LOGIC PROGRAMMING IN THE FIFTH GENERATION

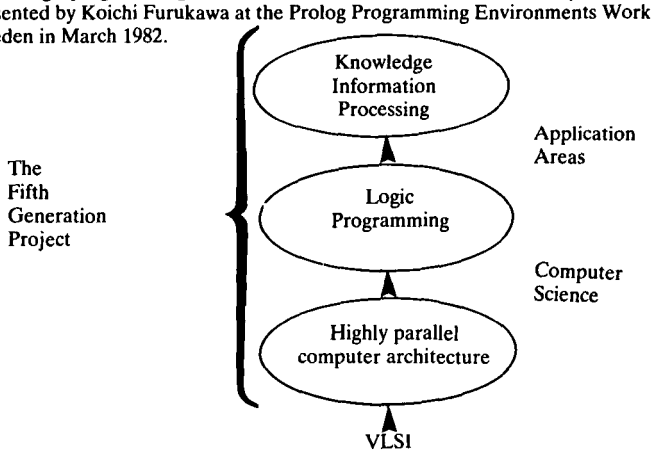
Robert Kowalski
Imperial College
London

(First appeared May 1982)

INTRODUCTION

The Japanese Fifth Generation Computer Systems (FGCS) project has chosen logic programming for its core programming language. It has recognized the major contribution that logic programming has to make not only in artificial intelligence but in database systems and software specification as well. It has recognized and intends to exploit the greater potential that logic programming has to offer for taking advantage of the parallelism possible with innovative multiprocessor computer architectures.

The role of logic programming in the Fifth Generation is best summarized by the following diagram, which was presented by Koichi Furukawa at the Prolog Programming Environments Workshop held in Linkoping, Sweden in March 1982.



The target for 1990 can be summarized as being a logic programming machine exploiting highly parallel computer architecture using VLSI technology for knowledge information processing applications. Logic programming provides the *missing link* between the very-high level rule based languages being used for applications in artificial intelligence today and the single-assignment languages and functional programming languages used by computer architects to take advantage of the parallelism made possible by VLSI.

The FGCS project also contains several shorter term targets. Notable among these are the design and construction of a "super personal Prolog machine". This machine, which is to be built within three years, will be used to support research and development for the rest of the project over the following four years.

As a precursor of the FGCS project H. Aida and Moto-Oka, who heads the FGCS project, have already built and tested a pure Prolog or-parallel, search machine with 24 Z80 microprocessors (8 for communication, 16 for processing) which they call PARALOG.

PROLOG AND LOGIC PROGRAMMING

Prolog is a modest, yet powerful, realization of the wider logic programming ideal. It is based upon the observation that declarative sentences of the form

A if B and C

(called *Horn clauses*) can be used as procedures which given problems of the form

A

reduce them to subproblems of the form

B and C.

This is the theoretical basis of both Prolog and logic programming [36, 46], which in turn is founded on

researches of the late 60's and early 70's into the design of proof procedures for automatic theorem-proving.

Prolog was developed and implemented by Colmerauer and his colleagues in Marseille in 1972. Significant improvements to the implementation of Prolog were made by Pereira, Pereira and Warren [51], who implemented a Prolog compiler in Prolog and showed that it compared in efficiency with pure LISP. Prolog has been used for such varied applications as symbolic mathematics [1, 2], plan formation [50], computer aided design [42], database description and query [25, 27], the solution of mechanics problems [3], drug analysis [21], natural language processing [15, 16, 19, 43], and automatic induction [48, 49]. Details of the language and its applications can be found in [14]. A more general introduction to logic programming is given in [38].

Logic programming is more general than Prolog both in the language features it provides and in the problem-solving facilities which it admits. For example, in the case where several "procedures"

A if B
A if C

have the same conclusion, it can exploit the possibility that the alternative ways of solving the same problem A can be pursued in parallel. This is called *or-parallelism*. In the case where a procedure has several conditions

A if B and C

The subproblems B and C can be investigated in parallel. This is called *and-parallelism*. Although Prolog already runs efficiently on conventional von Neuman computers, in the longer term logic programming offers greater scope for exploitation of highly parallel problem-solving techniques.

WHY LOGIC PROGRAMMING?

Perhaps the most surprising aspect of the FGCS project is the extent of its commitment to PROLOG and logic programming.

The explanation for this is given in the proceedings of the Tokyo conference [35] by Kazuhiro Fuchi, who is chairman of the FGCS theory group. Fuchi argues that logic programming synthesizes current trends in the otherwise disparate fields of software engineering, database systems, computer architecture and artificial intelligence including natural language processing.

In *software engineering*, as the subject matures, increasing emphasis is placed on the importance of specifications and specification languages. Symbolic logic is almost universally accepted as the most suitable formalism for problem specification. Research on automatic theorem-proving in general and logic programming in particular shows that specifications can be manipulated, executed and debugged by means of computer.

Program transformation work leads in the same direction. It starts with clear but inefficient programs written in functional programming languages and transforms them preserving correctness into more efficient programs expressed in the same formalism. Logic programming can be regarded as an extension of functional programming and program transformation techniques have been successfully extended to deal with it [4, 5, 6, 8, 11, 31, 32, 34].

In the field of *databases*, logic programming can be regarded as a significant generalization of relational databases. Query languages for user-orientated databases are based on logic. Integrity constraints, like program specifications and program properties, can only be expressed in formalisms which have the power of symbolic logic. Indeed as we shall argue later, logic programming can at the simplest level be regarded as an elegant unification of functional programming and relational databases.

In the field of *computer architecture*, researchers have invented single assignment languages to overcome the bottleneck created by the unrestricted assignment statement of conventional programming languages. These single assignment languages can be regarded as a restricted form of functional programming, which is a special case of relational programming, which is in turn a special case of logic programming.

artificial intelligence significant commercially exploitable results have begun to be achieved by *expert systems*. These expert systems are programmed in rule-based languages, most of which are implemented in the functional programming language LISP. Such rules have the form of condition-action statements.

if B and C then A

which are closely related to the Horn clause subset of logic. Recently, increasingly many expert systems have begun to be implemented in Prolog [10].

Both within artificial intelligence and in computational linguistics significant advances have been made in *natural language processing*. Both the extended attribute grammars invented for writing compilers and the generalized phrase structure grammars invented for parsing natural languages can be viewed as variants of the Horn clause subset of logic. For example, the statement

x followed by y is a sentence if x is a noun phrase
and y is a verb phrase

written in a sugared up form of Horn clause expresses a conventional context-free grammar rule. However the statement

x followed by y is a sentence if x is a noun phrase of number z
and y is a verb phrase of number z

written in the same formalism incorporates context sensitivity in a natural and computationally powerful way.

LOGIC AS A COMPUTER LANGUAGE FOR CHILDREN

At Imperial College, beginning in 1980, we have been teaching logic as a computer language to children starting at the age of 10 [26]. As a school subject it has the advantage, not only of teaching computer programming, but more importantly of teaching database definition and query as well as program specification. Moreover it fits in well with other subjects taught in school.

The children are taught a sugared up subset of logic. It is compiled by a Micro-Prolog program into Micro-Prolog internal syntax [40]. It runs on Z80 microcomputers under the CP/M operating system. A prototype system is running on the Sinclair ZX81. The following introduction to the Horn clause subject of the language is based on our experiences with teaching children.

Atomic sentences

The simplest form of Horn clause is an atomic sentence. The following are examples of atomic sentences constituting a simple database:

John likes Mary
Bob likes Mary
Mary wants bicycle
wheel part-of bicycle
tyre part-of wheel
Bob supplies wheel
Mary supplies brake-set
John supplies bicycle

Each *atomic sentence* expresses a binary relationship between two individuals and has the form:

name of individual	name of relation	name of individual

The spaces are significant and serve to separate the names of individuals from the names of relations. Such relationships, whether they occur as self-standing sentences or not, are also called *atoms*. For the moment only binary relations are allowed and individuals are named only by constants. Other options are introduced later.

Atomic queries

The simplest queries involve only a single atom and are of the Yes-No variety:

Does (John likes John)
No
Does (John likes Mary)
Yes

The system as it is currently implemented uses the closed world assumption [45]: If information cannot be derived from the database then the system assumes it to be false. This blanket assumption is, of course, extremely dangerous. For example, given the current state of the database:

Does (pedal part-of cycle)
No.

Queries can also contain variables standing for unknowns. For example,

Which (x John likes x)
Mary

Each such query has the form

Which (output-pattern conditions)

where the *output pattern* can be a single variable, i.e. one of

x, y, z, X, Z, x1, x2 etc.

or a *list* of names of individuals (including variables) enclosed in parentheses and separated by spaces, e.g.

(x y)

Each *condition* is either a simple atom (with variables standing for unknown individuals) or a conjunction of atoms. Thus we can ask

Which ((x y) x part-of y)
(wheel bicycle)
(tyre wheel)
(brake-set bicycle)

We shall see later that lists themselves can be used as names of individuals.

Compound queries

Several atoms joined by "and" and separated by spaces can be used as the conditions of a query:

Which (x Mary wants x and Bob supplies x)
No answer
Which (x Mary wants x and Bob supplies y and y part-of x)
bicycle.

General rules

A *general rule* has the form

conclusion if conditions

where "conclusion" is an atom and "conditions" is a conjunction of atoms (connected by "and" and separated by spaces). Atomic sentences can be regarded as general rules which have no conditions. This means that variables can be used in atomic sentences, e.g.

Bob likes x
i.e. Bob likes everything.

Whereas *variables in queries* stand for unknown individuals, *variables in general rules* stand for any individual. All occurrences of the same variable in the same sentence stand for the same arbitrary individual. However there is absolutely no connection between variables in different sentences even though they may have the same name. Thus the two sentences

x is female if x is-mother-of y
x is male if x is-father-of y

do not in any way imply that an individual might be both male and female.

The use of general rules promotes a database user to a programmer – or more accurately, because we have been using only the declarative semantics of queries and general rules, to a program specifier.

Recursion

General rules can be used to augment relations already in the database, e.g.

Mary likes x if x likes Mary

Here the general rule is recursive. Our experience has been that such recursion, stripped of its operational semantics, causes no conceptual difficulties for 10-11 year old children.

Not every recursion is quite so easy. Consider, for example, the "part-of" relation. At this stage, this describes only the relationship between a part and its immediate subparts. There is nothing to imply, for example, that

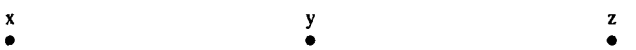
if tyre is part of wheel
and wheel is part of bicycle
then tyre is part of bicycle.

Such an inference, however, is a special case of the recursive general rule

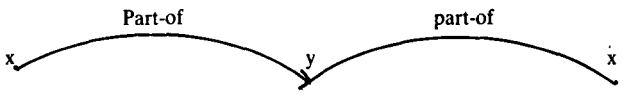
x part-of z if x part-of y
and y part-of z.

It can help to understand such recursions if we picture them graphically as "semantic networks" [22].

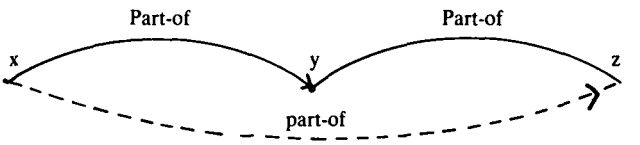
Suppose that we have three individuals, x, y, z, pictorially:



and that they are connected by part-of relationships



then we can add a new part-of relationship between x and z.



Such pictorial representations can certainly help to formulate and understand general rules. However, they are no substitute for the linear syntax. They bear a similar relationship to the logic of binary relations that Venn diagrams bear to the logic of one-place predicates, i.e. sets.

Loops

With recursive definitions we now have the possibility that the underlying PROLOG interpreter can go into a nonterminating loop. This happens already with the current state of the database and the query

Which (x x part-of bicycle).

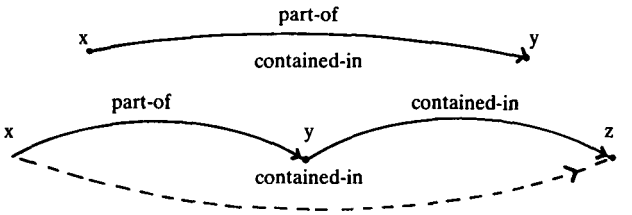
After successfully printing the first two answers

wheel
brake-set

the system goes into an infinite loop without printing the third answer

tyre.

In order to understand why this happens it is necessary to know something about the procedural interpretation and its PROLOG realization. But this defeats the object of restricting our understanding of logic to the declarative interpretation for as long as possible. For the moment we can escape this difficulty by noting tht the loop can be avoided in this and many similar cases by using different relations to distinguish between one-step part-of connections and multi-step connections. Pictorially



We can delete the old recursive rule for the part-of relation and replace it by the new rules:

x contained-in y if x part-of y
x contained-in z if x part-of y
and y contained-in z

Lists as names of individuals

The range of applications can be greatly extended by allowing lists as names of individuals. The items in a *list* are separated by spaces and can be constants, variables or other lists. Here are some examples:

John born-on (1 May 1984)
(John Mary) parents-of (George Jane Alice)

It is also possible to have partially specified lists. We use

x y

(x|y) to name the list
which starts with x followed by the list y.

in the same way that cons(x,y) is used in LISP. We use

( ) to name the empty list

like nil in LISP.

One of the most useful relations over lists is the "belongs-to" relation:

x belongs-to (x|y)
x belongs-to (z|y) if x belongs-to y.

Here the first sentence expresses that

any individual x belongs to any list (x|y)
which starts with x.

The second sentence expresses that

any individual x belongs to a list (z|y)
which starts with z followed by y
if it belongs to the list y.

The belongs-to relation can be queried as any other relation. It can be used to test that an item belongs to a list as if it were written in LISP, e.g.

Does (Bob belongs-to (George Jane Alice)).
No.

But it can also be used to generate items as if it were defined explicitly by a relational database, e.g.

```
Which (x   x belongs-to (George Jane Alice) and
           x belongs-to (Bob Mary Jane))
                 Jane
```

N-ary relations

For many applications it is natural to use one-place predicates and non-binary relations to construct atomic conditions and conclusions. We use the notation

P 1

where P is the name of an N-ary relation and 1 is a list of N arguments. Thus we can write, for example,

Supplies (x y z)

to express that x supplies y to z, and

Wants (x y)

to express that x wants y as an alternative to the "infix notation" we have used until now.

Database = program

Logic provides a straightforward and elegant method for combining explicitly stored data with general rules which compute data implicitly. The following very small database includes a program which augments the definition of the Wants relation.

Supplies (John Apples Mary)
Supplies (Mary Promises John)

Wants (Mary Friendship)
Wants (John Apples)
Wants (x y) if Supplies (z y x)
i.e. Everyone wants everything anyone supplies to him/her.

There is no difference between database query and program invocation. The same mechanism serves for both. Thus, for example, the "query"

Which ((x y) Wants (x y))
i.e. Who wants what?

both queries a database and invokes execution of a program.

The significance of this for program debugging is great. Programs written in logic can be queried as though they were a database. Rather than testing a program by sampling a sequence of inputs, e.g.

Which (x $P(i_1\ x)$)
Which (x $P(i_2\ x)$)
Which (x $P(i_3\ x)$)

it can be tested by asking for all input-output pairs

Which ((x y) P(y x)).

Moreover information about the operational behaviour of the program can be ascertained by noting the order in which the input-output pairs are generated.

PROLOG AND THE PROCEDURAL INTERPRETATION

A Horn clause of the form

A if B and C

is interpreted as a procedure which reduces A to the subproblems B and C. For the sake of efficiency,

PROLOG completes execution of the subproblem B before embarking on the subproblem C. When several clauses (atomic or compound) can be used to solve the same problem, e.g.

A if B
A if C
etc.

PROLOG tries them one at a time in the order in which they are written.

Applied, for example, to the problem

Which (x x belongs-to (George Jane Alice) and
x belongs-to (Bob Mary Jane))

given the previous recursive definition of the belongs-to relation, the PROLOG interpreter in effect generates a double nested loop: The outer loop generates consecutive elements of the first list and the inner loop tests whether they belong to the second list. Reversing the order of the two subproblems reverses the nesting of the loops.

The combination of sequential execution of procedure calls and sequential execution of procedures can lead to nonterminating loops. One such example is the recursive definition of the part-of relation which we saw before. Such loops and other inefficiencies can be overcome either by rewriting the original program or by using the extralogic primitives which PROLOG provides to control the behaviour of the interpreter. Information about these is given in [14] and [39], where it is argued that such extralogical features should be encapsulated, wherever possible, in the definition of logically defensible extensions of the language or of the problem-solving facilities of the interpreter.

Perhaps the most important extensions of the Horn clause subset of logic which can be implemented in PROLOG by using its extralogical features, are negation by failure, Horn clauses as conditions, and sets.

Negation by failure

This is illustrated by the example

Supplies (Bob x Mary) if Wants (Mary x) and
Not (Supplies (John x Mary))

i.e. Bob supplies Mary with everything she wants which is not supplied by John.

In general a negative condition

Not(P)

is judged to hold if the unnegated query

Does(P)

fails to hold, i.e. returns the answer "No". Keith Clark [7] has shown that negation by failure coincides with the classical notion of negation under the closed world assumption: that all the information needed to solve the problem P is present in the database.

Current PROLOG implementations of negation by failure cannot be used to generate answers, i.e. answer queries such as

Which (x Not(Wants (Mary x)).

This limitation is related to various restrictions on negation in relational database query languages.

Horn clauses as conditions

Sentences such as

Supplies (Mary Friendship x)
if For-all y [Supplies(x y Mary) if Wants(Mary y)]

i.e. Mary supplies friendship to anyone who supplies her with everything she wants

are a simple extension of Horn clauses, where a condition can itself have the form of a Horn clause. The PROLOG implementation of such a sentence gives it the obvious procedural interpretation:

```
to show Mary supplies friendship to x
        find all y such that Mary wants y and
        show that x supplies y to Mary.
```

Horn clauses as conditions occur commonly in both database queries and program specifications. For example:

```
Which (x Supplier(x) and For-all y[x supplies y if y part-of bicycle])

x is-a-subset-of y
            if For-all z [z belongs-to y if z belongs-to x]

Ordered(x)
        if For-all u v [u ≤ v if Consecutive(u v x)]
```

Here Consecutive(u v x) expresses that u and v are consecutive elements of the sequence x.

Horn clauses as conditions can be implemented in terms of negation by failure. Their current implementation suffers from the same restriction that they can only be used to test that given relationships hold.

Sets as individuals

In many applications it is necessary, not only to print out all answers to a query, but also to generate and manipulate a set of answers within the body of a program or query. The notation of set theory is well suited for this purpose: Goals of the form

```
y={x : P}
```

can be used to generate a list (without duplicates) of all x which satisfy the conditions P. For example,

```
  Which(z   y={x : John supplies x} and
            y has-cardinality z)
i.e. How many items does John supply?
```

Has-cardinality has the obvious recursive definition:

```
(|)     has-cardinality o
(x | y) has-cardinality u
        if y has cardinality v and
        u=v+1.
```

As another example,

```
x is-intersection-of (y z)
    if x={u : u belongs-to y and
              u belongs-to z}
```

defines the intersection of two sets.

The set constructor is a feature of many database query languages as well as the main feature of the programming language SETL [23]. It is also an important feature of several functional programming languages, most notably of Hope [53] and KRC [54]. An efficient implementation has been done by David Warren [52] for Dec10 PROLOG. A related facility has also been implemented in Micro-Prolog.

THE SCOPE FOR PARALLELISM

Or-parallelism is the easiest form of parallelism to implement. It includes as a special case the execution of both branches of a conditional in parallel. The conditional statement

```
to do A
    if P then do B
    else do C
```

becomes two sentences

A if P and B
A if Not(P) and C.

It also includes parallel or associative search in a database:

$P(t_1)$
$P(t_2)$
.
.
.
$P(t_n)$.

In addition to the PARALOG machine of Aida and Moto-Oka and the Fifth Generation Project more generally, machine architectures for or-parallelism are being investigated by Minker on ZMOB [47], Connery and Kibler [18] Pollard [44] and Darlington, Reeve and Gregory on ALICE [20].

And-parallelism in the most general case is more difficult. The special case where subgoals

B and C

share no variables is easiest to deal with. The two subproblems can be pursued independently and in parallel. A minor modification of or-parallelism is adequate to deal with this case.

Parallel reduction of problems to independent subproblems includes as a special case the parallel computation of the subterms of an expression in a functional programming language. For example, the subterms

g(r) h(s) and k(t)

of the expression

f(g(r) h(s) k(t))

can be computed in parallel. In logic programming, functions are represented by relations. In particular

g(x) =y can be represented by G(x y)
h(x) =y can be represented by H(x y)
k(x) =y can be represented by K(x y)
f(x1 x2 x3)=y can be represented by F(x1 x2 x3 y).

Expressions to be computed are represented as conjunctive queries

Which (y G(r y1) and H(s y2) and K(t y3) and F(y1 y2 y3 y))

Parallel computation of subterms becomes and-parallel execution of the subgoals

G(r y1) and H(s y2) and K(t y3)

which share no variables.

Another important case of and-parallelism is the one in which subgoals B and C share a variable which serves as a communication channel for a stream of values produced by B and consumed by C. Such streams can be represented by lists and the production of the stream by B can be executed in parallel with its consumption by C. Sequential coroutining implementation of streams is a feature of IC-PROLOG [9]. Parallel stream processing is the distinguishing feature of the Horn clause based relational programming language of Clark and Gregory [12].

Not-parallelism, All-parallelism and set-parallelism can be reduced to or-parallelism. To show Not(P), or-parallelism can be used to try the alternative ways of solving P in parallel. To show For-all x [B if C], or-parallelism can be used both to find all the alternative solutions of C and to test that all such solutions also solve B. To generate a set y={x : P}, or-parallelism can be used to find all solutions of P in parallel. Preliminary research suggests that the dataflow and reduction machine architectures which have been

developed for parallel execution of functional programming languages can also be adapted to provide at least limited forms of or-parallelism and and-parallelism for logic programming.

So far we have only considered execution strategies which use rules of the form

A if B and C

backwards to reduce problems A to subproblems B and C. It is also possible, and in certain circumstances desirable, to use such rules forward

given B and C
to derive A.

Many rule-based formalisms for expert systems use such forward reasoning as their only means of execution. It is attractive therefore to investigate implementations which combine forward and backward execution in parallel, especially for applications to expert systems.

CONCLUSION

We have argued that computer-executable logic is the missing link between the functional programming languages being investigated for the exploitation of highly parallel computer architectures and the rule-based languages being used for applications in expert systems. It also provides the link needed to unify nonprocedural specification languages and programming languages with query languages in database systems.

REFERENCES

[1] Bergman, M., Kanoui H., (1973), Application of Mechanical Theorem Proving to Symbolic Calculus. Third International Symposium on Advanced Computing Methods in Theoretical Physics, C.N.R.S., Marseilles. June 1973

[2] Belovari G. and Campbell J. A., (1980), Generating Contours of Integration: An Application of PROLOG in Symbolic Computing. In 5th Conference on Automated Deduction (Bibel W. and Kowalski R., Eds.) Springer-Verlag. Lecture Notes in Computer Science 87, pp. 14–23.

[3] Bundy A., et al (1979), MECHO: A Program to Solve Mechanics Problems. DAI Working Paper No. 50. Univ. of Edinburgh.

[4] Burstall R. M., Darlington J. (1977), Transformation for Developing Recursive Programs. J. ACM Vol. 24 No. 1. pp. 44–67.

[5] Clark K. L. and Sickel S. (1977), Predicate Logic: A Calculus for Deriving Programs. Proc. 5th Int. Joint Conf. on Artificial Intelligence. Cambridge, Mass.

[6] Clark K. L., Tarnlund S.-A., (1977), A First Order Theory of Data and Programs. Proceedings IFIP 77. North Holland, pp. 939–944.

[7] Clark, K. L., (1978), Negation as Failure. Logic and Data Bases. Plenum Press. New York, pp. 293–322.

[8] Clark K. L., Darlington J., (1978), Algorithm Classification through Synthesis. Computer Journal, pp. 61–65.

[9] Clark K. L., and McCabe F. G., (1979), Control Facilities of IC-PROLOG. *Expert Systems in the Micro-electronic age.* (D. Michie, Ed.). Edinburgh University Press.

[10] Clark K. L. and McCabe F. G., (1980), PROLOG: A Language for Implementing Expert Systems. In Machine Intelligence 10 (Hayes J. and Michie D., Eds.) Ellis and Horwood.

[11] Clark K. L., (1980), Logic as a Programming Calculus, to be published by Springer-Verlag.

[12] Clark K. L. and Gregory S., (1981), A Relational Language for Parallel Programming. In Functional Programming and Computer Architecture, ACM, New York.

[13] Clark K. L., Ennals J. R. and McCabe F. G., (1981), A Micro-PROLOG Primer. Logic Programming Associates Ltd., 36 Gorst Road, London SW11 6JE.

[14] Clocksin W. F. and Mellish C. S. (1981), Programming in Prolog. Springer-Verlag.

[15] Colmerauer A., Kanoui H., Pasero R., Roussel P., (1973), Un Systeme de Communication Homme-machine en Francais. Rapport, Groupe Intelligence Artificielle, Universite d'aix Marseille, Luminy.

[16] Colmerauer A., (1977), An Interesting Natural Language Subset. *Logic Programming* (K. Clark and S.-A. Tarnlund, Eds.). Academic Press. 1982.

[17] Colmerauer A. (1978), Metamorphosis Grammars. Natural Language Communication with Computers, (L. Bolc, Ed.), Lecture Notes in Computer Science No. 63, Springer-Verlag, Berlin, Heidelberg, New York. pp. 133–189.

[18] Conery J. S. and Kibler D. F., (1981), Parallel Interpretation of Logic Programs. In Functional Programming and Computer Architecture, ACM, New York.

[19] Dahl V., (1979), Quantification in a Three-valued Logic for Natural Language Question-answering Systems. Proc. 6th IJCAI, Tokyo.

[20] Darlington J. and Reeve M., (1981), ALICE: A Multi-processor Reduction Machine for the Parallel Evaluation of Applicative Languages. In Functional Programming and Computer Architecture. ACM, New York.

[21] Darvas F., Futo I., Szeredi P., (1978), The Application of PROLOG to the Development of QA and DBM Systems. Logic and Data Bases. Plenum Press, New York, pp. 347–375.

[22] Deliyanni A., Kowalski R. A., (1979), Logic and Semantic Networks. Comm. ACM. Vol. 22. No. 3, pp. 184–192.

[23] Dewar R. B. K., Grand A., Liu S-C., Schwartz J. and Schonberg E. (1979), Programming by Refinement, as exemplified by the SETL representation sublanguage, ACM Transactions on Programming Languages and Systems, Vol. 1. No. 1, pp. 27–49.

[24] van Emden M. H., and Kowalski R. A., (1976), The Semantics of Predicate Logic as a Programming Language. J. ACM, Vol. 23, No. 4, pp. 733–742.

[25] van Emden M. H. (1978), Computation and Deductive Information Retrieval. Formal Description of Programming Concepts, (E. Neuhold, Ed.), North Holland, pp. 421–440.

[26] Ennals J. R., (1981), Logic as a Computer Language for Children: A One Year Course. DoC. 81/6, Imperial College, London.

[27] Gallaire H., Minker J., (Editors), (1978), Logic and Data Bases. Plenum Press. New York.

[28] Gallaire H., (1981), The Impact of Logic on Database. Seventh International Conference on Very Large Data Bases. Cannes.

[29] Hayes P. J., (1973), Computation and Deduction. Proc. 2nd MFCS Symp. Czechoslovak Academy of Sciences, pp.105–118.

[30] Hayes P. J., (1977), In Defense of Logic. International Joint Conference on Artificial Intelligence, 5, pp. 559–565.

[31] Hogger C. J., (1978), Goal Oriented Derivation of Logic Programs. Proc. MFCS Conf., Polish Academy of Sciences, Zakopane.

[32] Hogger C. J., (1978), Program Synthesis in Predicate Logic. Proc. AISB/GI Conf. on AI. Hamburg. July, 18–20.

[33] Hogger C. J., (1980), Logic Representation of a Concurrent Algorithm. Imperial College, London.

[34] Hogger C. J., (1981), Derivation of Logic Programs, JACM, Vol. 28, No. 2, pp. 372–392.

[35] JIPDEC (1981), Proceedings of International Conference on Fifth Generation Computer Systems. Tokyo.

[36] Kowalski R. A., (1974), Predicate Logic as Programming Language. Proc. IFIP 74. North Holland Publishing Co., Amsterdam. pp. 569–574.

[37] Kowalski R. A., (1979), Algorithm=Logic+Control. CACM, August 1979.

[38] Kowalski R. A., (1979), Logic for Problem Solving. North Holland Elsevier. New York.

[39] Kowalski R. A., (1981), Prolog as a Logic Programming Language. Proceedings of AICA Congress. Pavia, Italy.

[40] McCabe F. G., (1980–81), Micro-PROLOG Programmer's Reference Manual. Logic Programming Associates Ltd., 36 Gorst Road, London SW11 6JE.

[41] McDermott D., (1980), The Prolog Phenomenon. SIGART Newsletter, No. 72, pp. 16–20.

[42] Markusz Z., (1977), How to design variants of flats using the programming language PROLOG based on mathematical logic. Proc. IFIP 77 pp. 885–889.

[43] Operating Systems, Inc. (1979), A Knowledge Based Automated Message Understanding Methodology for an Advanced Indications System. OSI R79-006, Woodland Hills, Ca.

[44] Pollard G. H., (1982), Parallel Execution of Horn Clause Programs. Ph.D. thesis. Imperial College, London.

[45] Reiter R., (1978), On Closed World Data Bases. Logic and Data Bases, (H. Gallaire and J. Minker, Eds.), Plenum Press, New York, pp. 55–76.

[46] Roussel P., (1975), PROLOG: Manuel de Reference et d'Utilisation. Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy.

[47] Rieger C., Trigg R. and Bane B., (1981), ZMOB: A New Computing Engine for AI. Maryland Artificial Intelligence Group, University of Maryland.

[48] Shapiro E. Y., (1981), An Algorithm That Infers Theories from Facts. In Proceedings of the Seventh International Joint Conference on Artificial Intelligence, Vancouver B.C., Canada. pp. 446–451.

[49] Shapiro, E. Y., (1981), The Model Inference System. (Program Demonstration in Proceedings of the Seventh International Joint Conference on Artificial Intelligence, Vancouver B.C., Canada. p. 1064.

[50] Warren D. H., (1976), Generating Conditional Plans and Programs. Proc. AISB Summer Conference, Edinburgh. pp. 344–354.

[51] Warren D. H., Pereira L. M., Pereira F., (1979), PROLOG – The Language and its Implementation Compared with LISP. Proc. Symp. on AI and Programming Languages, SIGPLAN Notices, Vol. 12, No. 8.

[52] Warren D., (1981), Higher-order Extensions to Prolog – Are they needed? To appear in "Machine Intelligence 10". Ellis and Horwood.

[53] Burstall R. M., MacQueen D. B. and Sanella D. T. (1980), HOPE: an experimental applicative language. Proc. LISP conference, Stanford, pp. 136–143.

[54] Turner D., (1981), The Semantic Elegance of Applicative Languages. In Functional Programming and Computer Architecture. ACM, New York.