HYPOTHESIS DRIVEN PROGRAMMING

Ross G. Hopmans
Brant Computer Services Limited
2605 Skymark Avenue
Mississauga, Ontario
CANADA L4W 4L5
(416) 238-9790

INTRODUCTION

In this paper I will introduce the basics of Artificial Intelligence, Knowledge-Based systems, Logic Programming and the Prolog language to establish the groundwork for, and a means to implement goal-oriented or Hypothesis Driven programming. Through the use of examples I hope to show the mechanics and benefits of this technology.

ARTIFICIAL INTELLIGENCE

The term ARTIFICIAL INTELLIGENCE is generally used to describe the technology for dealing with KNOWLEDGE (a superset of DATA).

Artificial Intelligence is concerned with making machines more useful by causing them to mimic processes that, if performed by humans, would be considered intelligent. Many countries have significant, national-level Advanced Information Technology programs underway. This approach, called "technology push", focuses on the development of the technology for yet unspecified commercial applications. In contrast, most of the successfully implemented AI applications have resulted from demand or "market pull" for new solutions to existing problems.

AI is concerned with the study of systems that represent, acquire and use knowledge in order to perceive, reason, plan, act and use language. Some differences between biological and artificial intelligence are as follows:

Biological Intelligence	*Artificial Intelligence*
perishable	*permanent*
difficult to transfer	*easy to duplicate*
erratic	*consistent*
difficult to reproduce	*easy to document*
creative	*uninspired*
learned	*programmed*
sensory input	*symbolic input*
wide context	*narrow focus*

KNOWLEDGE-BASED AND EXPERT SYSTEMS

Concentrating on the simulation of human expertise has produced the most successful AI programs in the form of expert systems that help make decisions by exploiting scarce or expensive expertise. They require a lot of high quality, specific knowledge about a particular topic.

Some characteristic categories are:

Interpretation	*inferring situation descriptions from sensor data*
Prediction	*inferring likely consequences of given situations*
Diagnosis	*inferring system malfunction from observations*
Monitoring	*comparing observations to expected outcomes*
Advisor	*expressing opinions after drawing from relevant facts*

An expert system can make deductions through inference, motivate its own actions and solutions, suggest alternatives to solutions and handle uncertain information.

The terms "Knowledge-based systems" and "expert systems" are often used interchangeably, but their meanings are quite different. The term knowledge-based systems describes an important technical issue: it indicates that the source of the program's power is primarily a large body of task-specific knowledge. Such systems may function in a variety of roles acting as assistants, colleagues and sometimes as experts.

The term expert systems refers primarily to an aspiration - the desire to have a system that works as well as a human expert.

Most of the interesting things we know about the world are not numeric and most of the knowledge we have about the world is not well modelled with arithmetic. We think and reason about problems using IF/THEN rules, but how do we capture these type of rules using arithmetic?

Knowledge-Based systems take a particular view of the answer to the question "Why are experts experts?" Do they think faster than the rest of us? Do they think differently? Do they have a general purpose trick of thinking or problem solving?

To call something a knowledge-based system subscribes to the belief that experts are experts because of what they know. Expertise arises from knowledge. The power of knowledge-based systems comes from their base of knowledge about the specific task at hand.

A Knowledge-Based Expert System is a close approximation to cloning. What we can not do biologically, we try to do intellectually. We take take rare and valuable expertise and try to clone it by talking at length with the person who has it. It is useful because almost all organizations have a knowledge bottleneck - a place where its productivity is limited by a scarcity of knowledge and skill.

Who do you miss when they are sick? Who would you hate to lose to the competition? Whose impending retirement has you worried? Who do you wish you had five more of?

Cloning that knowledge is one possible remedy to these problems and knowledge-based systems offer us the possibility of doing that. The focus of knowledge-based systems is on *knowledge* - collecting it, formalizing it and putting it to work in the computer.

WHY USE KNOWLEDGE-BASED SYSTEMS?

The oldest expert system in commercial use configures computer systems. But that type of problem is not unique to the electronics or manufacturing industries. Contracts built from standard clauses and insurance policies built from standard coverage also need to be "assembled". In all cases we want to ensure that all the needed parts are there, nothing is omitted, nothing extra is added and they all function together smoothly. So this technology can span across industry sectors.

There is also an issue of corporate memory. We would like to retain human skill and have it outlive any single individual. Using knowledge-based technology we can debrief the experts who have a lifetime of experience and capture a useful part of what they know - making more real the idea of corporate memory.

If the line of responsibility in a knowledge-based system falls mainly to the human, then the program is more of an assistant. If we can push that line of responsibility more to the machine end and make the program smarter then that may be something we can call an expert system. Useful applications can fall anywhere along that line. We can take an evolutionary approach and start a program out as an assistant and add more knowledge over time to give the machine more responsibility.

HOW DO KNOWLEDGE-BASED SYSTEMS WORK?

Knowledge-based systems are characterized by their reliance on large stores of rule-based knowledge as a basis of their expertise. In effect, we have done a dicing of the human expert's knowledge into individual rules. Rules are comprehensible and each stands on its own feet. Each rule makes one small decision that relays one thing to do under a particular set of circumstances. We use a rule whenever it is relevant - that is whenever its pre-conditions (the items in the IF part of the rule) have been met.

In traditional programming we write complete, decision-making procedures, whereas in AI we write individual decision rules. Traditional programming tells the computer what to do; here we tell it what to know. It turns out that a few hundred to a few thousand rules are adequate for capturing interesting, valuable and non-trivial skills.

A big difference between an expert system and a conventional system is that knowledge is stored in a knowledge base in the expert system but coded into a program in the conventional approach. Program code is a very primitive way of storing knowledge. It is rigid, static and inaccessible.

The art of building knowledge systems is one of collecting and accumulating facts and rules. This collection comprises the knowledge base. The other component is the inference engine - that part of the program that uses the rules of the knowledge-based system and applies them to the problem at hand. The engine's primary job is to do symbolic inference. The separation of knowledge base and inference engine is characteristic of knowledge systems and but is atypical of traditional systems.

Knowledge-based systems can generate an explanation of their behaviour and the answer is comprehensible because the computer is doing symbolic inference and not arithmetic; reasoning and not calculation. We have turned the computer into a logical reasoning engine - a device whose basic operation is rule retrieval and application. Traditional applications are not, nor were they intended to be models of human problem solving. Instead, they are powerful and have an important place but they are not necessarily comprehensible.

As a result, knowledge-based system have a property we call transparency. When you look inside them,

their operation makes sense to us. They are not black boxes. They reveal their line of reasoning and it is sensible.

Eventually a knowledge-based system reaches a conclusion with a list of plausible alternatives for decision support. It does not just pick out one answer it thinks is best. Transparency allows us to ask how the program decided what is plausible and it allows us to examine the reasoning chain because it keeps an audit trail of the logic it uses and allows us to examine that audit trail in as much detail as we like.

LOGIC PROGRAMMING

The study of symbolic logic goes back to the work of Aristotle in the fourth century B.C. First order predicate logic is a branch of symbolic logic that has evolved largely in the twentieth century. It is a universal, abstract language for representing knowledge and solving problems. Logic programming is based on a subset of first order predicate logic.

Aristotle attempted to codify into a scientific system the way that knowledge could be most effectively pursued through rational debate. He wanted to establish a standard whereby the correctness of a line of reasoning could be established.

For instance, given the following two premises:

Socrates is Human.
Humans are Mortal.

then the following conclusion is valid:

Socrates is Mortal.

The process of reaching a conclusion from the premises is also called making an inference. The above inference takes place because the predicate class of one statement matches the subject class of another. As a result of the inference, a new statement is formed out of the subject of one statement and the predicate of the other.

Logic provides a precise language for the expression of one's goals (or hypotheses), knowledge and assumptions. Logic provides the foundation for deducing consequences from premises, for studying the truth or falsity of statements given the truth or falsity of other statements, for establishing the consistency of one's claims and for verifying the validity of one's arguments.

Through Logic Programming we can *prove* program correctness!

The above example can be proven correct through the simplest and most powerful of Aristotle's inference rules known as *modus ponens* which states:

If P implies Q and P is true, then Q is true.

Restating our example in this format, we have a rule P implies Q, and a fact P as:

If a person is human then that person is mortal.
Socrates is human.

or, more programatically familiar:

If human(X) then mortal(X).
human(Socrates).

We can conclude that *Socrates is Mortal.*

In this example we applied a fact to a simple, comprehensible, general-purpose rule and we were able to infer a new piece of information which we can prove to be correct.

Although computers were intended for use by humans, the language for expressing problems to the computer and instructing it how to solve them was designed from the perspective of the engineering of the computer alone.

Logic programming departs radically from the mainstream of computer languages. Rather than being derived from the von Neumann machine model, it is derived from an abstract model, which has no direct relationship or dependency to one machine model or another. It is based on the belief that instead of the human learning to think in terms of the operations of a computer, which some scientists and engineers at some point in history happened to find easy and cost-efficient to build, the computer should perform instructions that are easy for humans to provide. Logic programming suggests that explicit instructions for operation not be given but rather the knowledge about the problem and assumptions that are sufficient to solve it be stated explicitly. This constitutes an alternative to the conventional program. The logic program can be executed by providing it with a hypothesis (or problem), formalized as a logical statement to be proven, called a goal statement. The execution is an attempt to solve the problem - that is, to prove the hypothesis given the assumptions in the logic program.

A major aim of logic programming is to enable the programmer to program at a higher level. Ideally one should write axioms that define the desired relationships, maintaining ignorance of the way they are going to be used by the execution mechanism. Current logic programming languages such as Prolog, however cannot ignore how their execution mechanisms work. Effective logic programming requires a certain implementation knowledge for effective execution.

PROLOG

Fifth generation languages are valuable because they help reduce the distance between the verbal description of a process and its representation in executable code. The resulting programs are thus easier to both read and change. The logical foundations of a program tend to be closer to the surface in fifth generation languages than in more conventional ones.

The evolution of computer languages is an evolution away from low-level languages, in which the programmer specifies how something is to be done, toward high-level languages, in which the programmer specifies simply what is to be done. Most languages, Lisp included, are "how-to" languages. Prolog breaks away from that, encouraging the programmer to describe situations and problems, not the detailed means by which the problems are to be solved.

Prolog is a programming language centered around a small set of basic mechanisms, including pattern matching, tree-based data structuring, and automatic backtracking. This small set constitutes a surprisingly powerful and flexible programming framework.

A Prolog program consists of a set of rules for deducing the truth of a given hypothesis (or goal) from a conjunction of other hypotheses. A set of rules with the same head represents alternative ways of establishing the same hypothesis. Such a set is sometimes called a procedure. Prolog procedures are somewhat analogous to procedures in mainstream languages and are conceptually like the *Case* statement. The goals in the body of a procedure are invoked sequentially, and a procedure is exited when all hypotheses in one of its rules have succeeded.

However, a fundamental difference in the flow of control arises when one of the hypotheses fails. Whereas in mainstream languages failed statements must be handled explicitly, Prolog automatically backs up to the previous hypothesis, attempting to satisfy it in a different way. This behaviour is known as backtracking.

Programming in Prolog often consists of merely describing essential properties of a problem, defining the relations and rules applicable to the problem and stating known facts relevant to the problem. With conventional programming languages, the programmer has to spell out a detailed sequence of steps which the computer must perform. The declarative style of programming made possible by Prolog is faster, easier and less error-prone.

Prolog's symbolic nature, dynamic memory management and flexible structure makes it an ideal language for the rapid prototyping of virtually any kind of system but Prolog is especially well suited for problems that involve objects - in particular, structured objects - and relations between them.

The control structure in Prolog is unification, or a pattern matching process operating on a sophisticated internal data base which contains a full relational database. Data and programs are stored in this internal database which in searched for solutions to goals. The Prolog system can search for alternative solutions (called non-determinism) by backtracking through the search space. Prolog programs in the database consist of statements which express relationships between entities. The "logical engine" in the Prolog system then runs the program by inferring true statements from the given relationships.

Unlike conventional languages, Prolog incorporates program control into the language itself. The programmer is relieved of the majority of the control of program flow and can focus on expressing data objects' relationships.

Prolog is not "just another language"; it may by THE language of the future. Prolog is based on logic, and not on a mapping of the machine architecture. Prolog provides a *Procedural Interpretation* which defines how the problem should be solved with an algorithm and a *Declarative Interpretation* which describes the problems and what is known. It is the Declarative Interpretation allows us to prove programs correct.

There is currently no universal Prolog standard. Over time, the implementation described by W.F. Clocksin and C.S. Mellish in *Programming in Prolog* (Springer-Verlag, 1985) has emerged as the de facto standard.

HOW PROLOG WORKS

The control in Prolog programs is like in conventional procedural languages as long as the computation progresses forward. Hypothesis invocation corresponds to procedure invocation, and the ordering of hypotheses in the body of clauses corresponds to sequencing of statements. The differences show when backtracking occurs. In a conventional language, if a computation cannot proceed (eg. all branches of a case statement are false) a runtime error occurs. In Prolog, the computation is simply undone to the last choice made, and a different computation path is attempted.

The data structures manipulated by logic programs (terms) correspond to general record structures in conventional programming languages. The handling of data strucures is very flexible in Prolog. Like LISP, Prolog is a declaration free, typeless language. Logical variables refer to individuals rather than memory locations. Consequently, having specified a particular individual, the variable cannot be made to refer to another individual. In other words, logic programming does not support destructive assignment where the contents of an initialized variable can change.

A question to Prolog is always a sequence of one or more hypotheses. To answer a question, Prolog tries to satisfy all the hypotheses. To satisfy a hypothesis means to demonstrate that the hypothesis is true, assuming that the relations in the program are true. In other words, to satisfy a hypothesis means to demonstrate that the hypothesis logically follows from the facts and rules in the program. If the question contains variables, Prolog also has to find what are the particular objects (in place of the variables) for which the hypotheses are satisfied. If Prolog cannot demonstrate for some instantiation of variables that the hypotheses logically follow from the program, then Prolog's answer to the question will be 'no'.

Prolog accepts facts and rules as a set of axioms and the user's question as a conjectured theorem; then it tries to prove this theorem - that is to demonstrate that it can be logically derived from the axioms.

The graphical illustration of an execution trace has the form of a tree. The nodes correspond to hypotheses and the arcs to the application of alternative program clauses that transform the hypotheses at one node into hypotheses at another node. The top hypothesis is satisfied when a path is found from the root node to a leaf node labelled "yes". A leaf is labelled "yes" if it is a simple fact. The execution of Prolog programs is the searching for such paths. During the search Prolog may enter an unsuccessful branch. When Prolog discovers that a branch fails it automatically backtracks to the previous node and tries to apply an alternative clause at that node.

DECLARATIVE VS. PROCEDURAL

We distinguish between two levels of meaning of Prolog programs; namely the declarative meaning and the procedural meaning.

The procedural meaning is concerned only with the relations defined by the program. The declarative meaning thus determines what will be the output of the program. On the other hand, the procedural meaning also determines how this output is obtained; that is, how are the relations actually evaluated by the Prolog system.

Consider the clause P :- Q,R. Some alternative declarative readings of this clause are:

P is true is Q and R are true.
From Q and R follows P.

Two alternative procedural readings of this clause are:

To solve problem P, first solve subproblem Q, then subproblem R
To satisfy R, first satisfy Q and then R

The procedural reading do not only define the logical relations between the head of the clause and the hypotheses in the body, but also the order in which the hypotheses are to be processed.

The declarative meaning of programs determines whether a given hypothesis is true, and if so, for what values of variables is it true.

The procedural meaning specifies how Prolog answers questions.

P :- P is declaratively quite correct but procedurally useless and can cause problems to Prolog as it results in a infinite loop. What is unusual about Prolog is that the declarative meaning of a program may be correct, but the program is at the same time procedurally incorrect in that it is unable to produce an answer to a question.

A general practical heuristic in problem solving is that is it often useful to try the simplest idea first.

The reason we should not forget about the declarative meaning is that progress in programming technology is achieved by moving away from procedural details toward declarative aspects, which are normally easier to formulate and understand. The system itself, not the programmer, should carry the burden of filling in the procedural details.

PROGRAMMING IN PROLOG

Prolog is related to mathematical logic, so its syntax and meaning can be specified most concisely with references to logic. But these concepts are not necessary for understanding and using Prolog as a programming tool.

Often the key step toward a solution is to generalize the problem. By considering a more general problem, the solution may become easier to formulate.

Prolog programs consist almost exclusively of declarations. The programmer is able to leave many of the control decisions to the problem solving mechanism. SQL has a similar declarative nature in that the user specifies what type of answer is required and the interpreter decides how to supply it.

Consider the following example Prolog program to to describe why a product benefits a customer. The program consists of 14 statements: 3 rules and 11 facts.

```
PRODUCT benefits CUSTOMER if
        CUSTOMER is-involved-with BUSINESS and
        PRODUCT improves BUSINESS.
PRODUCT benefits CUSTOMER is
        CUSTOMER employs PROFESSIONAL and
        PRODUCT increases-productivity-of PROFESSIONAL.

PRODUCT improves BUSINESS if
        BUSINESS requires TOOL and
        PRODUCT enhances TOOL.

hewlett-packard is-involved-with computers.
westinghouse is-involved-with power-plants.
metropolitan-life is-involved-with insurance-policies.

power-plants requires regulatory-analysis.
insurance-policies requires insurance-underwriting.

prolog enhances expert-systesms.
prolog enhances natural-language-processing.
prolog enhances insurance-underwriting.
prolog enhances regulatory-analysis.

hewlett-packard employes software-engineers.

prolog increases-productivity-of software-engineers.
```

Most Prolog implementations support the "syntactic sugar" used here to enhance the readability of the program. The equivalent, standard Prolog code is as follows:

```
benefits(PRODUCT,CUSTOMER) if
        is-involved-with(CUSTOMER,BUSINESS) and
        improves(PRODUCT,BUSINESS).
benefits(PRODUCT,CUSTOMER) if
        employs(CUSTOMER,PROFESSIONAL) and
        increases-productivity-of(PRODUCT,PROFESSIONAL).

improves(PRODUCT,BUSINESS) if
        requires(BUSINESS,TOOL) and
        enhances(PRODUCT,TOOL).

is-involved-with(hewlett-packard,computers).
is-involved-with(westinghouse,power-plants).
is-involved-with(metropolitan-life,insurance-policies).

requires(power-plants,regulatory-analysis).
requires(insurance-policies,insurance-underwriting).

enhances(prolog,expert-systems).
enhances(prolog,natural-language-processing).
enhances(prolog,insurance-underwriting).
enhances(prolog,regulatory-analysis).

employs(hewlett-packard,software-engineers).

increases-productivity-of(prolog,software-engineers).
```

The words in upper case are Prolog variables. Variables are local to a statement so that each rule can be understood in isolation.

In this program, "benefits" and "improves" are rules whereas "is-involved-with", "requires", "enhances", "employs" and "increases-productivity-of" are all facts. Prolog, however, does not differentiate between facts and rules. In other words, we could quite correctly add a new fact:

benefits(prolog,brant).

This Prolog program contains a knowledge base of facts and rules which we can use in many different ways. To execute any portion of the program, we give it a hypothesis or goal. The following hypotheses succeed simply because they exist in the fact base:

enhances(prolog,expert-systems).
is-involved-with(hewlett-packard,computers).
employs(hewlett-packard,software-engineers).

Similarly, the following hypotheses fail because they cannot be proven correct:

enhances(prolog,data-processing).
employees(brant,software-engineers).

Using the rules, we can hypothesize about a particular solution, such as:

(1) benefits(prolog,hewlett-packard)

or our goal may be to find a solution, such as:

(2) benefits(prolog,CUSTOMER).

In the case (1), the hypothesis would succeed because the second "benefits" rule can be successfully applied to the facts provided. In case (2), the hypothesis would succeed, returning the first value of the variable CUSTOMER for which it did succeed. In this case, the value returned would be "westinghouse".

Through its backtracking capability, Prolog is able to return all the values of variables for which the hypothesis succeeds. For example,

benefits(prolog,CUSTOMER), fail.

would locate all instances of CUSTOMER for which the hypothesis is true. By explicitly telling the hypothesis to fail after successfully finding a value for CUSTOMER, the inference engine will attempt to solve for the hypothesis in all possible ways.

In fact, using:

benefits(PRODUCT,CUSTOMER), fail.

Prolog will find all PRODUCT/CUSTOMER touples for which the hypothesis is true.

This small, simple example program begins to illustrate the power of this type of programming. The programmer tells Prolog what he knows. He provides the program with facts and rules and he can then hypothesize about anything in the knowledge base.

It is the inference engine and not the programmer that takes on the burden of solving the problem by doing the symbolic inference, pattern matching and tree searching.

CONCLUSION

As powerful as the numeric tools are, there is relatively little we know that can be reasonable expressed as numbers. But the computer is not just a calculator; it is a general purpose symbol manipulator.

The beauty of this technology so far is that is seems appropriate to all industries. The leaders have been in finance, manufacturing and insurance but everyone is participating.

Hypothesis driven, or goal-oriented programming is a means of implementing logic programming using the Prolog language. The whole purpose is to program at a higher level.

Because you tell the computer what you know about the problem rather than how to solve the problem, you can achieve vast improvements in productivity. Programs make sense and can be proven correct so they take less time to write, test and debug.

Most importantly, you are capturing the **knowledge** about the problem. *In the Knowledge Lies the Power.* The programs work because of the base of knowledge about the case at hand.

Maybe the real computer revolution is yet to come - to use the computer to do reasoning. That is what AI and the much talked about 5th generation is all about.

APPENDIX A

THE HISTORY OF PROLOG

Prolog's the two founders Robert Kowalski (Edinburgh) and Alain Colmerauer (Marseille) worked on similar ideas during the early 70's and even worked together during one summer. The results were the formulation of the logic programming philosophy and computation model by Robert Kowalski (1974), and the design and implementation of the first logic programming language, Prolog, by Alain Collmerauer and his colleagues (1973).

A more potent force behind the realization that logic can be the basis of a practical programming language has been the development of efficient implementation techniques, as pioneered by Warren (1977) with the Prolog-10 compiler.

In spite of all the theoretical work and exciting ideas, the logic programming approach seemed unrealistic. The main claim was that the languages such as Prolog were hopelessly inefficient and difficult to control and could not prove a substitute for Lisp. In the mid to late 70's the Prolog-10 compiler was developed and dispelled all the myths about the impracticality of logic programming. That compiler delivered performance comparable to the best Lisp systems available at the time. Furthermore, the compiler itself was written almost entirely in Prolog, suggesting that classical programming tasks, not just sophisticated AI applications, can benefit from the power of logic programming.

No doubt logic programming would have remained a fringe activity in computer science for quite a while longer were it not for the announcement of the Japanese Fifth Generation Project in October, 1981.

The syntax of Prolog stems from the clausal form of logic due to Kowalski (1974). Warren adapted Marseille Prolog for DEC-10 and many systems adopted most of the conventions of Prolog-10 which has become known more generically as Edinburgh Prolog. Its essential features are described in the widespread primer on Prolog (Clocksin and Mellish, 1984).

APPENDIX B

KNOWLEDGE ACQUISITION

We train people to have expertness but not expertise. We call people such as lawyers and engineers para-professionals - those who master sequences of activity but lack the generative capability for creating those sequences that true experts have. In a knowledge-based system we have to be certain that we are getting the expertise underlying the performance.

Experts guess a lot. They are effective at estimating things in ways that less expert people cannot begin to comprehend. They work at hard problems, get things right most of the time and do that quickly.

But expertise is not just lots of experience. It involves the use that experience to recognize key issues and ignore irrelevant ones. Human experts use *macros* that appear to be guesses. They cannot give IF/THEN rules or cause and effect rules directly. We expect them to reason intelligently rather than intuitively. The process tends to be invisible to the expert and they lack self knowledge. This is known as the Knowledge Engineering Paradox.

Human beings have developed the capacity to recognize and store away the things we do often to long term memory, like compiled programs, so we can run them in short term memory with far less expenditure of resources. The problem is that we no longer know what they contain.

Experts look for the shortest distance from problem to solution. Past experiences provide the shortest path. If not, novel problems can be a good technique to get at the underlying principles analytically using deductive steps.

Heuristics are used to avoid "garden path" results. Experts have *meta-rules* that say "be careful at this point".

Experts can gain a greater awareness of their knowledge by working with a Knowledge Engineer. Knowledge Engineering can create new knowledge by illuminating tasks where experts made mistakes or overlooked important things.

BIBLIOGRAPHY

Baxter, Lewis D. Computer Programming Management. *MPROLOG*. 1986.

Bratko, Ivan. *Prolog Programming for Artificial Intelligence.* Menlo Park, California: Addison-Wesley, 1987.

Engle, Steven; Hodgson, Jonathan; and Vits, James. Death by 1,000 cuts: Prolog on the PC. *Computer Language*, July 1987.

Malpas, John. *Prolog: a relational language and its applications.* Englewood Cliffs, New Jersey: Prentice-Hall, 1987.

Mathews, M. Haytham. Prolog and C. *Computer Language*. July 1987.

Sterling, Leon, and Shapiro, Ehud. *The Art of Prolog.* Cambridge, Massachusetts: The MIT Press, 1897.

Stevens, Lawrence. *Artificial Intelligence, the search for the perfect machine.* Hasbrouck Heights, New Jersey: Hayden, 1985.