Nial: A Candidate Language for Fifth Generation Computer Systems

C. D. McCrosky, J. J. Glasgow, and M. A. Jenkins

Department of Computing Science
Queen's University
Kingston, Canada, K7L 3N6

1. Introduction

The anticipated fifth generation of computing systems presents many challenges. One of the more important is the challenge of designing languages suitable for describing the parallel computations which these systems will achieve. The new systems will outpace the expressive power of most existing languages. As the hardware components and AI techniques are developed to achieve the fifth generation, so must appropriate languages be created.

Another view of the fifth generation is that it is to be distinguished from previous systems, in large part, by being language driven. That is, the abstraction of some functional language is to be the driving force behind the machine design, and that potential parallelism in the language should motivate parallelism in the machine.

Certainly the role of language in fifth generation systems will be vital. We propose a candidate language for these upcomming systems. The language is the Nested Interactive Array Language (Nial). We show that Nial is highly suitable for fifth generation systems. The current state and future directions of Nial language design, Nial architectures, and logic programming in Nial are examined.

2. Nature of Fifth Generation Systems

We view fifth generation systems as consisting of four components: hardware, languages, artificial intelligence software, and encoded knowledge. The hardware is briefly discussed below to set the stage for the language requirements. We assume that logic programming techniques will play an important but not totally dominating role for the AI software component. The nature of encoded knowledge is not discussed. The rest of the paper focuses on the language component, and its relationship to the hardware and logic programming.

2.1. Hardware Components

The structure of the new hardware is an open question, but given the initial statements by the Japanese and DARPA [Conway84, Uttal82], it seems that it will have the following characteristics:

1) Very fast devices: A large gain in processing speed will come from the use of VLSI devices with smaller feature sizes and higher degrees of integration. This aspect of the new processors is of the least interest in terms of its relationship to the software and language components. To the extent that the required speedup can come from faster devices, language definitions can remain unchanged.

2) Massive Parallelism: Even much faster devices will not be adequate to achieve the desired speedup. A solution is to devise architectures which allow massive parallelism, thus multiplying the speed gain of the basic devices. This innovation is the most challenging aspect of the fifth generation for architecture and language definition.

3) Logic Programming Support: Fifth generation hardware will certainly include primitive operations to support logical inference. The drawing of logical inferences is expected to be the major burden for these systems, so naturally, machines will implement the operations as efficiently as possible. Implementation in hardware or microcode seems most likely.

4) Language-Oriented Architecture: A primary objective is that an abstract language be supported. It seems that direct architectural support for the language offers benefits in code density and speed [Flynn83]. In systems where memory is distributed and slow, access to code is a major expense, and language-oriented architectures can reduce this

cost.

2.2. Fifth Generation Languages

The languages to be used on these new systems will have two fundamental requirements:

1) The language must capture the ideas of logic programming and/or other artificial intelligence techniques.

2) The language must take advantage of the massive parallelism of the underlying hardware.

Further requirements for the language come from concerns in software engineering.

3) The language should express very high level concepts concisely. It should be easy and natural to construct high level ideas using powerful, integrated primitive operations and a rich set of construction mechanisms.

4) The language should be mathematically tractable, allowing correctness proofs and automated optimization transformations.

These requirements do not strictly constrain the language design - there is much room for creative response. The language which we propose does satisfy these demands, but the unique aspect of Nial is that it implies a very direct relationship between the forms of the language and the parallel structure of the underlying machine. We will show that Nial can lead us to a particular parallel architecture which may well be realizable with extensions of current approaches.

3. Introduction to Nial

Nial is a programming language built on the theory of nested arrays developed by More [More79, More81]. More's theory includes three sorts of entities, arrays which are the data, operations which map arrays to arrays, and transformers which map operations to operations. It is useful to introduce the ideas of the theory by progressing from arrays to operations to transformers. We give only examples for sake of brevity. A development of the theory and language can be found in [Jenkins83, More81].

3.1. Arrays

Arrays consist of atomic arrays - such as the integers, booleans, reals, characters, and symbolic phrases - and rectangular non-atomic arrays which are constructed from both atomic arrays and other non-atomic arrays. All arrays are characterized by a small set of properties such as their valence (number of dimensions), shape (extent on each dimension), and tally (number of contained items).

Examples of atomic and non-atomic arrays are presented below. The examples correspond to the results of actually using the Nial implementation [Jenkins83]. User input is displayed offset to the right, the interpreter's response is displayed on the left margin, with minor editing to aid readability. Arrays are presented in pictures which give the structure and contents in a concise graphic way. The boxes are a canonical way to represent nested rectangular arrays [More73, Jenkins81].

An atomic array:

```
     1
  1
```

A list containing three atoms (phrase, fault and real):

```
     "phrase ??fault 10.94

  +------+------+-----+
  |phrase|?fault|10.94|
  +------+------+-----+
```

The above picture is given in the full diagram mode. A reduced picture is possible for lists in sketch mode. The above list would be shown as:

```
     "phrase ??fault 10.94
  phrase ?fault 10.94
```

A table containing the first six letters of the alphabet:

```
     2 3 reshape 'abcdef'
abc
def
```

A nested array containing an atom and a pair:

```
     1 (2.7j2.1 truth)

  +-+-----------+
  |1|2.7j2.1 1  |
  +-+-----------+
```

Arrays can be written in two syntaxes: strand notation and general array notation. Corresponding examples are given below. The general array notation is a complete notation, while strand notation can only express a subset of arrays. Strand notation is simpler to write, and often more convenient.

Strand	General Array
1 2	[1, 2]
1 (2 3)	[1, [2, 3]]

3.2. Operations

The operations of Nial map arrays to arrays. They fall into two general classes, those which deal with shape and structure only, and those which manipulate the atoms. All operations are total in that they map from all arrays to arrays, though they may result in a fault atomic datatype. Structure operations include the following examples:

```
     shape "a "b "c
  3

     2 2 reshape 'abcd'
  ab
  cd

     tally (2 2 reshape 'abcd')
  4

     cols (2 2 reshape 'abcd')

  |ac|bd|
```

There are a variety of ways to form lists:

```
     "tag hitch "x "y
  tag x y

     1 2 append 3
  1 2 3

     'hi' link 'jk'
  hijk
```

Operations which touch the values of atoms include the arithmetics. They are quite powerful operations:

```
     sum 1 2 3
  6

     sum 2 (1 2 3)
  3 4 5

     sum (1 2)(1 2)(1 2)
  3 6

     sum (1 2 (3 4) 5)  2  (5 4 (3 2) 1)

  |8|8|8 8|8|
```

The same patterns of application hold for the boolean operations ("l" denotes truth, "o" denotes falsehood, and a list of truth values may be denoted as a string composed of "o" and "l").

```
     and lol l ool
  ool
```

The operations of Nial can be gathered into atlases [More82] - similar to the forms used by Backus [Backus78] - which are part of the the general array notation of Nial. Each operation in an atlas is applied to the array argument, and the result of the atlas application is the array formed by all the individual function applications. The shape of the result is the same as the shape of the atlas. For example:

```
     [sum, tally] [2, 6, 4]
  12 3
```

3.3. Transformers

The transformers of Nial map operations to operations. The presence of transformers is one of the most important sources of expressive power in the language. The fact that higher order functions are limited to only the second order is an important simplification which eases implementation and allows a very powerful syntax. By convention, transformer names are written in uppercase.

The most common transformer is EACH. EACH applied to an operation, f, maps to an operation which applies f to each item of the argument array. Thus:

```
     EACH f (A B) = (f A) (f B).
```

EACH is considered a primitive transformer. There are numerous other primitive transformers, and a definitional mechanism to create arbitrary user defined transformers.

Examples of some common transformers include:

```
     EACH shape ("one (1 2)) ("two ?L ?A)
  2 3

     EACHRIGHT in `a ('abc' 'def' 'aaa')
  lol
```

FORK is a transformer which expects an atlas of three operations. The first operation is applied to the argument. If it evaluates to truth, then the application of the second operation to the argument is returned, otherwise the application of the third operation to the argument is returned.

```
     FORK [in, second, link]
        "bill ("sam "bill "david)
  sam bill david

     FORK [in, second, link]
        "joe ("sam "bill "david)
  joe sam bill david
```

LEAF is a transformer which applies its operation to the atoms (leaves) of its array argument. FIX is a transformer which computes the fixed point of an operation on an array. Transformers may also be gathered in bracket notation like general arrays and atlases.

3.4. Definitional Syntax

Nial has a small, powerful set of definitional mechanisms for expressions, operations, and transformers. All the definitions are based upon the IS form, in which a symbol is defined to be an expression, operation, or transformer. Operations and transformers can be defined with a type of lambda-form, or in a lambda-free form.

```
     second IS first rest

     EACH_PIECE is EACH EACH


     average is divide [sum, tally]

     average 6 2 4
  4.


     union is operation A B (
       if A allin B
       then B
       else cull link A B
       endif)
```

Two unique aspects of Nial provide much of the definitional power and simplicity. There is an equivalence of infix and prefix forms:

```
     f A B  =  (A f) B  =  A f B
```

and all of the possible juxtapositions of the three sorts of entities (arrays, operations, and transformers) have meanings. Results of these rules include the facts that "equal EACH" is a transformer and "1 +" is an operation.

```
     MONO is equal EACH

     MONO shape (1 2) (3 4)
  truth

     incr is 1 +

     incr 0
  1

     incr 0 1 2
  1 2 3
```

EACHCART is a simple example of a user defined transformer. It applies its operation to each item of the cartesian product of its argument.

```
     EACHCART is TR f (EACH f cart)

     EACHCART > (0 1 2 3) (0 1 2 3)
oooo
looo
lloo
lllo
```

4. Nial Captures Parallelism

We have claimed that languages for fifth generation systems must capture the notion of parallelism. Nial has the family of EACH transformers which encapsulate something of the notion of parallelism. When evaluating EACH f A, all the applications of f to the items of A can proceed in parallel.

However, Nial readily expresses a much broader range of parallel concepts. A measure of the ability of Nial to express parallelism can be had by comparing Nial expressions with Flynn´s classic categorization of computer architectures [Flynn66]. Flynn classifies machines as to whether they have one or more instruction flows and whether they have one or more data flows. The cartesian product of these attributes gives Flynn´s categories. Nial readily expresses all of Flynn´s machines, as functional applications.

SISD: Single Instruction and Data Streams

```
       f A
```

SIMD: Single Instruction Stream, Multiple Data Streams

```
       EACH f (a b)  =  [f a, f b]
```

MISD: Multiple Instruction Streams, Single Data Stream

```
       [f, g] A  =  [f A, g A]
```

MIMD: Multiple Instruction and Data Streams

```
       TEAM [f, g] [a, b]  =  [f a, g b]
```

Further, Nial can easily express more complex forms such as applying an operation to each item of the cartesian product of an array, or applying an operation to each atomic value found in a nested array. Such additional power derives from of the ability to define transformers, and is quite unlimited.

Clearly, the Nial language expresses parallelism in a very powerful and concise manner. This makes it a serious candidate for massively parallel machines. Nial´s view of parallelism also implies some aspects of the underlying architecture.

It will be useful to examine a possible supporting architecture.

5. A Parallel Nial Architecture

In this section we present our current ideas for an architecture which would be appropriate as a massively parallel Nial engine. The design presented has a course-grained parallelism. It is certainly not the only suitable architecture, but is of interest because of the very close correspondence of the language and the architecture.

On reflection, it is seen that the correspondence between Flynn´s machines and Nial expressions given above is appropriate only for individual applications of operations or transformed operations to arrays. Each Nial application specifies a process splitting strategy. SISD implies no splitting, SIMD implies one process for each item of the array, MISD implies one process for each operation of the atlas, and MIMD implies one process for each operation-item pairing. Each sub-process created can contain applications of any of these four types, and hence can generate more parallelism. Thus as stated, Flynn´s classifications correspond to each function application, and a rapidly branching tree of parallelism is possible.

How can an architecture cope with this sort of parallelism? One possibility would be to have each process creation wait until sufficient processors are available to handle the degree of parallelism. But this is unsatisfactory because it would be very difficult to gather processors for the many tasks waiting to split, and processors would be idle while waiting for enough to fulfill a task split.

It seems clear that each process splitting should create descriptors describing what tasks need to be run. The descriptors need contain only a description of the operation to be applied, the array to be used as an argument, and where the result should be placed.

Now consider a machine in which there are a large number of processors, a global memory system, and a network connecting each processor to the global memory. (The processors may well have local memories for temporaries and caching). Examples of similar architectures are the Ultracomputer of NYU [Schwartz80, Gottlieb83] and the Butterfly, designed and implemented by Bolt Beranek and Newman in Cambridge, Mass. [BBN83].

Add to this architecture a set of task storage nodes which are capable of holding the task descriptors split off by the processors, and a second network which connects each processor to a task-node and all the task-nodes together. (The degree of connectivity of both the memory and task networks are difficult questions. We do not address these issues here, except to suggest that the connectivity of the task network could be lower than the connectivity of the memory network. Further, the task-nodes and network could share the hardware of the processor-nodes and memory network - they could just be separate activities.)

Now, each time a processor finds an application other than a SISD, it creates an array container for the result, generates a set of task descriptors, and puts them in a task-node. If the processor finds a SISD application, it does the work itself. When a processor is idle, it appeals to a task-node for a task descriptor, and undertakes that work.

When an application has resulted in a creation of task descriptors, the task it represents must wait. The waiting and synchronization required can be managed by keeping a count of incomplete subtasks in the result container which was created for the task split. When the last task completes, the processor doing that work can signal the completion of the now complete mother task.

The injection of tasks into the task-nodes is not predictable, but a complex, dynamically determined process. It may happen that tasks would concentrate in one task-node, while other task-nodes are empty and their connected processors idle. Consequently, it is necessary to have the task-nodes redistribute task descriptors over the task network to keep the task load balanced. The task-nodes would have a simple protocol with which each connected pair would decide whether they needed to shift tasks. This process could be made more intelligent in a system where processors were "closer" to certain parts of memory. The tasks could be rebalanced in such a way that task-nodes accumulated tasks which needed memory close to their connected processors.

As it would be possible for this system to generate too many tasks, flooding the task-node memory, the task-nodes would watch for a too-many-tasks condition. When that condition occurred, they would give priority to SISD tasks, thus reducing congestion. Likewise, when a too-few-tasks condition is detected, non-SISD tasks would be given priority.

A last point is that it would not be worthwhile splitting some of the SIMD, MISD, and MIMD applications if they are too trivial. For instance it would be foolish to split [first, second] [a,b]! It might be possible to develop effective heuristics in the processors to decide whether a process should be split. Some measure of the difficulty of the subtasks, coupled with the count of tasks could be the basis for this heuristic.

6. Nial Helps Smart Memory Access

Access to memory is the most difficult problem in massively parallel systems. Many processors require access to common programs and data, yet access to global memory is limited. Much research has addressed this difficulty, but the issue remains problematic [Hwang84].

Nial has aspects which may significantly relieve this problem. Assume that a Nial architecture has some local memory with each processor, and some shared memory. Clearly, if a processor needs to use an array from global memory more than once, it is advantageous to move a copy of that array to the local memory. Yet how, in conventional languages and architectures, is it possible to know when this caching technique will pay? Conventional systems with caches usually copy to the cache regardless, but this may displace important cache contents to no advantage.

Nial executions can know that an array will be used more than once. Consider the SIMD and MISD applications. In the first case, it is known that the program will be used as many times as there are items in the array, and in the second case, it is known the data will be used as many times as there are operations in the atlas. Thus if we give all (or some minimum number) of such applications to one processor, then the cache can be used effectively. In the cases where it is known that an array will be used only once (e.g. the data items in a SIMD) it is possible to avoid disturbing the cache. It should be possible to design a memory system to take advantage of this knowledge.

7. Nial and Logic Programming

The techinques of logic programming may play a dominating role in fifth generation systems. It is certainly necessary for these techniques to be supported. There is a temptation to design a fifth generation Prolog machine - as the Japanese are doing [Uttal82]. Although we agree with the motivation of building a machine which directly supports reasonably abstract ideas, we reject the Prolog-machine course: 1) it will probably be necessary to use more traditional AI techniques on the machine as well, and 2) Prolog, as an instantiation of logic programming, is too rigid.

Nial has been examined as an implementation basis for the traditional AI techniques such as frames and semantic nets. It appears that Nial has no difficulty in expressing these techniques [Glasgow84b]. In this section, we will examine the issue of a more flexible implementation of the essential ideas behind Prolog.

Prolog combines two distinct ideas. The first is resolution logic, the second is a strategy for searching a space of possible solutions. [Kowalski79]. The reader is assumed to be familiar with the basic principles of resolution logic as used in Prolog.

The method of control used in Prolog is a form of resolution referred to as top-down interpretation of Horn clauses. In trying to prove that an instance of the goal statement is true, the Prolog system searches for the first clause whose consequence is unifiable with the goal, and continues to exhaustively search that subtree before trying subsequent unifiable clauses. This strategy is a depth-first tree search. The weaknesses of this rigid approach have been widely recognized in the AI community. It is possible to follow a non-terminating path down the tree, and if the strategy does succeed, it may be very wasteful of processing resources compared to "smarter" heuristic search strategies [Mackworth75].

It is not necessary to tie the ideas of resolution logic to a particular search strategy [Glasgow84a, Robinson82]. We propose, and have experimented with, replacing the Prolog strategy with the and/or tree formalism coupled with programmer specified heuristics. This work is done in Nial because the nested arrays of Nial are a very powerful data structure and easily capture the data necessary for resolution logic. Also, the high-level nature of Nial makes the programs easy to write and understand.

7.1. And-Or Trees

A common method for solving a goal is to reduce it to a series of subgoals to be solved. We may need to solve either a conjunction (solving all) or a disjunction (solving one) of the subgoals. An and-or tree is a tree composed of alternating layers of such conjunctions and disjunctions. A goal is solved by successively replacing goals by subgoals until only solved subgoals are found, and the initial goal is solved.

Prolog´s control strategy can be viewed as a traversal of an and-or tree with a pre-selected path. The disjunction levels correspond to the choice of a unifiable clause, and the conjunction levels correspond to the conditions of one clause. Prolog always takes the disjunction alternatives in a given order, and pursues the selected one to a conclusion.

We introduce heuristics to determine which alternative of a disjunction to choose, and to determine how deeply to pursue any search. With the ability to define heuristics, arbitrary search strategies may be used, from depth-first to breadth first [Barr81].

By implementing resolution logic in Nial´s arrays, and being free to choose appropriate representations for the Horn clauses, we leave ourselves free to include heuristic functions in the representations. These functions are to be used by the general search routines to determine if a given path should be pursued.

It is desirable to associate arbitrary heuristic functions with each clause of a sentence - thus the representation of clauses should include a way to find the associated operation. Nial has a natural method to support this requirement in its "casts". All Nial programs are represented as arrays. Using the linguistic operations provided, the expression:

```
parse scan ´f 1 2 3´
```

computes a valid array, which is an executable program representation. There is a notation in the language called a "cast" which captures this notion. Thus the array:

```
!(f 1 2 3)
```

is a notation for the array computed above. Casts can be taken of expressions, operations, or transformer forms. They can be freely included in other arrays, such as Horn clause representations. Then the resolution logic in Nial program can apply the specified operation as the heuristic mechanism. For example:

```
HornClause is Goal !(Heuristic)...

eval (second HornClause)
```

or:

```
apply (second HornClause) ...
```

The basis of these ideas have been implemented in Nial [Glasgow84a]. Much experimentation remains to be done with heuristic functions, both as to what they should be and when they should be applied. Although the current execution times are excessive, we are confident that a suitable implementation can achieve good results. A Nial engine (parallel or not) will be able to implement this concise statement of resolution logic in microcode or hardware, thus greatly accelerating the logic operations. Further, the Nial implementation of these ideas is highly parallel (unifications are formed and conjunctions are evaluated using EACH), making it a very suitable algorithm for a parallel Nial engine.

We have implemented resolution logic ideas in Nial in a significantly different way than others have put them in Lisp [Komorowski82, Robinson82]. The Lisp-based logic systems are imbedded languages - the logic ideas are presented as a separate language, which may use the underlying functional Lisp to varying degrees. The Nial-based approach we use does not create an imbedded language, but extends the functional environment to include operations for logic programming. Thus, in the Nial approach, there is no difficulty in using other aspects of the functional environment, and the implementation of logic ideas can easily be modified in the underlying array-theoretic model to achieve flexibility.

8. Mathematical Properties of Nial

Nial is based upon a formal theory of arrays. This fact gives (non-imperative) Nial programs a strong formal basis, and allows several substantial advantages to Nial users.

The mathematical tractability of Nial is best illustrated by presenting one of the program equivalences which are derived from More´s theory:

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

This result is useful in deriving the theory of Nial, and has the practical consequence of allowing important optimizations. In a sequential processor, the two EACH form of the expression is equivalent to two successive loops, with a temporary array used to communicate between the loops. The one EACH form is an optimization akin to code motion of optimizing compilers - it results in only one loop, and avoids building an extra temporary [Aho79].

The combining of EACH´s also has advantages for a parallel Nial engine such as the one presented above. The two EACH form requires two splittings and two synchronizations, and these four actions must be serial. On the other hand, the one EACH form requires only one splitting and one synchronization. Thus less work is required, and the work removed is serial.

The unusual feature of Nial is that such equations hold for all operations and all arrays, without exception. They hold for arbitrarily shaped arrays, and even empty arrays. Thus such program transformations may be applied with complete confidence, and without analysis.

9. Relationship to Previous Languages.

Although Nial has obvious roots in existing languages (most notably APL, LISP, and FP), the combination of features is unique, and seem uniquely applicable to fifth generation computing systems.

Nial shares general, shaped arrays of atoms with APL. But Nial extends the idea to nested arrays. New APL´s are including nested arrays, but only as a partially integrated feature [APL2].

APL has higher order functions, limited to the second order, integrated with the syntax (e.g., reduce), but unlike Nial, APL does not permit user definition of second order functions, nor does APL have such a rich set of given transformers [APL].

LISP has deeply nested data structures, but only the cons cell pairs are fully general. The arrays of LISP are restricted and poorly integrated with the theory of s-expressions [McCarthy65, Foderaro80, Weinreb81]. The nested arrays of NIAL are well defined, and exhibit no awkward discontinuities.

LISP allows explicit control of sharing of data objects, and sideeffects through ownership pointers. These undisciplined features make generalization to massively parallel architectures difficult.

LISP is based on the untyped lambda calculus, and thus has arbitrary power for passing and returning functions as arguments and values of functions. NIAL supports only transformers which take operations as arguments and return operations. In practice, however, LISP provides only "mapcar" and the other "maps", while NIAL has both a rich set of primitive transformers and mechanisms to create new transformer definitions. The advantages of NIAL´s treatment of second order functions are: 1) the more restricted definition makes transformers easier to conceptualize and hence more used, and 2) the more general data structures of NIAL provide the ground for powerful analogies between data structure and transformer (e.g., arrays as sets relate to EACH, and arrays as trees relate to LEAF and TWIG).

FP is the source of NIAL´s atlas notation, but NIAL generalizes this notation to arrays and transformers as well. The data structures of array theory are much richer than those of FP. FP and array theory share the property of mathematical tractability [Backus78].

10. Conclusions

Nial has many features which make it a serious candidate for fifth generation systems. In summary, these are:

1) It captures a very powerful notion of parallelism. Programmers using Nial can express parallel computations without explicit specification.

2) It allows a natural and useful implementation of resolution logic. Yet it does not commit the fifth generation system totally to the logic approach, more general purpose programming could be undertaken with the same resources.

3) It is a very high level language.

4) It is a mathematically tractable language, allowing important optimizations on programs and perhaps correctness proofs.

These aspects of Nial make it a serious contender for use in fifth generation systems. The relationship between the forms of Nial and our proposed parallel architecture indicates how advantageous it can be to consider both language and underlying machine simultaneously.

11. Future Work

Nial is implemented by a conventional, single-process interpreter written in C, and running on half a dozen machine and operating system combinations (e.g., VAX/UNIX, VAX/VMS, IBM PC/DOS, MC68000/XENIX, PDP 11/UNIX, DEC 20/TWENIX, and NS16032/UNITY). It is in use for research and teaching at a dozen institutions in North America and Europe.

Several lines of research are underway:

1) Work on the underlying theory of arrays and the design of the language Nial is continuing.

2) The current portable interpreter is being improved and ported.

3) A single-process VLSI-compatible NIAL architecture is being developed. Later we expect to produce an operational implementation.

4) A parallel architecture akin to the one described here is being investigated.

5) Further development of logic programming in Nial is underway. In particular, new array representations for and/or trees and resolution logic forms are being investigated, and further work on appropriate heuristics is being done.

6) Array theory based models for vision and natural language understanding are being implemented in Nial.

We feel that the current state of the Nial language and research offer excellent prospects for arriving at a language which is suitable for the upcoming generation of machines, and can serve both for logic programming and more general applications.

12. References

[Aho, A. V., and Ullman, J. D., Principles of Compiler Design, Addison-Wesley, 1979, pp. 410-415.

[APL]
Falkoff, A. D. and Iverson, K. E., APL/360 User's Manual, Thomas J. Watson Research Center, IBM Corp., Yorktown, N.Y., July, 1968.

[APL2]
APL2 Language Manual, IBM Corp., SB21-3015-0, (June, 1982).

[Backus78]
Backus, "Can Programming be Liberated from the von Neuman Style? A Functional Style and Its Algebra of Programs," Comm. ACM 21,8 (Aug. 1978), pp. 613-641.

[Barr81]
Barr, A., and Feigenbaum, E., The Handbook of Artifical Intelligence, vol 1, HeurisTech Press, Stanford, CA, 1981, pp. 74-83.

[BBN83]
Bolt Beranek and Newman, presentation given to the AI Lab, M.I.T., Dec., 1983.

[Conway84]
Conway, L., "Strategic Computing: A Strategic Plan for the Development of Machine Intelligence Technology and Its Application to Critical Problems in Defense," DARPA, Conference on Advanced Research in VLSI, M.I.T., Penfield, ed., Jan., 1984.

[Flynn66]
Flynn, M. J., "Very High-Speed Computing Systems," Proceedings of the IEEE, 54,12 (Dec., 1966), pp. 1901-1909.

[Flynn80]
Flynn, M. J., "Directions and Issues in Architecture and Language," IEEE Computer, October 1980, pp. 5-22.

[Flynn83]
Flynn, M. J., "Execution Architecture: The DELtran Experiment," IEEE Computer 32,2 (Feb, 1983), pp. 156-175.

[Foderaro, J. K., and Sklower, K. L., The Franz Lisp Manual, University of California, Berkeley, CA., 1980.

[Glasgow84a]
Glasgow, J. J., "Logic Programming in Nial," in preparation.

[Glasgow84b]
Glasgow, J. J., and Browse, R., "Programming Languages for Artificial Intelligence," to appear in the International Journal of Computers and Mathematics.

[Gottlieb83]
Gottlieb, A., Lubachevsky, B.D., and Rudolph, L., "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," TOPLAS 5,2 (April, 1983), pp. 164-189.

[Hwang84]
Hwang, K., and Briggs, F., Computer Architecture and Parallel Processing, McGraw-Hill, 1984.

[Jenkins81]
Jenkins, M. A. and Schmidt, F., "Array Diagrams and the Nial Approach," Tech. Report No. 81-131, Queen's University, Kingston, Canada, Nov. 1981.

[Jenkins83]
Jenkins, M. A., Q,Nial Reference Manual, Queen's University, Kingston, Canada (Dec., 1983).

[Jenkins84a]
Jenkins, M. A., "A Recursive Development of Array Theory," in preparation.

[Jenkins84b]
Jenkins, M. A., "The Role of Equations in Nial," in preparation.

[Komorowski82]
Komorowski, H. J., "QLOG - The Programming Environment for Prolog in Lisp," Logic Programming, Clark and Tarnland (eds), Academic Press, 1982.

[Kowalski79]
Kowalski, R. A., "Algorithm = Logic + Control", CACM, August, 1979.

[Mackworth75]
Mackworth, A. K., "Consistency in Networks of Relations," Technical Report 75-3 (July, 1975), Dept. of Comp. Sci., University of British Columbia.

[McCarthy65]
McCarthy, et. al., Lisp 1.5 Programmer's Manual, MIT Press, Cambridge, MA, 1965.

[More73]
More, T., "Axioms and Theorems for a Theory of Arrays," IBM Journal of Research and Development, 17,2 (March 1973), pp. 135-175.

[More79]
More, T., "The Nested Rectangular Array as a Model of Data," APL79 Conference Proceedings (May, 1979), pp. 55-73.

[More81]
More, T., "Notes on the Diagrams, Logic and Operations of Array Theory," Oyvind Bjorke and Ole I. Franksen (eds.), "Structures and Operations in

Engineering and Management Systems," Tapir Publishers, Trondheim, Norway, 1981, pp. 497-666.

[More82]
More, T., private communication to Jenkins and McCrosky, August, 1982.

[Robinson82]
Robinson, J. A. and Sibert, E. E., "LogLisp: Motivation, Design and Implementation," Logic Programming, Academic Press, 1982.

[Schwartz80]
Schwartz, J. T., "Ultracomputers," TOPLAS 2,4 (Oct., 1980), pp. 484-521.

[Uttal82]
Uttal, B., "Here Comes Computer Inc.," FORTUNE (Oct., 1982), pp. 82-90.

[Weinreb80]
Weinreb, D., and Moon, D., Lisp Machine Manual, MIT, Cambridge, MA., 1980.

CR Categories and Subject Descriptors: I.2.5 [Artificial Intelligence]: Programming Languages and Software; C.1.2 [Processor Architectures]: Multiple Data Stream Architectures -- multiple-instruction-stream, multiple-data-stream processors (MIMD); D.3.m [Programming Languages]: Miscellaneous