Comp. & Maths. with Appls. Vol. 11, No. 5, pp. 431–448, 1985
Printed in Great Britain.

0097–4943/85 $3.00 + .00
© 1985 Pergamon Press Ltd.

PROGRAMMING LANGUAGES FOR ARTIFICIAL INTELLIGENCE

J. GLASGOW and R. BROWSE
Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada

(*Received September* 1984)

Abstract—This paper briefly describes the two most popular programming languages for artificial intelligence applications: LISP and PROLOG. The capabilities and limitations of each language are reviewed in the context of establishing the main requirements placed on artificial intelligence languages.

The nested interactive array language, NIAL, is introduced as a language that combines logic and functional programming capabilities. Through comparisons with LISP and PROLOG, it is shown that the NIAL system meets the basic requirements for artificial intelligence programming.

1. INTRODUCTION

Artificial Intelligence (AI) is primarily concerned with the development of computational methods for accomplishing aspects of human intelligent behavior. This branch of computer science brings with it particular needs in the area of programming languages. Historically, AI has been a force in the creation of these special languages.

One important ingredient of an AI programming language is that it provides the ability to implement a physical symbol system[1]. This basic ability to retain and transform symbolic structures is generally viewed as central to the development of any intelligent system. While most programming languages center on an ability to manipulate numeric data, AI systems typically exploit knowledge of concepts through their representation as symbolic structures.

The exact nature of the data structures involved in AI programs is not usually known ahead of time, but is determined dynamically during the execution of the program. Thus it is generally agreed that in order for a language to be useful for AI, it should provide simple access to arbitrarily complex and dynamic structures such as lists, trees, and arrays. It is also important that reference to such data structures be kept simple.

An ideal AI programming language should provide mechanisms for the expression and manipulation of real world knowledge. This is normally done using a logic formalism that allows inferences. To be effective such an AI language must contain a standardized control mechanism and, at the same time, be conducive to the development of improved control and inference methods.

AI languages exhibit four additional properties: (1) They are interpreted rather than compiler-based; (2) they do not rely on an underlying Von Neumann style of computation; (3) they are based in solid mathematical frameworks so that formal reasoning about the adequacy and correctness of programs may be accomplished; (4) they have friendly programming environments and good user interfaces. These programming language features are desirable for the implementation of any complex system and may only be more firmly entrenched in the AI languages because of the inherent complexity of AI programming requirements.

The first part of this paper describes the two prototypical AI programming languages: LISP and PROLOG. Some of the main applications and extensions of these languages are reviewed in order to explore their merits and limitations.

Programming languages for AI have normally been a proving ground for new languages and data forms that later become part of the general language stream. As a result, multipurpose programming languages can be constructed that also meet the necessary criteria for practical AI problem-solving. We introduce such a language in this paper. NIAL is a high-level, interactive programming language that synthesizes many semantic concepts from LISP and APL in one notational framework. NIAL also includes conventional, structured programming tools and implements many of the ideas in Backus's FP[2]. It is based on the formal theory of the nested rectangular array as a mathematical data object. This theory was introduced by T. More[3,4]

of the IBM Cambridge Scientific Centre and was later extended to the nested interactive array language, NIAL, by More and Jenkins[5].

A subsystem of NIAL has been defined by Glasgow[6] that allows for logic programming within the NIAL environment. Thus NIAL not only has the ability to functionally manipulate flexible data objects, but it can also perform inferences on logical data represented as embedded arrays. Although the logic component of the NIAL logic system is based on resolution, as is PROLOG, the control strategy corresponds to the heuristic style of search common in AI applications. This paper includes a description of the programming language NIAL, its logic system, and a discussion of NIAL as a tool for AI problem-solving.

2. LANGUAGES FOR ARTIFICIAL INTELLIGENCE

2.1 *LISP*

Dating back to 1957[7], LISP is the second oldest programming language among those still in use. The language was formulated in an attempt to improve upon IPL[8], and, in particular, the motivation for the language centered on its list processing capabilities (for which the name LISP is an acronym).

LISP has developed along lines that are quite separate from those of the algorithmic languages such as FORTRAN, ALGOL and PASCAL. The result has been the availability of programming tools capable of succinct representation of AI concepts[9]. The existence of LISP is the factor most responsible for the rapid progress of AI research over the last 20 years.

Today several elaborated versions of LISP exist, complete with assortments of debugging and editing facilities: INTERLISP[10], UCI-LISP[11] and FRANZLISP[12]. Specialized computers have also been developed that are dedicated to LISP execution[13].

Following a brief outline of LISP, this section will describe some of the capabilities that have been added to LISP to meet the requirements of specific AI applications.

2.1.1 *Description of LISP.* There are only two levels of data objects in LISP. The most basic form is the *atom*, which is simply a character string used as a symbolic representation. The following are some examples of atoms:

```
broccoli  carrots  t15  12.75.
```

The higher-level data structure is the S-expression (symbolic expression). An S-expression is either a single atom or a linked list of S-expressions.† S-expressions are therefore capable of encoding arbitrary tree-structured data. Associated with this linked list data type is an unambiguous one-dimensional surface representation through which the LISP user may communicate. In this representation, lists are enclosed in parentheses as shown in the examples below:

```
(broccoli  carrots  beans)
(vegetables  (broccoli  carrots  beans)).
```

S-expressions are the only form encountered in pure LISP. All programs, as well as all data, conform to this structure and are expressible in the surface representation as parenthesized lists.

While it is traditionally thought that in order to be more useful a programming language should provide a variety of data structures (such as records, sets and arrays), the philosophy of LISP is to provide a single powerful representation and permit the user to focus on the problem without the burden of attending to a variety of syntax and type regulations.

LISP is an interpreted language that executes a simple cycle of accepting input S-expressions, evaluating them, and printing out the results of evaluation. In the jargon of the language this is known as the Read–Eval–Print loop.

There are mechanisms by which individual atoms may be *bound* to arbitrary S-expression values. If an atom is passed to the evaluator, then it simply returns the value to which it is bound. Numeric and truth value atoms evaluate to themselves. Most often, however, list S-

†One may alter the linked list structure to obtain variants of the S-expression.

expressions will be passed to the evaluator. In this case the first element of the list is taken to be a function name, and the rest of the elements in the list are taken to be arguments of the function. For example:†

```
⟶ (times 7 2).
14
```

Before the application of the function, the arguments are, in most cases, passed through the evaluator.

```
⟶ (times (plus 7 3) (plus 4 1)).
50
```

Thus LISP list S-expressions encode arbitrarily complex prefix expressions. The language includes a large set of built-in functions that provide the capabilities expected of high-level languages.

There are functions that allow the definition of new functions. The newly defined functions are represented using the λ-calculus notation of Church[14]. The example below demonstrates the definition of a function for computing the area of a circle.

```
⟶ (def circle-area
      (lambda (r)
          (times pi (times r r)))).
```

The function ''def'' does not evaluate its arguments, but simply associates the second element of the list (in this case the atom ''circle-area'') with the λ-expression given as the third element of the list. In the event that this new function is applied to some arguments, such as

```
⟶ (circle-area 7.0)
153.93791
```

the atom *r* would be temporarily bound to the value 7.0 during the evaluation of the λ-expression's body. Following the rule that arguments are evaluated before functions are applied, the value 7.0 will become the actual argument to the first invocation of the function ''times.'' Within the execution of this particular function, the variable ''pi'' is not bound within the λ-expression. Thus the value of pi must have been set outside the function as a global (or free) variable. Most programming languages that permit the use of unbound variables resolve their values using a *static scoping* mechanism. That is, functions and procedures are defined to be embedded within one another, and the unbound variables are resolved by their location in an enclosing function. LISP is unusual in that it employs a *dynamic scoping* mechanism. This method resolves unbound variables by using the current value of the variable and imposes no embedding structure on LISP funtions. This means that a function may reference any other function, provided that the proper values of its unbound variables (if any) have been set. This feature allows the programmer to concentrate on the sequence of events that will take place at the time of execution of the program, rather than focusing on the static, or pre-execution, relation among functions.

The elimination of this static relation among functions is another means by which LISP reduces the bookkeeping overhead in program development, but the approach of dynamic scoping presents serious pitfalls in the use of programs that pass functions as arguments. Suppose we wished to implement the LISP system function ''mapc'' that takes two arguments—a function and a list (the operation of ''mapc'' is to apply the function to each element of the list). The parameters and variables used in our version of ''mapc'' must be such that no function that ever appears as an argument to ''mapc'' ever expects a variable of the same name to be resolved

†The conventions of FRANZLISP are used in the examples. The arrow is the interpreter's prompt character.

externally through dynamic scoping. Otherwise the temporary values bound inside "mapc" will override the expected values, causing erroneous results. There are several ways around this problem (known as the "funarg" problem), but these solutions are beyond the scope of this summary description of LISP (see [15]).

LISP provides a primitive data-base mechanism. Each atom may have associated with it a list of property-value pairs known as the *property list* or p-list. Simple retrieval and storage functions are provided that require the specification of the atom and property. For example,†

```
——→ (putprop 'broccoli 'green 'color)
green
——→ (get 'broccoli 'color)
green
```

One of the common uses of property lists in AI programming is as a means of implementing networks. Atoms are used to represent the nodes, and the properties are the labelled arcs (see[16]).

The property list is a global data base. Whereas the value of an atom is often updated in response to its use as a parameter of a function, the property list is only modified explicitly through the access and retrieval functions.

LISP was originally designed as a language for the expression and manipulation of mathematical formulae[9]. As a result, the language includes conditional expressions and recursive definition capabilities that permit the user to express computation in a manner that corresponds well with the usual recursive function definition format.

The LISP evaluation operation is available to the programmer as the function "eval." This, together with the fact that the syntactic form of LISP programs corresponds to the data structure of the language (S-expressions), means that it is possible to manipulate and analyze LISP code under program control. In addition, "eval" describes the interpretation of LISP programs, providing a clear and well-understood operational semantics for the language.

2.1.2 *LISP extensions and applications.* Within most LISP systems only a small portion of the functions are coded in assembly or some other low-level language. The remainder of the programming language capabilities are coded in LISP. If a user is dissatisfied with some particular LISP function, it can be easily modified. Similarly, the surface syntax can be changed to incorporate the familiar keywords of algorithmic languages. More likely, a user will add functions to LISP in order to facilitate the particular problem-solving approach being used. In many cases the net result can be viewed as a specialized language embedded in LISP, particularly if the user modifies "eval" itself, or changes the Read-Eval-Print loop.

In the early 1970s many experiments took place aimed at the expansion of LISP into new AI programming languages through the inclusion of sophisticated data-base access and control mechanisms[17–19]. Some of the ideas embodied in these systems have a legacy in logic programming systems such as PROLOG, but the real success of these experiments was to demonstrate the versatility of LISP. It soon became apparent that a wide variety of specialized extensions could be quickly implemented. LISP textbooks have appeared that provide concise descriptions of prototypes for these extensions[16, 20–22], making their implementation straightforward enough that interest has shifted from exotic AI languages built upon LISP to the modern versions of LISP with their rich programming environments. The rest of this section describes some simplified examples of the type of capabilities that may be structured into LISP to facilitate AI applications.

AI application programs are often required to represent some large body of information about a problem domain. The notion of a global data base seems to be an appropriate vehicle for this representation. The global data-base facility offered by LISP is appropriate only in the simplest cases. For example, consider the requirement to identify atoms on the basis of the

†The quote mark appearing in front of an atom inhibits its evaluation as an argument, but rather the atom itself is bound to the function's parameter.

values of several properties. This would be an opaque and computationally expensive operation, using only the "putprop" and "get" facilities described earlier. The need for a more comprehensive data-base capability has been the motivation for many LISP extensions.

One approach has been to separate or divide the problem-domain knowledge into a set of assertions, where each assertion (or tuple) represents a separate fact, rather than accumulating facts as properties. For example:

```
(broccoli color green)
(broccoli vitamin-content high).
```

A generally useful data-base access method is found in the notion of *pattern matching*. In its simplest form a pattern is constructed that is structurally similar to an assertion, but may include variables that are to be consistently replaced by atoms in a data-base assertion in order to provide a match. For example, the following pattern and assertion match:†

```
(?x color green)
(broccoli color green).
```

The numerous variations on this theme of pattern matching in a data base have been influential in the development of LISP extensions. The programming language PLANNER[17] permits the definition of "theorems" capable of establishing matches in the event that none is found in the data base.

The theorems may be viewed as procedures for which a *pattern-directed invocation* establishes a nondeterministic control mechanism that backtracks through the potential methods for the establishment of a match for an initiating pattern (or goal). Techniques exist by which these capabilities may be programmed into a LISP system[16,21], but the logic programming system PROLOG (described in the next section) provides a clear and formal explanation of these operations.

In the programming language CONNIVER[18] the notion of pattern matching in a data base is carried even further. One extension is a tree of related data bases or *contexts*. This makes it possible for programs to operate on a variety of different sets of assertions and permits easy switching of contexts to facilitate, for example, search-based applications. Whereas PLANNER has automatic restoration of the data base whenever backtracking occurs, CONNIVER provides the user with access to the intermediate states of the data base. This capability is made available through a general coroutining facility that permits functions to suspend their operation, maintaining their variable bindings and data-base condition, pending later resumption.

The experiments of PLANNER, CONNIVER, and QLISP made it apparent that LISP offered an adequate base system from which the more sophisticated requirements of AI programming could be implemented in a generally useful form. But as the layers of additional capabilities mounted, the clear semantics inherent in LISP became somewhat obscured. A more formally defined programming capability incorporating backtrack search using a data base of assertions required the emergence of a programming system based on formal logic.

2.2 *PROLOG*

Programming in logic was initiated by research in automatic theorem-proving. Early programs were inefficient since they performed exhaustive searches of all possible proofs. In the early 1970s the PROLOG system (for PROgramming in LOGic) was introduced as a joint effort between the University of Marseille[23] and the University of Edinburgh[24,25]. This language proved to be a powerful tool for problem-solving. The real impetus for logic programming came in 1981 when the Japanese announced the fifth generation project. The controversial decision to base fifth-generation systems on PROLOG brought a great deal of attention to this language.

†By convention atoms preceded by "?" are takes to be variables for the pattern to match.

2.2.1 *Description of PROLOG.* The semantics of PROLOG is that of resolution logic. All PROLOG statements (axioms) are written in the form "In order to prove B first prove A_1 and A_2 and . . . and A_n." Such a statement is equivalent to the logical formula

$$A_1 \,\&\, A_2 \,\&\, \cdots \,\&\, A_n \longrightarrow B.$$

PROLOG uses logic as a tool for solving problems in AI. Knowledge about a problem environment can be stored as axioms in the language, and logical inferences performed on these axioms to derive new information. For example, we may have in our knowledge base the fact that all vegetables are edible and that broccoli is a vegetable. This would be represented as

```
edible (x) ⟵ vegetable (x).
vegetable (broccoli) ⟵.
```

From this we can state the goal

```
⟵ edible (broccoli)
```

which asks if broccoli is edible. Using the PROLOG rule of inference, we can derive this fact from the axioms of the knowledge base. The remainder of this section is a more formal description of resolution logic and the control strategy implemented in PROLOG.

In resolution logic a *clause* is an expression of the form

$$B_1, \ldots, B_m \longleftarrow A_1, \ldots, A_n,$$

interpreted as

$$(x_1) \cdots (x_k)\, A_1 \,\&\, \cdots \,\&\, A_n \longrightarrow B_1 \vee \cdots \vee B_m,$$

where the A's and B's are atomic formulas, and m and $n \geq 0$. We call the A's the conditions of the clause; the B's, the consequences. The free variables $x_1, \ldots, x_k$ of the statement are all considered to be universally quantified.

An *atomic formula* is an expression of the form $P(t_1, \ldots, t_r)$, where P is an r-place predicate symbol, the t's are terms, and $r \geq 1$. The expression is interpreted as the relation P that holds for terms $t_1, \ldots, t_r$. In general, a *term* can be a constant symbol, a variable, or an expression of the form $f(t_1, \ldots, t_r)$, where f is an r-place function symbol, $t_1, \ldots, t_r$ are terms, and $r \geq 1$. Finally, a *sentence* in resolution logic is a possibly infinite set of clauses interpreted as a conjunction of the clauses.

A PROLOG program involves a sentence consisting of a finite set of *Horn clauses*. Syntactically, a Horn clause is one containing at most one consequent. This special subset of clauses has four classifications:

unconditional assertions of the form $B \longleftarrow$ (simply written B),
procedure declarations of the form $B \longleftarrow A_1, \ldots, A_n$,
goals $\longleftarrow A_1, \ldots, A_n$,
and contradictions $\longleftarrow$ (i.e. no conditions or consequence).

Execution in a PROLOG program is initiated by a goal statement. That is, a procedure

$$B \longleftarrow A_1, \ldots, A_n$$

is invoked by a procedure call given by the goal clause

$$\longleftarrow B_i,$$

provided B can be *unified* with B_i. We say that two expressions E_1 and E_2 are unifiable if and only if there exists a *substitution* $\Theta = \{x_1/t_1, \ldots, x_s/t_s\}$ such that $E_1\Theta = E_2\Theta$. For any expression E, $E\Theta$ is the expression identical to E except that every free occurrence of variable x_i is replaced by term t_i as defined in Θ, where $1 \leq i \leq s$. For example, consider the clause edible(x)←vegetable(x) and goal ←edible(broccoli). The substitution Θ = {x/broccoli} results in the two expressions edible(x) and edible(broccoli) being unifiable.

Once two clauses are unified (unifying substitution has been applied to both clauses), we can apply the *principle of resolution* to obtain a new goal clause. In the previous example the resolvent would be

$$\longleftarrow \text{ vegetable (broccoli).}$$

The resolvent is the conjunction of the two sets of conditions other than the unified condition from the two clauses.

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 given goal. If such a clause exists, then the conditions resulting from the unifying substitution on the clause becomes the current goal statement. When no unifying clause can be found, or all alternatives have been tried, the system backtracks to the last choice point where unification was carried out and searches for another alternative.

We can view the solution of a goal as a series of *implication trees* describing the current state of the proof. Such a tree consists of nodes labelled with subgoal expressions. The root of the tree is always the initial goal of the proof, and the *children* of any node are the subgoals generated by applying resolution to the given node expression and a unifiable clause. The control strategy of PROLOG corresponds to a leftmost, depth-first search of the implication tree. A tree is completed and the initial goal solved if all subgoals have been proven. This occurs when all the leaf nodes of the implication tree have been unified with an unconditional assertion. If a tree cannot be completed, then PROLOG has failed to prove the goal.

Since the announcement of the Japanese fifth-generation project, there has been much research towards the development of a parallel inference machine. This also has led to the design of several extensions and variants of PROLOG to allow for parallel implementations. An extended PROLOG for execution on a data-flow machine, EPILOG, has been defined[26]. Shapiro's CONCURRENT PROLOG[27] is another attempt at defining a parallel PROLOG.

There are three general approaches to implementing parallelism in PROLOG:

AND parallelism assigns a process to each conjunct in a clause. Variables shared between goals must be reconciled when separate processes result in nonunifiable substitutions.

OR parallelism assigns a process to each list of subgoals resulting from a unification with a goal. The solution is considered to be a disjunction of the results of the individual processes. Thus only one of the subgoal lists must be solved to imply that the original goal has been proven. There is no sharing of variables between processes in OR parallelism and, therefore, no reconciliation problem.

STREAM parallelism can be considered as pipelining data. If two goals are acting on a list, where the output of one is the input to the other, then the list elements could be pipelined through processes that correspond to the independent goals.

Variants of PROLOG have also been developed for particular AI applications. Motivated by frame-based languages, such as FRL[28] and KRL[29], an extension of PROLOG has been designed for the purpose of knowledge representation. PROLOG/KR[29] provides a multiple-world mechanism. This, combined with the logical semantics of PROLOG, results in a modal logic formalism.

PROLOG-like systems also exist as embedded logic systems in functional languages. Examples of these will be discussed later.

2.2.2 *AI applications of PROLOG.* PROLOG is currently being used for many AI applications, including natural language understanding, theorem proving, expert systems, and architectural design. In this section we concentrate on two of these areas: expert systems and natural language understanding systems.

The development of *expert systems* is a rapidly growing field in AI. Because of this it is difficult to discern a common set of requirements or desirable features for these systems. Clark and McCabe[31] describe how PROLOG can be used to implement what they consider three such important features. The first of these is the ability to request input based on system-generated inferences. This is relatively simple to achieve in PROLOG, using the primitive predicates "read" and "assert." These allow input data to be entered at the terminal and new assertions and rules to be added to the program when desired. Also, the "assert" primitive can be used to allow the program to modify itself.

Another essential feature described by Clark and McCabe is the incorporation of some means for attaching probabilistic weights to inferences. This also can be accommodated in PROLOG by attaching an extra argument to a predicate that denotes a probability associated with the predicate.

The final characteristic necessary in an expert system is the ability to explain the inference system to the user. Since the semantics of PROLOG is based on the formal system of resolution logic, a single inference in the language is simple to explain and understand. PROLOG also provides a trace facility that may be used to keep track of the assertions invoked. Thus, if the user queries the system, a sequence of inferential steps can be displayed. Aside from the mentioned criteria for expert systems, PROLOG has the feature of both representing and making inferences on knowledge, using one logical formalism.

As mentioned previously, another major application of logic programming is *natural language* processing. The idea of using logic as a framework for a natural language system is not a new one. Green's work in the late 1960s[32] was one of the first systems to use logic as a framework for a question answering system. The fact that logic can deal with the notion of logical consequence makes it particularly well suited for representing the meaning of natural language utterances.

Kowalski[24] first suggested that natural language grammars can be expressed in predicate logic and, more specifically, in a subset of predicate logic consisting of Horn clauses. His basic premise is that Horn clauses are simply a natural extension of the rewriting rules of a context-free grammar.

There are several advantages to using a logic for natural language processing. The most obvious one is the need for inference for handling world knowledge. Natural language systems developed within the framework of logic programming employ logical inference as the sole procedural mechanism in the systems, as opposed to other systems that must use special-purpose formalisms.

PROLOG has made programming in logic feasible. Coupled with the fact that most PROLOG implementations include a version of metamorphosis grammars[23], a logic-based formalism for describing natural language grammars, logic can now be used both as the underlying formalism and as the programming tool.

2.2.3 *Problems with PROLOG.* Although PROLOG was a major step forward in logic programming, there are several drawbacks to the language. One of these exists in the control mechanism of PROLOG. The main difficulty here is that the user must understand a complex method of search and backtracking in order to construct a program. As well as being hard to comprehend, the control strategy does not embody the simplicity of the logic theory on which the language is based. "Trick" predicates, such as "/," have been introduced to the syntax to control backtracking. The semantics of this predicate has no basis in resolution logic.

PROLOG has been avoided by many LISP users because of its poor programming environment. User interaction, error detection, interactive program construction, etc., are all difficult to do in current PROLOG implementations. This has been a major motivation behind development of systems that embed PROLOG within the LISP environment.

There are several research projects currently underway to implement parallelism in PROLOG. Since PROLOGs control strategy does not naturally conform to concurrency, most of this work

involves modifying the language to fit a model that does correspond to parallelism. As a result, the simple semantic model of PROLOG is once again compromised.

3. NIAL

NIAL is an interactive, general-purpose, programming language that has data-structure concepts similar to those of APL and LISP. Based on array theory, it provides concise mathematical descriptions of real world data objects and their manipulations. Diagrams are used in the language to illustrate the nesting, content, and rectangularity of arrays.

3.1 *Array theory*

Array theory consists of a universe of data of one particular type: finite, nested, rectangular arrays. Similar to set theory, array theory involves concepts of aggregation and membership of objects within a collection. The theory is one-sorted because every data object is an array. The items of an array that are gathered together in rectangular arrangement along any number of axes of various lengths may themselves be arrays to any depth of nesting. A *table* of coefficients, such as a matrix or multiplication table, a *list* of letters, such as a word, and a *single* object, such as a number or truth-value, are examples of rectangular arrays having two, one and zero axes, respectively.

In pure array theory there are only two types of primitive objects—ordinal numbers and Boolean truth-values—but extensions have been made to include several other types, including characters, phrases, faults, etc.

An operation in NIAL is a function that maps an array to another array. The basic operations of the theory are total in that they are defined for all arrays. User-defined operations can be constructed in several ways, such as composition, transformation, currying, or lambda form.

3.2 *Array diagrams*

Arrays in NIAL are displayed using diagrams to illustrate the contents of the array as well as the degree of nesting and the number of dimensions. We will consider three primitive array types: single, list, and table with valences zero, one, and two respectively. Figure 1 is an example of each of these types.

A diagram of nested arrays can be built out of primitive arrays. Consider the list that contains the list array and table array above as its two components (Fig. 2).

3.3 *Operations*

The operations of NIAL are primarily designed for the manipulation of nested arrays. These operations can be broken down into several subclasses: those that are applied to lists, logical operations, selectors, arithmetic operations, etc. We describe some of these operations and present examples of their use in the remainder of this section.

3.3.1 *Lists.* As illustrated earlier, the list is one form of array in NIAL. There are several operations associated with lists. In this section we will discuss those operations employed later in the user-defined operations for resolution logic.

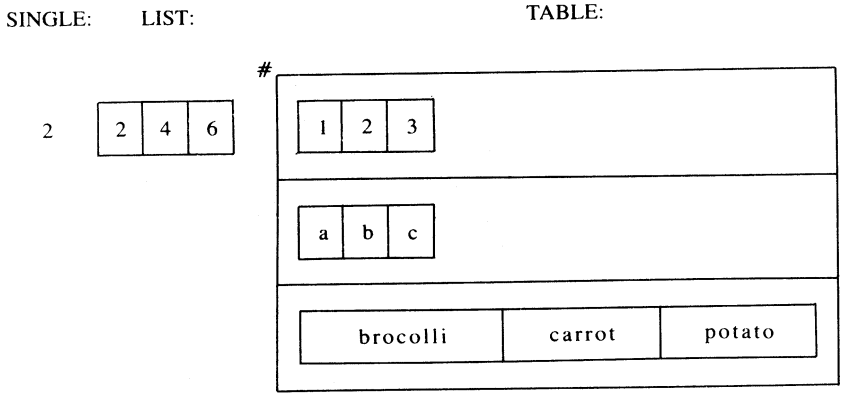


Fig. 1.

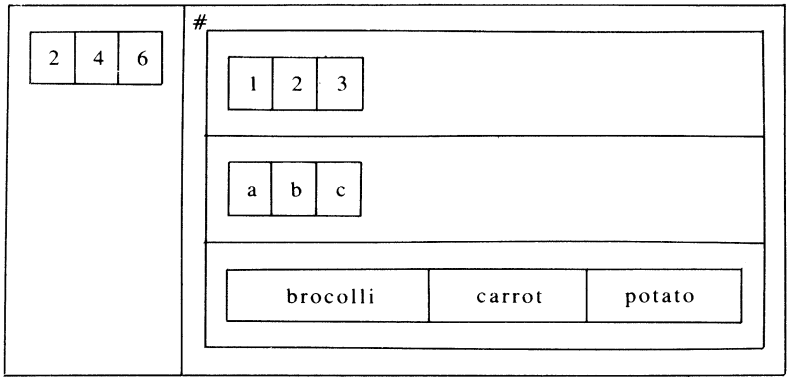
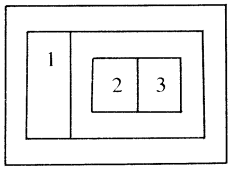


Fig. 2.

The prefix operation *solitary* is used to construct a list with one item. For example,

solitary 1 (2 3)



Strands are the notation for explicit lists in NIAL. Each item of a strand is considered an individual syntactic unit or is made one by enclosing parentheses.

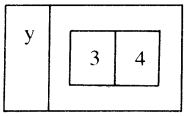
The operations *first* and *rest* are used to decompose a list into two components: the first item of the array, and the list consisting of all items but the first. For example,

first (``x 2) ``y (3 4)

and

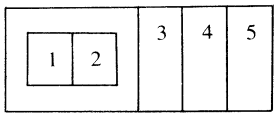
rest (``x 2) ``y (3 4)

is the array



To rebuild arrays from their first and rest items, we can apply the operation *hitch*: i.e.

1 2 hitch 3 4 5



3.3.2 *Logical operations.* NIAL provides several operations to perform logical functions. To denote the constants true and false, the special characters o and 1 are used (corresponding to boolean 0 and 1). For later programs we are primarily interested in the operations *equal*, which confirms or denies whether two arrays are identical, and the logical operations *and*, *or*, and *not*. The following examples illustrate the use of these logical operations.

```
  2  (3  4) equal 2  (3  4)
I
  (2  3)  4 equal 2  (3  4)
o
  and IIo
o
  or IIo
I
```

3.3.3 *Array operations.* As illustrated by the following examples, NIAL contains many operations to construct, decompose, etc., arrays:

```
          shape(4  5  6)
3
          tell 4
0  1  2  3
#                              2  2 reshape (tell  4)
```

0	1
2	3

The operations of NIAL can be grouped into atlases similar to the combining forms used by Backus[2]. Each operation in an atlas is applied to the array argument, and the result is the array formed by all individual function applications. For example,

```
[first,second,last][0,1,2,3,4]

0  1  4
```

3.4 *Transformers*

Transformers in NIAL allow the mapping of operations onto new operations that may be applied a number of times to their arguments. In particular, EACH is a transformer that applies an operation to all items of an array. For instance

```
EACH foo A  B  C
```

is the array with three items: the values of foo A, foo B, and foo C.

FORK is a transformer that expects an atlas of three operations. The first operation is applied to the argument, and if it evaluates to true, then the second operation applied to the argument is returned. If it is false, then the application of the third operation is returned.

```
FORK [equal,first,second](1  2)

2
```

3.5 *Programming in NIAL*

In the following examples we restrict ourselves to a small subset of the language and to a particular style of programming. The user-defined operations presented are functional, in that they are strictly lambda expressions describing the result of applying the definition to some argument list of arrays.

NIAL has a small, powerful set of definitional mechanism for expressions, operations, and transformers. These are all of the IS form, in which an identifier is defined to be either an expression operation or transformer. Operations and transformers can be defined in a lambda or lambda-free form.

```
second IS first rest
```

(note that the above definition is interpreted as the composition of the operations first and rest)

```
foo IS OPERATION A   B(
            if A allin B
                then B
                else cull link A   B
            endif)
each2 IS EACH EACH
```

An important characteristic of NIAL is that it allows for either infix or prefix forms:

```
foo A   B  =  (A foo) B  =  A foo B
```

NIAL also has nonfunctional features that allow for assignment of variables and iterative constructs.

3.6 *NIAL implementations*

NIAL is currently implemented by a single-process interpreter written in C, and running on several machine/operating-system combinations (i.e. VAX/UNIX, VAX/VMS, IBM PC/DOS, MC68000/XENIX, SUN/UNIX, IBM/VM-CMS, DEC 20/TWENIX, and NS16032/UNITY). A single-process VLSI-compatible architecture is also under development. Future work includes investigating a parallel architecture for implementing NIAL. It has already been recognized that the language captures a powerful notion of concurrency[33]. This is partially due to the family of EACH transformers. When evaluating EACH foo A, all the applications of foo to the items of A can proceed in parallel.

4. NIAL AS A PRACTICAL TOOL FOR AI PROGRAMMING

In this section we discuss NIAL as a practical tool for solving problems in AI. NIAL contains many features useful in such a language. It is interactive and supports flexible control structures and data types. It also allows a natural and useful implementation of logic.

4.1 *Functional versus logic programming*

Similar to LISP, NIAL is considered a functional language. It also resembles LISP in its convenient interactive environment and use of recursion. Within NIAL, files can be created and edited; functions defined and edited; workspaces created and retrieved; etc.

Although functional languages can be used to represent and make inferences on real-world knowledge, we demonstrated in a previous section that logic programming languages do this more easily, although in a limited working environment. On the other hand, there are applications in which the computational powers and expressive data structures of LISP-like languages are more advantageous. This implies that a practical programming language for AI must have both the expressive powers of PROLOG and the rich computational environment of LISP. Research involving such systems is currently being carried out. LOGLISP[34] is a system that provides a logic programming LISP-like environment offering both LISP and logic. Another programming environment for PROLOG in LISP is presented in QLOG[35]. In this paper we attempt to combine ideas from each of these approaches. We embed, similarly to QLOG, logic programming within an already existing environment that allows for efficient implementations. A major feature of LOGLISP, which we maintain, is the ability for the user to create the logic control strategy.

4.2 *Logic programming in NIAL*

As illustrated earlier, logic programming languages allow for meaningful representation and manipulation of knowledge. Thus they are useful for AI applications such as expert systems

and natural language systems. In this section we develop a logic programming system based on array theory. This is done by defining the concept of resolution logic, using nested rectangular arrays and operations that perform logical functions on such arrays. The state of a logic proof is represented by an array corresponding to the tree generated by the and-or tree problem reduction method[36]. Unlike PROLOG, whose control strategy is a rigid depth-first search, we allow the user the flexibility to introduce alternative control mechanisms. Thus, what we are providing is a set of logical tools that can be utilized within the functional environment of NIAL.

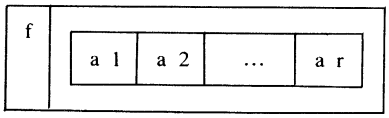
4.2.1 *Resolution logic in NIAL.* The same principles of resolution logic employed in PROLOG can also be represented in NIAL. We illustrate this by defining an embedded array representation of PROLOG syntax and providing NIAL functions that perform substitution, unification and resolution on these array structures.

Representation of Horn clauses in array theory. The approach we take to represent clauses as arrays is based on two factors: ease of understanding, and ability to effectively carry out the functions that will be performed on the arrays.

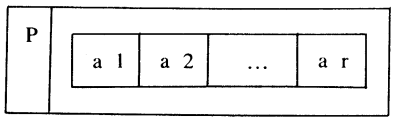
We inductively define the array representation of the syntactic classes of the language PROLOG. Consider first a term. If the term is a constant or variable symbol c, then we will represent it as the single c (array of valence 0)

$$c$$

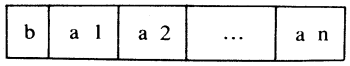
(Henceforth we will use the symbols $a,b,c, \ldots$ to denote constants and $u,v,w,x, \ldots$ to represent variable terms.) For the case where term t is an expression of the form $f(t_1, \ldots, t_r)$, where f is an r-ary function symbol and the t's are terms such that a_j is the array that represents term t_j $(1 \leq j \leq r)$, represented as the array:



Suppose we had an atomic formula of the form $P(t_1, \ldots, t_r)$, where P is an r-place predicate symbol and the t's are terms. Then, assuming that t_j $(1 \leq j \leq r)$ is represented by array a_j, we would represent the atomic formula as the array:



Using the above array definitions, we would represent the Horn clause $B \leftarrow A_1, \ldots, A_n$ as the array:



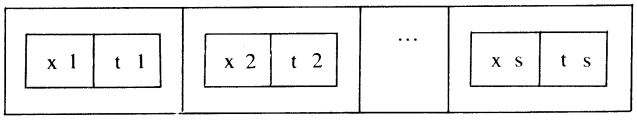
where b and a_j represent atomic formulae B and A_j respectively.

Finally, we can consider a logic sentence as a list of clauses or a table in which each row contains a definition of a procedure.

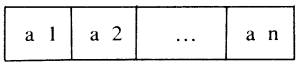
Substitution, unification, and resolution. In the previous section we illustrated how the syntax of logic can be represented using embedded arrays. NIAL also provides functions that

can perform logic operations on such arrays. In particular, built-in functions SUBSTITUTE, UNIFY, and RESOLVE are provided as tools for logic programming.

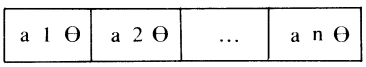
Given an expression E, represented by array A, and a substitution $\Theta = \{x_1/t_1, \ldots, x_s/t_s\}$ for some distinct variables $x_1, \ldots, x_s$ and terms $t_1, \ldots, t_s$, we must define the new array that presents $E\Theta$. The substitution Θ can be considered as a list of substitution pairs:



The result of applying the operation SUBSTITUTE to the array A and the substitution list Θ can be defined inductively on the structure of A. If A is a single, then "SUBSTITUTE A Θ" results either in the single A or in t_i, depending on whether A occurs on the left side of a substitution pair or not. Suppose that array A is the list:



Then "SUBSTITUTE A Θ" is the array:



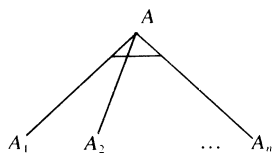
where $ai\Theta$ is the subarray resulting from "SUBSTITUTE $ai\Theta$."

Unification in NIAL logic consists in determining, if possible, a unifying substitution for two arrays. Array A_1 and array A_2 are unifiable if there exists a list, Subarray, of legal substitutions such that "SUBSTITUTE Subarray A1" and "SUBSTITUTE Subarray A2" result in the same array. Thus, the function UNIFY applied to two arrays either returns an array representing the substitution pairs resulting from the most general unification of the two arrays or returns o (failure) if no such substitution list exists.

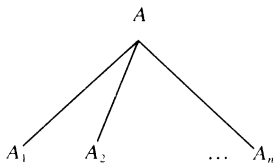
Resolution is also defined by NIAL logic. If array A1 represents a Horn clause that is unifiable with an array A2 representing a goal clause, then "RESOLVE A1 A2" results in the array that denotes the resolvent of the two parent clauses.

4.2.2 *Control component of NIAL logic.* The logic component of NIAL, as described in the previous section, is based on resolution logic. Thus it is similar in nature to the logic of PROLOG. Since we wish to maintain a direct correspondence between our system and a formal model, we develop the control strategy of the NIAL logic system around the theory of *and-or tree* representation for problem reduction. In addition, a heuristic search method is introduced that determines how the trees are to be generated in order to carry out a proof. Because we are developing a set of logic tools in NIAL, it is possible for a user to specify an alternative heuristic function to the one described in this paper.

Goal reduction using and-or trees. A common method for solving a goal is to reduce it to a series of subgoals to be solved. This may consist of either a conjunction (solving all) or a disjunction (solving one) of the subgoals. Suppose we had a goal A and subgoals $A_1, \ldots, A_n$. We can represent the conjunctive tree as:



which denotes A_1 & A_2 & $\cdots$ & $A_n \rightarrow A$, and the disjunctive tree as:



representing the relationship $A_1 \vee A_2 \vee \cdots A_n \rightarrow A$.

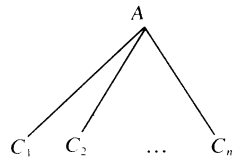
The task of repeatedly replacing goals by subgoals can be performed using the and-or tree representation until the initial goal is solved. We can inductively define what it means for a goal to be solved:

—If goal A is a leaf node of an and-or tree and A is an assertion in our system, then A is solved.

—If goal A is the conjunction of goals all of which are solved, then A is solved.

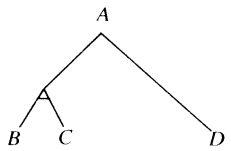
—If goal A is the disjunction of goals of which at least one has been solved, then A is solved.

We can interpret the current state of an and-or tree as a logical statement containing all the leaves of the tree. This expression will be the description of the subgoals that must be solved in order for the initial goal to be solved. We can also transform all trees into equivalent disjunctive normal form trees of the form:



where C_i are one level conjunctive trees for $i = 1$ to n. For the initial goal A to be solved, it is necessary to solve all the subgoals in any one of the conjunctive subtrees. Such a tree can be represented as an array whose items are subarrays denoting each of the conjunctions.

Goal reduction applied to resolution logic. Resolution logic can be interpreted as a form of goal reduction that can easily be represented using and-or trees. For instance, given Horn clauses $A \leftarrow B$, C and $A \leftarrow D$ and a goal A, we can reduce A to the problem of solving either goals B and C or goal D, represented as the tree:



PROLOG performs a limited version of problem reduction in its control strategy. The restriction is that it only considers one Horn clause at a time and thus creates a conjunctive implication tree. Backtracking is performed in order to attempt alternative solutions.

The and-or tree goal reduction method for determining a proof is used in the NIAL system. At each step an array corresponding to the current and-or tree in disjunctive normal form is generated.

Heuristic search strategy. Before we can determine how to generate the arrays corresponding to the and-or trees, we must define the order in which the subgoals will be solved. It has been noted that the leftmost, depth-first traversal used in PROLOG assumes a preordering on clauses. To eliminate this need we attach a heuristic to each conjunction of subgoals that is currently being considered. The heuristic function can be specified by the user to correspond to the problem being solved. One general heuristic that is currently implemented is based on

the minimum number of resolutions that must be performed in order to possibly solve the conjunction of goals—i.e. the number of subgoals in the conjunction.

One methodological advantage NIAL logic has over PROLOG is the ability to represent the stages of a proof, using an illustrative data structure. As described earlier, the state of a NIAL proof can always be denoted as an and-or tree in disjunctive normal form. Corresponding to such a tree is the disjunctive normal form array. It is possible in NIAL to generate the successive goal arrays that lead to a logic proof.

5. CONCLUSIONS AND FUTURE WORK

NIAL derives partially from LISP, so it is not surprising that there are many similarities between the languages. There is a direct correspondence between the representational capabilities of the one-dimensional nested arrays of NIAL and linked lists in LISP. Most of LISP's primary list operations have counterparts in the built-in operations of NIAL. Both languages are conducive to a functional style of programming, exploiting recursive function definitions. While LISP has its mathematical foundation in the λ-calculus, NIAL is based on the mathematical functions of array theory. Each is able to represent programs in a natural way within the data structure of the language, each provides access to the evaluation capabilities of its interpreter, and, thereby, each maintains a clear operational semantics.

The most apparent difference between the languages is that NIAL offers a richer surface syntax than LISP. It may be argued that there are merits in the uniformity of LISP's representations, particularly in such applications as planning. However, it is common criticism that LISP requires the user to balance unwieldy depths of parentheses. NIAL maintains a context-dependent, surface syntax that requires some familiarity with the language to take full advantage of its expressive power.

LISP demands a view of the construction of programs that is quite different from conventional languages. Researchers in AI have found this style generally appropriate for their applications. However, in those circumstances for which a problem has a more natural solution in the style of algorithmic languages, programmers often find LISP to be cumbersome in its expression. NIAL, on the other hand, provides mechanisms for the blending of these programming styles.

One fundamental difference between the two languages is in the treatment of functional mechanisms. LISP has only one—the untyped λ-expression—whereas NIAL draws the distinction between functions that accept only arrays as arguments (operators) and those that accept operators as arguments (transformers). With this distinction, NIAL is able to avoid most occurrences of the funarg problem (see Section 2.1), because functional arguments are detectable by the interpreter. This stratification of functions is a major contributer to the more complex surface syntax, since the meaning of the juxtaposition of objects depends on their class.

LISP provides a few simple second-order functions, such as "mapc" and "mapcar," that apply functional arguments to each element of a list, whereas NIAL offers a rich assortment of transformers that embody the same effect. One example is LEAF, which applies a function to each of the leaf nodes of the tree represented by the array argument. Of course, any of these transformers may be easily coded in LISP, but, because of the funarg problem, inclusion of these capabilities would result in an *ad hoc* distinction between variables available for use in encoding the transformers and those available to the functions that may be applied by the transformers. The philosophy of NIAL is that this required distinction is more properly relegated to the level of a distinction among functional elements clearly expressed in the surface syntax of the language.

In a paper describing the history of LISP[9], John McCarthy says

> LISP will become obsolete when someone makes a more comprehensive language that dominates LISP practically and also gives a clear mathematical semantics to a more comprehensive set of features.

It is too early to say, because existing NIAL implementations are still in the experimental stage, but to date the programming language NIAL has a great possibility of fulfilling these require-

ments. This possibility is particularly encouraged by the demonstrated elegance in incorporating logic programming capabilities into the language.

The nature of the problems that are solved in AI imply that languages used in this area must have unique qualities. We have shown how the languages PROLOG and LISP individually possess some of these properties. It was also pointed out that a system that combines functional computational powers with logical expressiveness and manipulation increases the ability for solving AI problems. NIAL was introduced as such a system.

Unlike current implementations of PROLOG, NIAL allows the user a rich environment to solve logic problems. It also permits flexibility in the control component of the logic system by allowing user specification of search strategies. Unlike systems that merge LISP and PROLOG, we have not created a new language that has both functional and logical subsets. Instead, NIAL is primarily a functional language containing operations that perform logical functions. Thus, the array-theory-based semantics of NIAL is not modified by this extension.

Much of the current and future research in NIAL applies to the area of AI. These projects can be considered in three categories: logic programming, implementations, and applications.

NIAL provides a powerful environment for research in logic programming. Currently, work is being done on extending the NIAL logic system to allow for more generalized clauses and quantification. Representation of modal logic systems is also being investigated.

To the present time the main thrust of developing logic operations in NIAL has been in describing a system that was formally definable, powerful, and, above all, simple to understand. Efficiency was not a major concern. Now that such a system exists, work is being done to improve the performance of the NIAL logic system. This includes coding the logic operations in C, the implementation language of NIAL. In addition, a NIAL chip and eventually a NIAL machine, where the basic computations are array manipulations, are being designed.

The final research project being investigated is the application of NIAL in particular areas of AI. The initial phase of this work is in natural language understanding systems. This includes a mapping from a theoretical model for natural language to the array theory on which NIAL is based. From this, a subset of the language will be implemented on concurrent architecture to perform the particular task.

Acknowledgements—This work was supported by grants from the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

1. A. Newell and H. A. Simon, Computer science as empirical enquiry: symbols and search. *Comm. ACM* **19**, 113–126 (1976).
2. J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM* **21**, 613–641 (1978).
3. T. More, The nested rectangular array as a model of data. *APL79 Conf. Proc.*, pp. 55–73 (1979).
4. T. More, Notes on the diagrams, logic and operations of array theory, in *Structures and Operations in Engineering and Management Systems* (Edited by Oyvind Bjorke and Ole I. Franksen), pp. 497–666. Tapir Publishers, Trondheim, Norway (1981).
5. M. A. Jenkins, *QNial Reference Manual*. Queen's University, Kingston, Ontario (1983).
6. J. I. Glasgow, *Logic Programming in Nial*. Technical Report 84-158, Department of Computing and Information Science, Queen's University, Kingston, Ontario (1984).
7. J. McCarthy and M. Minsky, *Quarterly Progress Report* 56. Research Laboratory of Electronics, MIT, Cambridge, Mass. (1956).
8. A. Newell, J. C. Shaw and H. A. Simon, Programming the logic theory machine. *Proc. Western Joint Computer Conf.*, pp. 230–240 (1957).
9. J. McCarthy, History of LISP. *ACM SIGPLAN Notices* **13**, 217–223 (1978).
10. W. Teitleman, A. K. Hartley, J. W. Goodwin, D. G. Bobrow, P. C. Jackson and L. M. Masinter, *Interlisp Reference Manual*. Xerox Palo Alto Research Center, Palo Alto, Calif. (1974).
11. J. Meehan, *The New UCI-LISP Manual*. Erlbaum, Hillsdale, N.J. (1979).
12. J. K. Foderaro and K. L. Sklower, *The FRANZ LISP Manual*. University of California at Berkeley, Berkeley, Calif. (1980).
13. A. Bawden, R. Greenblatt, J. Holloway, T. Knight, D. Weinreb and D. Moon, The Lisp machine, in *Artificial Intelligence: An MIT Perspective* (Edited by P. H. Winston and R. H. Brown), pp. 341–373. MIT Press, Cambridge, Mass. (1979).
14. A. Church, *Calculi Of Lambda Conversion*. Princeton University Press, Princeton, N.J. (1941).
15. D. G. Bobrow and B. Wegbreit, A model and stack implementation of multiple environments. *Comm. ACM* **16**, 591–603 (1973).

16. E. Charniak, C. K. Riesbeck and D. V. McDermott, *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, Hillsdale, N.J. (1980).
17. C. Hewitt, *Description and Theoretical Analysis (Using Schemas) of PLANNER*. Ph.D. Dissertation, MIT, Cambridge, Mass. (1971).
18. G. J. Sussman and D. V. McDermott, *Why Conniving Is Better than Planning*. AI Memo 255a, AI lab, MIT, Cambridge, Mass. (1972).
19. B. M. Wilber, *A QLISP Reference Manual*. Technical note 118, SRI International, Menlo Park, Calif. (1976).
20. P. H. Winston and B. K. P. Horn, *LISP*. Addison-Wesley, Reading, Mass. (1981).
21. R. Wilensky, *LISPcraft*. Norton, New York (1984).
22. J. Laubsch, Advanced Lisp programming, in *Artificial Intelligence: Tools, Techniques, and Applications* (Edited by T. O'Shea and M. Eisenstadt), pp. 63–109. Harper & Row, New York (1984).
23. A. Colmerauer, Metamorphosis grammars. *Natural Language Communications with Computers* **1** (1978).
24. R. A. Kowalski, Predicate logic as a programming language. *Proc. IFIP* 74, pp. 569–574 (1974).
25. R. A. Kowalski, Algorithm = Logic + Control. *CACM* (1979).
26. M. J. Wise, A parallel prolog: the construction of a data driven model. *Proc.* 1982 *ACM Symp. on Lisp and Functional Programming*, pp. 56–66 (1982).
27. E. Shapiro and C. Mierowsky, Concurrent prolog. *Proc.* 1984 *Int. Symp. on Logic Programming* (1984).
28. R. B. Roberts and I. P. Goldstein, *The FRL Manual*, MIT AI Laboratory (1977).
29. D. G. Bobrow and T. Winograd, An overview of KRL, a knowledge representation language, *Cognitive Sci.* **1**, 3–46 (1977).
30. H. Nakashima, Knowledge representation in Prolog/KR. *Proc.* 1984 *Int. Symp. on Logic Programming*, pp. 126–130 (1984).
31. K. L. Clark and F. G. McCabe, Prolog: a language for implementing expert systems. *Machine Intell.* **10** (1982).
32. C. Green and B. Raphael, The use of theorem proving techniques in question-answering systems. *Proc.* 1968 *ACM National Conf.* (1968).
33. C. McCrosky, J. Glasgow and M. Jenkins, Nial, a candidate language for fifth generation computer systems. *Proc. ACM* 1984 *Annual Conf.* (1984).
34. J. A. Robinson and E. E. Sibert, LogLisp: motivation, design and implementation, in *Logic Programming* (Edited by K. L. Clark and S. A. Tarnlund), pp. 299–314. Academic Press (1982).
35. H. J. Komorowski, QLOG—the programming environment for Prolog in Lisp, in *Logic Programming* (Edited by K. L. Clark and S. A. Tarnlund), pp. 315–324. Academic Press (1982).
36. H. Gelernter, Realization of a geometry theorem proving machine, in *Computers and Thought* (Edited by Feigenbaum and Feldman), pp. 134–152. McGraw-Hill, New York (1963).