A DEVELOPMENT SYSTEM FOR TESTING ARRAY THEORY CONCEPTS

M. A. Jenkins
Computing and Information Science
Queen's University
Kingston, Canada K7L 3N6
(613) 547-2784

This research was started during a leave of absence at the IBM Cambridge Scientific Center in 1979 and continued since that time under a consulting agreement with IBM.

Abstract

An experimental programming system has been written in APL to allow exploration of array theory and its use in a programming environment. The system combines array theory notation with programming language constructs in a simple but powerful language that has both applicative and imperative aspects. The system builds on the work of Hassitt and Lyon, using a shared variable interface to access their low level implementation of array operations. The system allows a small number of primitive operations to be chosen from which one builds other definitions. It has been used to study two different models of the theory, one based on set-theoretic intuition following More's series of papers, and a second that uses recursion to build definitions much in the style of Lisp. This paper describes the overall design, how it has been used and some of the internal algorithms.

1. Introduction.

Array theory is a model of data that combines concepts from set theory, APL and Lisp in one unified theory of data [Mo79]. Its inventor Trenchard More, views data collections as physical entities with their geometrical arrangement determining equations that hold everywhere. The development of the theory has been driven by the search and discovery of equations that characterize the underlying "physics" of data. Over the years the theory has undergone a number of major revisions each one triggered by the discovery of a new equation. The potential importance of array theory as an evolutionary step from APL was recognized by W. Bouricius who implemented an early APL model of the theory [Mo75]. Later A. Hassitt and L. Lyon implemented a package of array theory operations as a means of supporting the theory efficiently while accessing it from a high level language [Ha79].

This paper describes a development system, called Nial, written in APL that is being used to assist in the development of array theory concepts by exploring their use in a programming environment. It has been designed in a flexible way to allow future developments to be incorporated with ease. It builds on the AT/370 package described in [Ha79,Ha81] using a shared variable interface to connect the implemented array theory operations with an evaluator written in APL.

Originally Nial was created as a vehicle to test the AT/370 routines; however, it became apparent very quickly that it could be used to demonstrate array theory concepts and to allow the definition of operations and operators that were not implemented directly in AT/370. Thus, a two week effort to write a quick and dirty evaluator, turned into a two year effort that has produced a working high level language development system. Here, we will describe what Nial is, its overall design, how it has been used and some details on how it is implemented.

2. A description of Nial.

Nial is implemented by a VSAPL workspace that presents an array theory programming environment to the user. It is, in essence, an interpreter written in an interpreter. Figure 1 gives a sample session.

The user can input expressions, statements, definitions and system commands. The first 3 inputs of the session are expressions. An operation has a name and some have abbreviations as symbols. Either may be used in any context. An expression is evaluated and displays the result as an array diagram. "tell" is the generalization of the index generator function of APL. Its items are themselves arrays; but they are not boxed since we supress the boxing of the simple homogeneous arrays as a default convention.

System commands are used to control attributes such as the display format, to edit or display definitions or to interface with the CMS file subsystem. The session illustrates the use of commands to switch between default and full boxing in diagrams and the use of the SEE command to display definitions.

A statement is either an assignment statement, an iterative statement, or a sequence of statements. It is executed, having some side-effect on the state of the system. For example, the assignment statement to "A" associates with that name in the global environment the list of 4 items given in strand notation (the expression following the arrow). An expression can consist of a statement sequence followed by an expression as the example involving "B" and "C" illustrates.

An operation definition uses a notation similar to lambda notation; however the semantics are different from the usual Lisp usage. Names used in an operation definition are treated as strictly local variables or parameters unless they refer to operations or operators. "foo" is defined to be the increment by one function. Notice that on display the function is put in a canonical display format in which variables begin with a capital, operations are lower case, operators are all upper case and reserved words are underlined (or boldface). These are output conventions adopted to ease readability of Nial definitions. All fonts are treated as equal on input except in character strings and phrases.

An operator definition is a parameterized operation definition with a special header that specifies the adicity of the parameter(s). "FOLD" is an example of an operator that converts a monadic operation, formally denoted by "f", to a dyadic operation which uses "f" in its lambda form. The definition implies that FOLD f applies f to the right argument as many times as indicated by its left argument, which normally would be an ordinal number.

```
CLEAR WORKING AREA

   1+1
2

   1 plus 1
2

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

   ]FULLBOX

   A←(tell 3) ('APE') 4 (5 6)
   A
 -----------------------------
| ----- | ----------- |4| --- |
||0|1|2|||'A'|'P'|'E'|| ||5|6||
| ----- | ----------- | | --- |
 -----------------------------
   ]BOX

   B←3;C←4;B+C
7

   FOO IS ∇A(A+1)

   ]SEE FOO
foo is ∇A(A+1)

   FOO 3
4

   ]SEE FOLD
FOLD as ∇1 f ∇A.B(if mult A = 0 then B
           else bate A FOLD f f B)

   5 FOLD FOO 8
13

   ]OFF
```

Figure 1. A sample Nial session.

The system is used by entering definitions and variables in the global environment and invoking operations by the use of statements or expressions. The linguistic mechanisms available in Nial include expressions, input and output capability, an assignment statement, grouped statement sequences, an if-then-else construction, a loop construction, and the definitional mechanisms described above. For a more complete introduction to Nial and its use see [Je81], [Bo81].

3. Overall design of Nial.

The Nial workspace interfaces with the CMS file subsystem and AT/370 through shared variables. Figure 2 gives an overview of the design.

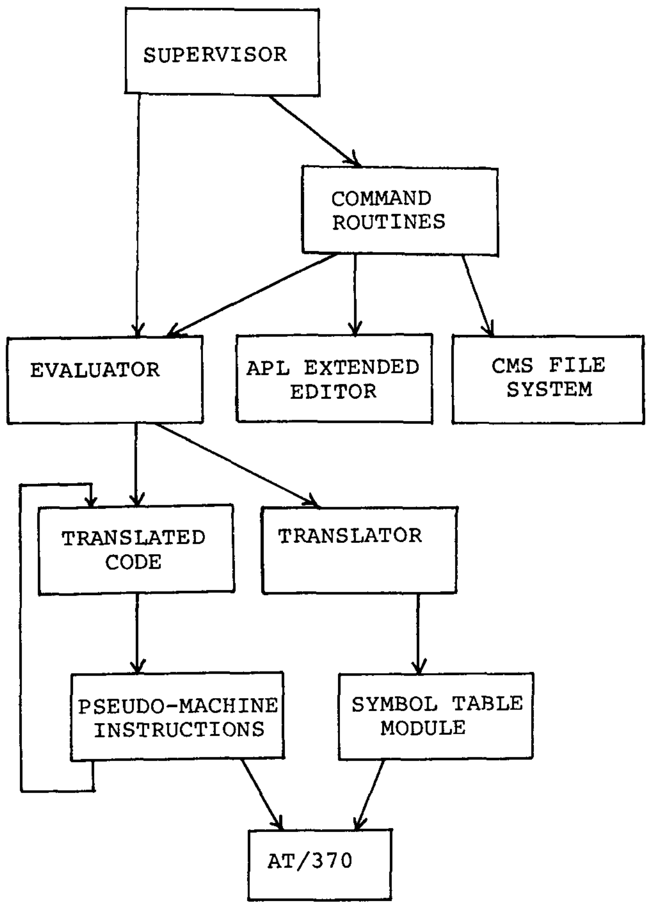

Figure 2. Control Flow in Nial

The supervisor is a simple looping routine that accepts expressions, statements, definitions, or commands from the user. If the input is a command it is handled by the appropriate command processor routine, which either carries out the action directly, uses the evaluator to establish a definition, or interacts with CMS. One of the possible command actions is to use the VSAPL Extended Editor to edit data, definitions, or files.

If the input is a definition, then the evaluator calls the translator to compile the operation or operator into an APL function pointed at by the Nial symbol table. The translation is done in two steps: first, the function is lexically analyzed into a sequence of symbol table numbers and then a top-down recursive descent parse take place, generating the APL code along the way. The target code consists of calls on APL functions which simulate the instructions of a simple stack machine. The pseudo-machine applies operations to arrays by either invoking the AT/370 interface or by recursive use of its own mechanism.

If the input is an expression or a statement then the tranlator is invoked to produce temporary APL code corresponding to the input. This code is immediately executed to achieve the side effect of the statement or to produce the value of the expression. In the latter case the value returned by the AT/370 routine is in a canonical linear encoding that is transformed by the Nial output routine into an array diagram. The code produced by the translator for an expression or statement, as for a definition, consists of calls on primitive instructions of the pseudo machine, which may in turn apply previously translated definitions.

The symbol table and associated tables retain information concerning the syntactic and semantic role of each symbol or token encounterd by Nial during a session. At the beginning of the session the tables are initialized to reflect a clear workspace with all the standard definitions in place. There are facilities to save and restore the workspace in a state that has additional definitions and variables.

As well as the supervisor routine that controls the interactive loop with the user, there are two other important global routines. One is a restart routine which sets up the connection to AT/370 and CMS, restores the symbol table to its clear workspace state and loads the AT/370 memory with initial values. A restart is automatically invoked when Nial is loaded and it may be called by the user through a system command. The second global routine is the startup facility which is available only in the master version of the Nial workspace and controls the system generation. This is the topic of the next section.

4. The startup facility.

A major design goal of the Nial system was to allow great flexibility in the development of the theory so that decisions concerning the use of names and symbols could be reconsidered with only minor disturbance to the working system. In addition, we wished to study the dependence among the definitions of operations and operators to be certain that we could specify the semantics of all of the theory starting with only a few primitive operations and our recursive definitional mechanism. Thus major effort went into finding a workspace organization that would facilitate such flexibility without undue cost at execution time.

The major concept in achieving this degree of flexibility was our realization that all tokens must potentially be able to take on any role in the language or system. Thus almost all decisions must be table driven, so that by changing table entries a token's role could be changed. We have not carried this philosophy to an extreme, however, but used it as a guiding principle in making design decisions.

The startup process has two major steps. First, the symbol table (or token table as we call it) is setup to reflect its minimal state and a number of variables that control syntax analysis, default command settings, and values of array denotations are initialized. The reserved words used in the linguistic aspects of Nial are also installed in this process.

The second step is to use a table, which we call the master list, to install, in the order they appear in the table, the constant arrays, and predefined operations and operators of the theory. The master list has fields giving the name of the object being defined, the kind of object it is (array, operation, or operator), its status, that is whether it is primitive, implemented, or formally defined, its symbol if any and its AT/370 code if any. If the staus is primitive or implemented then an AT/370 implementation is available. The symbol field indicates a graphic that can be used as an abbreviation for the name. For example "+" for "plus". A sample master list entry is

```
HITCH    D  I  ,  520
```

which indicates that hitch is an implemented dyadic operation with symbol and internal code 520.

The installation process is as follows. When the status field indicates that the formal definition is to be used, the definition is passed to the evaluator which translates it and sets up the symbol table pointer to the translation. If the status indicates an object defined in AT/370 then the symbol table entry is associated with AT/370 code. In either case if the symbol field is nonblank the symbol is installed by associating its symbol table entry with the same definition or AT/370 code. (A symbol can point to both a monadic and a dyadic operation and hence a level of indirection is required here.) The symbol table for the name is marked to indicate that the user may not redefine its use.

There are a number of utility routines associated with the startup facility that allow one to do a partial startup, to incrementally add definitions to those already in place, to gather definitions into a CMS file so they can be edited or listed, and to spread the definitions from a CMS file back into their corresponding APL variables. Other utilities available through the system command mechanism allow one to replace an AT/370 routine with its formal definition and optionally to trace the definition.

5. Uses of the development system.

The original purpose of the effort that resulted in the Nial system was to provide a convenient mechanism to generate test cases to validate the AT/370 routines. These were developed during a period when the theory was undergoing considerable change and hence the structure of the low level routines did not always match that of the current high level definitions. Moreover, by the use of special case testing, or less elegant but more efficient algorithms, many of the AT/370 routines are considerably faster than they would be if the formal definitions were directly translated. Thus, there is a need to ensure that the AT/370 code actually produces the expected results in all cases.

It is not too surprising to learn that many discrepancies have been discovered by comparing the output from the formal definitions with that from the implemented versions. Other inconsistancies were discovered by checking the validity of identities on a wide range of test cases. Of the discrepancies found, most were straight blunders of the kind typically made in low level programming, some were due to a misunderstanding of the semantics, mostly at the boundaries, and a few were due to trivial blunders in writing out the formal definitions. Thus the system has proved invaluable in ensuring that the implementation and the theory match correctly.

The testing program carried out to this point has been primarily hit or miss, since the manpower to carry out a systematic testing scheme and to maintain the AT/370 code has been somewhat limited. However, our experience indicates that by adopting a systematic testing strategy based on the Nial implementation the AT/370 implementation of the array operations could be validated to an extent far beyond that of most software systems of similar complexity.

Nial has been extensively used as a demonstration system to teach array theory concepts to people with no previous background in the theory. The operations can be explained by setting up simple examples which show how an operation transforms a given array diagram into the result diagram. This technique of teaching by pictures has proved to be very successful in one-to-one demonstrations at a CRT terminal and the output of such sessions is useful documentation of the theory. (See [Si80] for an example of the use of Nial in this manner.) Since the user can display the formal definitions at any time and may also trace their execution, the connection between the diagrams and the linguistic definitions are gradually assimilated.

Another major use of the system has been in exploring the theory itself. By having the defining sequence stored in a table that may be easily reorganized or that may exist in alternate versions, we have explored many different styles of presenting the theory. Trenchard More has concentrated on refining the development based on 8 primitives chosen because of their close association with set theory concepts. His development uses recursion in the definition of operators but all operation definitions avoid the use of explicit recursion. A preliminary version of More's defining sequence appears in [Si80] and a recent paper [Mo81] gives the current description. The refinements have been primarily aimed at choosing words that best describe the operations and at defining new operators that capture an underlying principle which can be utilized in the definition of several operations. The end result is that Nial, unlike APL, emphasizes words as well as symbols, and like functional programming [Ba78] uses operator-operation combinations to define new operations.

We have been using Nial to explore an alternate defining sequence for array theory based on six primitives, closely related to primitive operations used in APL and Lisp [Je80]. In my version, the master list also contains axioms and theorems in the form of predicates. During testing these are installed and tested incrementally, giving a preliminary validity check that the new defining sequence is correct. Since this approach is orthogonal in concept to More's development, depending heavily on recursion, it provides an additional testing mechanism to validate the AT/370 routines. All of this is made possible by the generality of the startup facility and the ability to use it incrementally.

Nial is more than an evaluator for array theory expressions. Its definitional mechanisms, along with the control structure facilites and assignment give it the semantic power of a full programming language. One of the goals of our work has been to explore how best to adapt the applicative expression language of array theory syntax into a programming language framework. We have used an imperative assignment syntax for associating names with values, yet in many contexts it may be interpreted as Landin's let construct, which is viewed as applicative [Te81].

In our explorations of trying to resolve imperative and applicative design goals we have twice revised the syntax and semantics of statement sequences as expressions. Major revisions in control structure design have also been made. This is accomplished quite simply; one modifies the grammar as necessary and then rewrites the parser routines to match the new syntax. Although it sounds like a lot of work, it can be done very quickly since most of the routines are just a few short lines of APL code and most revisions affect only a small number of routines. We diverged from the table driven approach in this area since we felt the recursive descent parsing mechanism gave far greater flexibility than any of the standard table driven methods.

Nial has also been used extensively in preparing documentation concerning array theory concepts. It is possible from within the system to capture in a CMS file the examples that are displayed on the terminal. The CMS files can be edited directly from Nial and commentary added to give a full explanation of the example. These facilities have been implemented by Neil Sorensen with John Shaw's assistance in making magic happen with the 6670 copier/printer. [Bo81],[Mo81] are examples of documents prepared using the system.

6. Internals of the Nial implementation.

In a paper of this length it is impossible to give a complete description of how Nial works internally; however, much of the methodology is a straightforward application of ideas from the literature on compiler writing. We sketch the basic approach with somewhat more detail where an unusual tack has been taken to take advantage of APL's strengths or to avoid its weaknesses.

Symbol table management

The symbol table contains a logical entry for each token encountered by the system during initialization or use. Each entry consists of a token, its syntactic role, its semantic role and its value. The syntactic role is used in parsing and is used to distinguish whether a token corresponds to a delimiter, an array, an operation , an operator, a reserved syntactic word, or is free to be used as a variable. The semantic role is used to enforce restrictions on the reuse of names in definitions. The value field is a pointer to an AT/370 array value or a pointer into the operation table.

The physical structure of the symbol table is designed to preserve space and yet achieve reasonablly rapid lookup. The design is due to Jean Michel [Mi76]. The four fields of the logical symbol table are stored as separate arrays. The token field is stored as a character vector with each token followed by an unused element of the atomic vector. Single character tokens are not stored; their position in the logical symbol table corresponds to their position in the atomic vector. The syntactic and semantic roles are stored as single characters in vectors of the length of the logical symbol table. The value fields are stored in an integer vector of the same length.

Rapid lookup is achieved by having a character vector of the same length as the vector of tokens which encodes in corresponding positions the length of the token in that part of the token vector. The encoding is that in the positions corresponding to a token of length n, the nth item of the atomic vector is used. To do a lookup of a multi-character token, the vector of tokens is compressed using a mask constructed from the length map and the separators, leaving only the separators and tokens of the given length. A standard fast string search can be used to find the position of the token in the abbreviated string. The position in the logical symbol table is computed from the latter value by adding the number of preceding separators plus the length of the atomic vector. Figure 3 gives the APL code that implements the lookup algorithm corresponding to the above scheme.

Relatively large symbol tables can be built in this manner with lookup times competitive to the usual character matrix approach. By using our compact token vector we are able to treat long string constants in exactly the same manner as all other constants without excessive use of space.

An input string is lexically analyzed into tokens using a standard state transition technique and converted to token indicies using the lookup mechanism.

```
⍝ S IS CHARACTER STRING TO BE SCANNED

LTKNS←((LENGTHS=⎕AV[100⌊⍴S)∨
     TOKENS=SEPARATOR)/TOKENS

→((⍴T)<POSITION←S STRINGSCAN LTKNS)/
     NEWTOKEN

INDEX←++/SEPARATOR=POSITION↑LTKNS

→0

NEWTOKEN:TOKENS←TOKENS,S,SEPARATOR

LENGTHS←LENGTHS,((⍴S)⍴⎕AV[100⌊⍴S]),'0'
```

Figure 3. The token lookup algorithm.

Parsing and code generation

The parser consists of a collection of routines corresponding to the nonterminals of the Nial grammar. Each routine has as its goal to find a match that fits one of the production rules for that nonterminal. Code is generated as the parse proceeds with each routine returning a success/fail flag and pointers to the generated code. To accomodate the full generality of array theory syntax, the parser has a backup capability so that if a trial production does not succeed an alternate rule can be tried.

The generated code is in the form of calls on APL routines that simulate a simple stack machine. Thus, the translation for the Nial expression

```
tell (I+1)
```

is equivalent to

```
push I
push 1
apply plus
apply tell
```

The operands of the instructions are actually in the form of symbol table pointers for "I" and "1" and in operation representations for "plus" and "tell". Other instructions include pop, assign, copy, duplicate, split and makelist. The control structure mechanism of Nial are translated to branches to generated labels in the APL code.

If the translation is of an operation or operator then prologue and epilogue code is added to manage the environment and set up parameter correspondences and the entire set of instructions is fixed in an APL function.

Representation of operations

In Nial all actions on arrays are performed by operations that are either part of the predefined set of operations, defined by the user, or derived from the former categories by composition or the application of operators. Predefined operators and operations may be either implemented in AT/370 or defined by standard precompiled definitions. The various kinds of operation forms are represented by numeric vectors that encode the kind of operation form, the adicity of the operation, and the pointer(s) to either the AT/370 code(s) or the APL translation(s).

The apply pseudo-instruction uses the first field of the operation representation to determine what action to take in applying an operation. If it is a defined operation apply invokes the APL function that contains the translation of the definition. This is a parameterless function that receives the arguments to the defined function from the stack and leaves its result there. If it is an operation implemented in AT/370 the shared variable interface is triggered with the appropriate arguments from the stack and the result is pushed back onto the stack.

A defined operator is implemented by the same mechanism used for operations except that the resulting APL function has parameters. The parameters are used to pass the operations to the operator's body as operation representations. Operator representations are compiled out during translation resulting in operation representations that have references to the appropriate operator. This is possible because the semantics of Nial force the role of all tokens in a Nial expression to be known at the time the expression is parsed.

The array diagrams

The ease with which Nial users grasp basic array theory concepts is due in part to the use of diagrams to give a clear indication of the data structures being manipulated. There is a close relationship between the diagrams, the use of strand notation, and the semantics of the replacement transforms EACH and LEAF which reinforces the item-of/array relationship that is central to the theory. The basic concepts of the diagrams go back to "ham and eggs" pictures More used in lecture notes in discrete mathematics [Mo60] and the specific windowpane format is evident in his early work at Yorktown [Mo73]. The diagrams used in [Gh73] and those in [Gu79] are different from each other and More's diagrams. Having worked with all three, we are convinced the the windowpane diagrams are the most effective in explaining nested array concepts. The present diagrams reflect some minor refinements that have evolved after some experience in using Nial.

An array diagram is a picture in a plane of a multidimensional nested array. The nesting is flattened by recursively displaying the pictures of the items in the panes of a window frame which illustrates the top level structure of an array. The dimensionality is displayed by the positioning of the frame (or frames for arrays with more than 2 dimensions) on the plane. An example is

```
                        2 2 3
 -----------    -----------
|   |   |   |  |   |   |   |
 --- --- ---    --- --- ---
|   |   |   |  |   |   |   |
 -----------    -----------
```

which is the diagram for an array of shape 2 2 3 with diagrams of the items omitted. For a further discussion of array diagrams and their meaning see [Mo81].

The array diagrams are constructed as large character matrices by the Nial output routine. It uses a pair of recursive routines based on the following algorithms.

Diagram algorithm.

1. If the array is a mote return the diagram for that mote.
2. If the array is empty find the diagram of the one-list containing its prototype as the item and mark the diagram with the empty mark.
3. If the array is a single find the diagram of the one-list with the same item and mark the diagram with the single mark.
4. Otherwise,
   - 4.1 Find the diagrams of all the items of the array.
   - 4.2 Use the paste algorithm to build the windowpane diagram with boxing and minimal spacing.

Paste algorithm.

1. If the valence is greater than 2 then
   - 1.1 Use the paste algorithm with no boxing to combine the diagrams of items for each of the subarrays of the next lower even valence. The spacing factor is determined by the valence of the subarrays.
   - 1.2 Combine these using the paste algorithm with increased spacing and no boxing.
2. If the valence is less than 2 then
   - 2.1 Compute the size needed for each row and column using the algorithm suggested in [Je78] for format.
   - 2.2 Loop over row and columns adding items with suitable overtaking to adjust space to the sizes computed in step 2.1 and allow for the spacing factor.
   - 2.3 If the boxing flag is set, expand the diagram between rows and columns and insert the graphical characters that create the window frame.

The implementation of the diagram algorithm is complicated by the fact that we are determining the array value by parsing the canonical representation simultaneously with building its picture. Moreover, we attempt to utilize APL format in several simple cases. The paste algorithm requires four parameters: the shape, the list diagrams of the items, the flag to indicate boxing, and a spacing factor. In APL the list of diagrams is represented by a character matrix of appended diagrams with a numeric matrix giving the shape of the corresponding diagram as supplementary information. The numeric data is passed as actual parameters and the character matrix passed in a free variable. This requires some saving and restoring of parameters in local variables in order to get the recursion in the paste algorithm to come out correctly. It is interesting to note that the paste algorithm is considerably easier to express in Nial itself than in APL because of the generality of arguments that can be passed recursively.

7. Concluding remarks.

We have presented an example of the use of APL in exploring programming language design. Our purpose has been two-fold. First, to illustrate that APL is well suited to building prototype implementations of interpreted languages and that the resulting system can serve a variety of purposes. A second motivation was to give the APL community an update on the progress in array theory since its advancement may affect future directions for APL.

Acknowledgments

The work reported here was done in close collaboration with Trenchard More and many of the details of the Nial system were designed to meet his expectations. I would also like to thank Tony Hassitt, Len Lyon, Neil Sorensen, and John Shaw for their direct help in putting Nial together, Willard Bouricius, whose early implementation led the way and Jean Michel, whose pragmatic ideas on building interpreters in APL are reflected in the Nial design.

8. References.

[Ba78] J. Backus, Can programming be liberated from the von Neumann style? Comm. ACM 21, 8 613-641.

[Bo81] W. G. Bouricius, N. R. Sorensen, An informal introduction to array theory with applications to a language and a data base. to appear in Structures and Operations in Engineering and Management Science. O. Bjorke and O. I. Franksen, Editors. Tapir Publishers, Trondheim, Norway. Spring 81.

[Gh73] Z. Ghandour, J. Mezei, Generalized arrays, operators and functions. IBM J. Res. Devel. 17, 4 335-352.

[Gu79] W. Gull, M. A. Jenkins, Recursive data structures in APL. Comm. ACM, 22, 2 79-96.

[Ha79] A. Hassitt, L. Lyon, Array theory in an APL environment. Proceedings APL79, Rochester, New York. 110-115.

[Ha81] A. Hassitt, L. Lyon, A description of AT370, in preparation.

[Je78] M. A. Jenkins, J. Michel, Oerators in an APL containing nested arrays. APL Quote Quad, 9, 2 8-20.

[Je81] M. A. Jenkins, The Nial reference manual. In preparation.

[Mi76] J. Michel, private communication.

[Mo60] T. More, Class notes on the algebraic foundations of switching circuit theory developed at MIT.

[Mo73] T. More, Notes on the development of a theory of arrays. Rep 320-3016, IBM Scientific Center, Philadelphia Pa.

[Mo75] T. More, A theory of arrays with applications to data bases. Rep G320-2016, IBM Scientific Center, Cambridge, Mass.

[Mo79] T. More, The nested rectangular array as a model of data. Proceedings APL79, Rochester, New York. 55-73.

[Mo81] T. More, Notes on the diagrams, logic and operations of array theory. to appear op.cit. [Bo81]

[Si80] S. M. Singleton, An investigation of More's array theory. Tech Rep. 80-99, Computing and Information Science, Queen's U. Kingston, Canada.