidelines provided below, most of the advantages of modules can be achieved.

Each module consists of a code file *.c* and a corresponding header file *.h* . A module is a collection of related routines. The routines are grouped to minimize the number of cross module linkages by either routine calls or shared global variables. Minimize is used informally here, as no formal tools have been used to achieve the modularization.

The code file consists of both local and exported routines, and local and exported global variables. A local routine or variable is declared to be static and if it is a routine its prototype is provided near the top of the file. The exported routines and global variables are described in the header file for the module using an extern declaration for variables and a prototype for the routines. Evrey code file for a module include its header file to ensure that the prototype does not conflict with the actual definition.

The header files are also used to define constants and pre-processor macros that can be used either within the code file, or by other modules that need them to use the routines exported from the module.

Most of the modules are organized as follows:

Copyright comment
include of "switches.h"
include of C libraries needed in th emodule
include of other Q'Nial modules headers as needed
declaration of prototypes for static routines
global variables, declared static or not
the routines of the module

If the module is optionally included by the package builder, described below, there is an enclosing #ifdef <feature name> #endif pair after the include of "switches.h" that eliminates the module's code if the package is not selected by the package builder.

Most of the modules also include a fair number of header files for other modules. These correspond to module dependencies. Most of the modules require the header files:

qniallim.h	- size limitations
lib_main.h	- the global struct G, and other globals
absmach.h	- for the abstract machine model
if.h	- for the host system interface

When other modules are needed, we have indicates in comments why they are required so that subsequent shifts of code may be considered to reduce cross-module dependency.

12. Writing new basic expressions, operations, or transformers

This section describes how to add new basic capabilities to Nial by writing the C code to execute the semantics of basic expressions, operations or transformers. Adding new basic capaibilities in V7 Nial is done by adding a module that can be included using the pkgblder described in the next chapter.

First we describe the form that basic routines must take and give an example with some explanation.

A basic routine is always of the form

```
void i<name> () {
   ... code to generate the result  ...
   apush(result);
}
```

where `<name>` is the Nial name of the basic capability.

If it is a basic expression, then there is no argument provided on the stack. If it is a basic operation, then there will be an argument on the stack. This is retrieved by a call to `apop()` and must be freed after computing and pushing the result.

```
void i<name> () {
   nialptr x = apop();
   ... code to generate the result using x ...
   apush(result);
   freeup(x);
}
```

If it is a basic transformer, then there will be a function argument as well a value argument on the stack. They are retrieved using `apop()`. The value argument must be freed after computing and pushing the result. The function argument is never temporary and so does not need to be freed.

```
void i<name> () {
   nialptr x = apop(),
           f = apop();
   ... code to generate the result using x and applying f  ...
   apush(result);
   freeup(x);
}
```

The result can be computed by applying other basic capabilities, or by allocating a result container and filling it. An example of the first method is:

foo is op a { first rest a }

which can be implemented by:

```
void ifoo ()
{ irest();
  ifirst();
}
```

There is very little gain in efficiency by implementing *foo()* at the C level compared to using the Nial definition.

Consider the operation *rotate* defined in *defs.ndf* by:

```
rotate IS OPERATION N A {
   Ta := tally A;
   shape A reshape (Ta + N + tell Ta mod Ta choose list A)
}
```

A basic version of it written in C can be implemented by:

```
void irotate()
{ nialptr x,z,a,b;
  int vb;
  nialint tb,n,m,j;
  x=apop();    /* x is the argument */
  /* check that x is a pair */
  if (tally(x)!=2)
  { apush(makefault("?rotate expects a pair"));
    freeup(x);
    return;
  }
  /* get the items of x and store them in a and b */
  splitfb(x,&a,&b);
  /* check that a is an integer */
  if (!atomic(a) || kind(a)!=inttype)
  { apush(makefault("?first arg of rotate must be an integer"));
    freeup(x);
    return;
  }
  n = intval(a); /* get the C value of the integer */
  /* create the result container z of the same kind and shape as b */
  vb = valence(b);
  z = new_create_array(kind(b),vb,0,shpptr(b,vb));
  /* do the rotate as two copies */
  tb = tally(b);
  /* compute j, the position in b where first item of z is selected*/
  j = (n + tb) % tb;
  m = tb - j;  /* length of first move */
  copy(z,0,b,j,m); /* copy the first piece of length m */
  copy(z,m,b,0,j); /* copy the second piece of length j */
  apush(z); /* push the result */
```

```
  if (kind(x)!=atype)
     /* if x is homogeneous, then a and b are temporary */
    { freeup(a); freeup(b);}
  freeup(x); /* freeup the argument */
}
```

The above example illustrates a number of key concepts.

1. The argument of the operation must be checked to ensure it has the correct structure. In this case the argument must be a pair with the first item being an integer. The following support functions are used in testing the argument:

apop() - pop an array value from the stack
tally() - returns the number of items in the array argument
makefault() - generates a fault object from its string argument
splitfb() - used to extract the items from a pair
kind() - returns the storage kind for the object
atomic() - indicates if the object is an atom

2. We must extract the data inside a Nial object to operate on by C. Here we get the rotation amount from array a using:

intval() - returns the integer stored in a Nial atom of kind inttype.

3. The result container is allocated using:

new_create_array(k,v,0,sh) - creates a container for an array of kind *k*, valence *v*, and shape *sh*
valence() - valence of the object
shpptr(x,v) - C pointer to int for the shape vector of *x* of valence *v*

4. The result container, *z* could have been filled by a loop that selects items from *b* and places them in *z* one at a time. However, since except at the wrap around boundary, adjacent positions are moved to adjacent positions. This suggests the faster technique of doing two copies using

copy(dest,dstart,src,sstart,cnt) - copies *cnt* items from *src* to *dest* using the two starting positions

5. The result container, *z* is pushed on the stack and the argument is freed in case it is a temporary. Note that the act of extracting *a* and *b* from *x* by *splitfb* can create temporaries if *x* is not of storage class *atype*.

apush() - put array onto the stack
freeup() - test if the argument has *refcnt==0*, if so free its storage

Every basic operation is assumed to consume its argument if it is temporary. Thus, you must free up the argument at the end of the basic operation unless you have used it in a call to another basic operation. Note that if you still need the argument after calling another basic operation, you must protect it during a primitive call as follows:

```
apush(x); /* to protect x */
apush(x); /* x is arg to ishape */
ishape();
shx = apop();
apop();  /* to unprotect x */
```

There are many other support routines available in *absmach.c*. They are either routines or macros as indicated in the *absmach.h* file. These include routines:
to fetch and store to arrays of the various storage kinds,
explode() to convert a homogeneous array to an atype one,
implode() to do the opposite,
type testing routines, and
additional stack support routines.

Examples of their use can be found be examining the primitive routines provided with Q'Nial.

13. Building the executable

The GitHub repository, QNial7 provides executable *nial* binaries for several platforms. See its *README.md* file for details on how to set one up for use on your system. It also provides two directories *BuildCore* and *BuildNial* that are used to create an executable binary on a different platform or to build an executable *nial* with a different choice of features.

If you want to create an executable on an unsupported platform, then you use *BuildCore* to create a core version called *nialcore*. After creating it, you copy it to *BuildNial*. The *README.md* files in each directory give the details of how to build the desired executable.

For a new platform, you use *nialcore* to build a basic version, *nial_basic* that adds definitions written in Nial to the core, and then use *nial_basic* to build a package that contains the features you want to add to the core. The Nial script *addfeatures.ndf* is used to gather the source files needed for the build and to generate a *CmakeLists.txt* that is used by CMake in the build. If you want to build an executable *nial* with a different choice of features you can use the provided *nial* with *addfeatures.ndf* to build a package that has the features that you want. See the *README.md* in *BuildNial* for details.

The *pkgblder* directory and the *addfeatures.ndf* Nial script have been designed to make it straightforward to add new features to QNial7. The term *feature* is used in this context to describe an extension to the Nial interpreter that involves the addition of one or more new primitive functions each of which may be *a basic expression, a basic operation, or a basic transformer.* The new primitives are implemented in C following the instruction give in Chapter 11 above. The implementation may involve one or more C source files and any corresponding header files they require. A *feature* may also add additional functionality in the form of Nial definitions that are installed at startup of the executable. The *README.md* in *BuildHelp* has the details.