Programming in Logic

Would you be interested in a class of programming languages that have no assignment statements, no control constructs and no types? Unlikely! Yet logic programming languages provides just such an environment and they are powerful tools for solving a variety of programming problems. This article describes the abstract model implicit in logic programming and compares it to the models used in conventional languages. A simple example in Prolog illustrates how programming in logic can provide a concise problem description while offering a very flexible framework to solve for unknowns. The last section of this article surveys the current fields of active research in the area of logic programming and concludes with a prognosis for the future of this method.

The Logic Programming Model

What distinguishes one class of programming languages from another is the programming model used to define a computation. The programming model for *imperative* programming languages like FORTRAN, Ada, Pascal and C is dictated by the architecture of a von Neumann computer. In that model the machine is assumed to be in a certain state that is modified by a sequence of instructions in a given order. This model also assumes that there is a fundamental distinction between data and program code: a variable is the name of a cell holding changing data elements and the code is a *procedural* sequence of actions to be performed according to control constructs (such as `while` and `if-then-else`).

Another model for a programming language is the one offered by *functional* languages like LISP, ML and Miranda. For some classes of applications, functional languages have several advantages over imperative languages: they provide a greater degree of abstraction, they make it possible to create self-transforming programs, and they are—theoretically anyway—independent of sequential "von Neumann" computer architectures and hence can exploit parallel architectures more easily. Functional languages that are *pure* (that do not have side-effects), are also *declarative* because they allow programmers to specify a description of the problem (*what* is computed) rather than itemize the sequence of execution steps required to solve the problem (*how* to compute it) (see Box 1). Programs written in a declarative language also lend themselves better to formal analysis, verification and transformation than programs written in an imperative language.

1

Just as functional programming is based on the mathematical theory of recursive functions, the model for logic programming languages such as Prolog (*Pro*gramming in *Log*ic) and Nail! (*N*ot *a*nother *i*mplementation of *lo*gic!) is first order symbolic logic. Since mathematical logic and the theory of recursive functions share common theoretical foundations, the programming languages based on them also share many characteristics. For example both logic programming languages and functional languages eliminate the usual distinction between code and data, and dispense entirely with variables (in the way imperative languages use them). Variables in logic programming languages, like variables in pure functional languages, are bound to the same value everywhere that they occur in a given expression: a logic variable either has a precise value or it is unknown (to a certain degree). This characteristic of variables is referred to as *referential transparency* and is important for proving the correctness of programs.

Perhaps the most important *difference* between imperative and logic programming languages is that computations in logic are *non-deterministic*, that is to say they can return *several* values. A logic programming language like Prolog, for example, can search a solution space for alternative solutions by *backtracking,* a process which undoes some portion of a computation and attempts to find other solutions by binding different values to variables. This non-determinism reflects the fact that first order logic describes the properties of *relations* between data objects which may have more than one set of values for a solution.

Unlike imperative languages, logic programming languages permit a high degree of abstraction, have promise for parallel architectures and make it easier to write declarative specifications. The promise of declarative languages is that they free programmers from the need to explicitly encode every task the computer must perform to solve a problem. Instead, explicit coding is replaced by a task better suited to human thinking, that of declaring one's thoughts in the language of logic.

Prolog

The most widespread logic programming language is Prolog. Since its inception in Marseilles and Edinburgh in the early 1970's, the research and development of Prolog has taken place mostly in European universities and research labs. It is sometimes said that Prolog was the upshot of research done in automated theorem proving, but it is more accurate to say that its roots lie in natural language processing and machine translation.

In the early 80's Prolog received an initial boost in popularity when it was announced that Japan's 5th generation project was adopting Prolog as the

language of choice for its software development. The justification behind this choice was that future generation computing would be focussed primarily on knowledge and database transactions and that Prolog with its inferencing capability would be an ideal programming language for manipulating knowledge. Moreover, Prolog showed promise as a language amenable to parallel hardware architectures.

Prolog is often mentioned in the same breath with LISP because both have been used in AI research such as natural language processing and search-intensive problems. Although in some respects these languages are similar, Prolog has been slow to catch on in North America primarily because of the influence of major educational institutions (MIT, Stanford, Carnegie Mellon) who, in the late 1970s and early 1980s strongly promoted the LISP language and dedicated LISP machines. For a while there was some rivalry between the LISP and Prolog communities about which language was best for which purposes, but the consensus now is that they are both typically good for writing expert systems and knowledge based systems. Prolog is more commonly used to handle problems involving natural language, automated reasoning, or searching, whereas LISP is preferred for work in knowledge representation and general purpose artificial intelligence because of the large number of tools now available.

But Prolog is not just useful for symbolic processing: it is a general purpose programming language. Since it is high-level, however, there are overhead costs in both memory consumption and speed (a factor of about 10 in the worst case) that make it less attractive for real-time software. Typically, its domain of use has been restricted to applications where overhead costs are negligible in comparison with processing or development costs. For example Prolog has been used in a number of applications in the areas of natural language processing, compiler writing, expert systems knowledge acquisition, and software prototyping.

Programming in Prolog

The basic structure in prolog is a *term*. A compound or structured term in Prolog consists of the name of a relationship, followed in parentheses by a sequence of its components, separated by commas. For example the Prolog term

```
book(joyce, 'Finnegans Wake', 1939)
```

is the equivelent in Prolog of what could be represented in Pascal as a record:

```
type
  Book = record
      Author: string[80];
      Title: string[80];
      Date: integer
```

```
    end;
var
   fw: Book;
begin
   fw.Author := 'joyce';
   fw.Title := 'Finnegans Wake';
   fw.Date := 1939;
end;
```
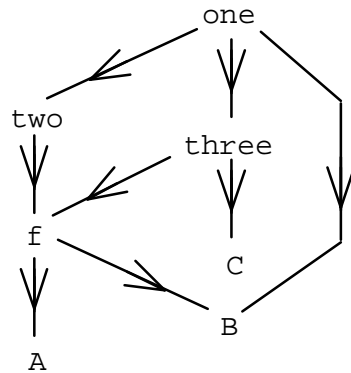
Terms can be arbitrarily complex and sub-components of terms may be shared by means of *logic variables* (sequences of characters begining with upper-case letters or an underscore) . Thus the term

```
        one( two( f( A, B ) ), three( f( A, B ), C ), B )
```

can be thought of as the directed acyclic graph:

where A, B and C are unbound variables whose values are undetermined. Several implementations of Prolog, like BNR Prolog, Prolog-II and SICStus Prolog also allow the creation of cyclic trees. For example the result of the unification

```
        X=f(X)
```

is the tree

Although this kind of term is not possible to construct in some implementations of Prolog, it is sometimes useful for representing recursive data structures.

A Prolog program is written as a set of rules defining the relationships between data items. For example, the following expression is a rule:

```
father(X) :- parent(X), male(X).
```

which is read 'X is a father' is true if 'X is parent' and 'X is male' are true. The logic variable X behaves exactly like the familiar variables of mathematics: if X takes on a value anywhere in the expression, it has the same value everywhere else. Variables take on values by a process called *unification* which occurs when, for example, the conditions of a rule are pattern-matched with other rules or simple facts.

Facts are simply rules that have no conditions. For example:

```
parent(john).
```

```
male(john).
```

Computations in Prolog are initiated by asking questions such as "?-father(X)" which attempts to bind a value to X. The process of finding a suitable value for X involves seeking a fact or rule which defines the conditions under which parent(X) is true and male(X) is true. In this case, the question would bind X to 'john'. Notice that if there were some additional facts such as:

```
parent(husband_of(hellen)).
```

```
male(husband_of(hellen)).
```

the question would yield a second solution (X=husband_of(hellen)).

Prolog also makes it possible to operate on sequence of elements by grouping them into lists. For example, the list [socrates, plato, aristotle] can be treated as a single data item. Lists can also be decomposed by using unification to pattern match for elements in a list. Thus unifying [A | B] with [socrates, plato, aristotle] binds A to socrates and B to the list [plato, aristotle]. Similarly, unifying [A, B | C] with [socrates, plato, aristotle] binds A to socrates, B to plato and C to the list [aristotle]. Lists can be examined in this manner by recursive procedures that operate on selected elements, as we shall see in the example below.

Example

To illustrate the kind of power and simplicity of a logic language like Prolog consider the following simple Prolog program that analyses the behaviour of a 5 NAND gate circuit.

First, let's define the facts governing the behaviour of a 2 input NAND gate under nofault or fault conditions. The input and output parameters have value 'off' or 'on' and the fault parameters have the value 'fault' or 'nofault'. A possible set of nand gate states are:/* no fault states: */
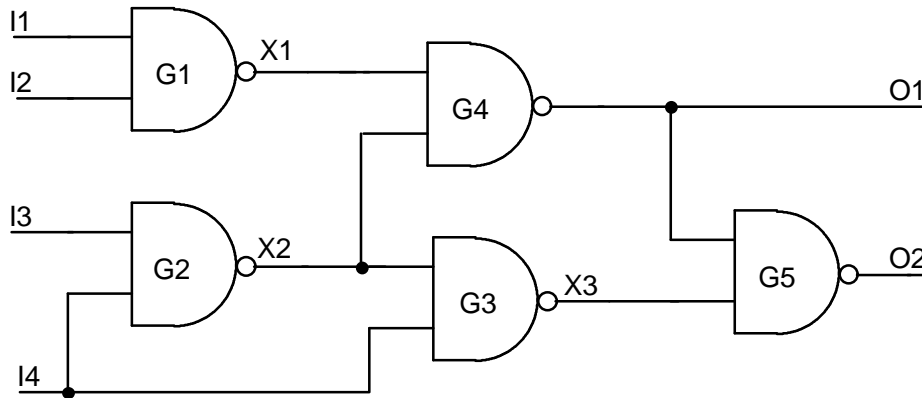
nand([on,on],off,nofault).

nand([on,off],on,nofault).

5

```
nand([off,on],on,nofault).
nand([off,off],on,nofault).
/* fault states:      */
nand([on,on],on,fault).
nand([on,off],off,fault).
nand([off,on],off,fault).
nand([off,off],off,fault).
```

Now let's define a circuit composed of 5 nand gates. (See diagram).

The circuit is defined as a relation between a list of inputs, `[I1,I2,I3,I4]`, a list of outputs, `[O1,O2]`, and a list of fault conditions (one per gate), `[F1,F2,F3,F4,F5]`. The interconnections between gates in a circuit are specified using logic variables. The circuit in the diagram is defined by:

```
circuit([I1,I2,I3,I4],[O1,O2],[F1,F2,F3,F4,F5]) :-
     nand([I1,I2],X1,F1),
     nand([I3,I4],X2,F2),
     nand([X2,I4],X3,F3),
     nand([X1,X2],O1,F4),
     nand([O1,X3],O2,F5).
```

We now have a complete specification of the simple 5 nand gate circuit. What kinds of questions can we ask about it?.

1. Generate truth table for no fault conditions.

```
?- circuit(Ins,Outs,[nofault,nofault,nofault,nofault,nofault]).
```

2. What outputs are possible if first and third inputs are off?

```
?- circuit([off,_,off,_],Outs,_).
```

6

(Note that a variable whose value we are not interested in can be ignored by using an underscore '_' as a place-holder.)

3. What is the truth table if only gate 2 is faulty?

```
?- circuit(Ins,Outs,[nofault, fault, nofault, nofault, nofault]).
```

and there are lots more.

We can enhance this program with the predicate `count_faults`, which relates an arbitrary length list of faults and nofaults to the number of faults in it. It is recursively defined by the following set of three rules:

```
count_faults([],0).
count_faults([fault|Ps],N) :-
     count_faults(Ps,N1),
     N is N1 + 1.
count_faults([nofault|Ps],N) :- count_faults(Ps,N).
```

The first rule says that an empty list of faults has `0` fault count: this is the terminating condition for the predicate. The second rule says that the list whose first element is a fault contains `N` faults where `N` is the number of fault counts in the remainder of the list (`N1`) plus 1. The third rule says that a list whose first element is `'nofault'` has the same number of faults as the remainder of the list.

This count_fault predicate makes it possible to ask more complex questions about the circuit, such as:

4. Under what single fault conditions will the output be `[off,off]`:

```
?- circuit(Ins,[off,off],FaultList), count_faults(FaultList,1).
```

5. What are the outputs of a single nofault condition if the first input is `on`?

```
?- circuit([on|_],_,FaultList), count_faults(FaultList,4).
```

By only specifying what is necessary to define the circuit, we have maximized the number and types of questions we can ask about it. By their nature, imperative languages require an overspecification of the problem in order to define machine execution, which in turn limits the types of questions that can be asked.

The example described above is relatively simple. Prolog has been put to use for reasoning about circuits to solve real problems in gate assignment, specialization of standard definitions and determination of signal flow (See for example, [Clocksin, 1987]).

Current Research in Logic Programming

There are two broad areas of active research that deserve mention: parallel logic programming and constraint logic programming.

Parallel Logic Programming

One of the promising features of the Logic Programming model is that it does not *require* sequential execution. Consequently there has been a substantial amount of research recently in two different areas: the development of new concurrent logic programming languages [Shapiro, 1989] and the compilation of Prolog to parallel machines [Ciepielewski, Haridi and Hausman, 1989].

The parallel logic programming languages that have been developed for multi-processors include Guarded Horn Clauses (GHC) from ICOT in Japan, Concurrent Prolog from the Weizmann Institute in Israel and PARLOG from Imperial College in London. Concurrent Prolog has been experimented with on the Intel Hypercube and GHC is the "assembly language" for the Japanese 5th generation hardware systems. In all those languages the model of computation differs somewhat from that of pure logic and also from that of Prolog. The essential difference arises from introduction of commit operations for concurrent processes that restrict the possibility of backtracking. In some of these systems backtracking is not present at all.

One of the major attempts to parallelize the Prolog language was undertaken by the Aurora project. Aurora is one of the first Or-Parallel implementation of Prolog for a multiprocessor shared-memory machine (Sequent Symmetry). The project was developed jointly by teams of researchers at Swedish Institute of Computer Science, Argonne National Labs and Manchester University, Or-parallelism allows different processors to explore alternative choices in a computation which, if they had been executed sequentially would have required backtracking. For example a call to the predicate `count_faults` defined in the program for NAND gates (in Box ???) would, in an or-parrallel Prolog, compute all the possible set of values by assigning a different processor to each of the 3 rules for the `count_faults` predicates.

The results are encouraging so far: on problems that lend themselves to parallel execution, Aurora shows near linear speedup on machines with up to 30 processors. The next step in this research project is Andora which will explore the addition of And-parallelism to Aurora. And-parallelism allows the parallel execution of Prolog procedures that follow in sequence. For example, in the NAND gates example the values of each of the pairs of gates G1, G2 and G4, G5 could be computed in parallel in order to obtain a value for `circuit`.

Constraint Logic Programming

Another promising area of research is in the solution to constraint satisfaction problems. One of the limitations of declarative specification languages like Prolog is that the efficiency with which they will search for solutions can be very sensitive to the order of the statements in the problem. For example, consider a database join problem which seeks to find the solutions to "what are the phone numbers of employees meeting the constraints K and L" (where K = "whose department number begins with 0R" and L= "whose last name begins with V"). If there are many more people that satisfy the constraint K than L, it could be a lot slower to search for the join of "$K \& L$" than to search for the join "$L \& K$". Although the solution sets are the same the sequential search for the first conjuction requires many more operations than the second.

This general efficiency problem is compounded by the generally non logical nature of computer arithmetic which obliges the programmer to write arithmetical constraints in the order in which they are going to be (functionally) executed. Ideally one would want a declarative logic programming language to compute "X is Y + 2 and Y is 3 * 4" in the same way that it computes "Y is 3 * 4 and X is Y + 2". In Prolog systems without constraint satisfaction mechanisms, however, the first case will produce an error because Y has no value at the time that "Y + 2" is being evaluated. In those systems, the second ordering *must* be used because Prolog will first compute Y *and then* X, in sequence.

These two problems—(A) the sensitivity of logic programs to the order of statements and (B) the non-logical nature of arithmetic—have led researchers from several quarters to consider more general techniques for constraint satisfaction. The basic idea behind a constraint logic programming (CLP) system is to provide a mechanism to restrict the set of values to which a logic variable may bind. Thus, in a CLP system, an expression like "X == Y * 3" *constrains* X to be three times Y. It does not *compute* X from Y or Y from X: it restricts the set of values X can have as a function of the set of values Y can have. Hence, if Y is further constrained by "Y >= 1", then an attempt to bind X to a value less than 3 or even the constraint that X be less than 3 will fail and force backtracking.

 The following is a discussion of four different CLP systems: CHIP, CLP(R), Prolog-III and BNR-Prolog.

CHIP

CHIP (Constraint Handling in Prolog) is a system developed by the European Computer Research Center (ECRC) in Munich [van Hentenryck, 1989]. (ECRC is a pre-competitive consortium jointly owned by Seimens, Honeywell-Bull and ICL). CHIP addresses the two problems (A) and (B) described earlier to the

extent that the problem domain is restricted to a finite set of elements. The constraints that can be imposed on these finite domains are restricted to either boolean constraints or equality, inequality and elementary arithmetic on integers.

These are substantial limitations but CHIP is nevertheless an impressive system. For example, their system allows users to express, declaratively, the design of a circuit with more than 1000 logic gates for which test patterns can be generated automatically in times that are comparable to specialized algorithms written in procedural languages [Dincbas, Simonis and van Hentenryck, 1990]. Similar applications have been written in CHIP for solving scheduling problems and other operations-research problems. CHIP is in the process of being commercialized and is expected to enter the market in 1990.

CLP(*R*)

CLP(*R*) [Jaffar and Michaylov, 1987] is an experimental extension of Prolog, for solving linear constraint satisfaction problems. It was developed at the Monash University (Australia) and subsequently bought by IBM (the researchers are located at the IBM-T.J. Watson Research Center). It restricts itself to the solution of problem (B) by extending Prolog to solve systems of linear equations and inequalities (using the simplex method) over the range of real numbers (floating point numbers actually). If a set of equations is non-linear (and cannot be handled by the solver) its evaluation is delayed until the problem becomes linear. CLP(*R*) and CHIP can be considered complementary since CHIP cannot operate on real numbers. CLP(*R*) has been applied to electrical engineering problems and options trading problems.

Prolog-III

Prolog-III [Colmerauer, 1987] is the brainchild of Alain Colmerauer and the successor to Prolog-II. In addition to the extended features of Prolog-II, Prolog-III incorporates a linear programming equation solver (like CLP(*R*)), a boolean constraint solver (like CHIP), and extended facilities for imposing constraints on data-structures (such as lists). Since it has not yet been made available for general use, it is hard to assess the effectiveness of this approach.

BNR Prolog

BNR Prolog is a full-featured Prolog system developed at BNR (see Box 3) for the Macintosh. In some respects the design of BNR Prolog is similar to Colmerauer's Prolog-II but it contains several extentions including a system for performing constraint-based interval arithmetic [Older and Vellino 1989]. In this system, mathematical real numbers are represented as intervals (with floating point lower and upper bounds). Imposing constraints on an interval can narrow

its range and propagate the narrowing to other intervals related to it by constraint equations or inequalities. When combined with Prolog's built-in search mechanism, relational interval arithmetic can be used to obtain numeric solutions to linear and non-linear rational constraint satisfaction problems over the reals (e.g. n-degree polynomial equations). BNR Prolog addresses the problems (A) and (B) but only on arithmetic constraints.

Future of Logic Programming

In many respects Prolog is the flagship of logic programming languages and the future of logic programming in general is closely tied to Prolog's fate. Despite the availability of efficient, compiled Prolog systems for industry standard workstations, mainframes and personal computers (see Table of Products), the general penetration of Prolog into the computer industry is relatively low. The reasons for this are both sociological and technical.

Since the programming model for Prolog is substantially different from that of conventional programming languages, the learning curve is generally higher than it would be for another imperative language. (Early experience with object oriented methodologies has indicated similar steep learning curves.) However, once Prolog is mastered, many programmers prefer using it to imperative languages where performance and memory consumption are not the overriding issues. Most cite productivity gains due to the high level nature of the language and the interactive environment as the main reason.

Open technical issues such as system modularity and appropriate interfaces to other languages tend to impede the use of Prolog in industry, as do the lack of real standards.

On the other hand, Prolog compilers and improvements in processor and memory technology are enabling practical usage of Prolog, and programmer productivity gains seem to be real (although hard to quantify). This would seem to indicate that general acceptance of Prolog may well hinge on its successful industrial application.

The most likely class of applications for Prolog and declarative languages generally appears to be in the rapid prototyping of designs and specifications. The general consensus seems to be that for small to medium sized projects (up to 2-man years, say), programmer productivity increases by about a factor of between 2 and 5, whereas the size of programs is reduced by a factor of between 5 and 10 compared to the same program in a imperative language such as C or Pascal. Although there is very little experience in the development of large scale Prolog programs, it appears that they will not materialize until modules become an intrinsic part of Prolog systems. While many implementations of Prolog use

some notion of module, no system is entirely satisfactory for large scale programming.

It is hard to predict the fate of logic programming as a whole, but it is safe to say that in the niches where it is best suited such as prototyping and natural language processing, its future is secure. Whether or not logic-based languages succeed in penetrating larger application areas depends in part on the development of better software engineering tools, development environments as well as the sucessful integration of the logic paradigm with other programming paradigms.

References

W.F. Clocksin "Logic Programming and Circuit Analysis" in *Journal of Logic Progamming* **4**, 1, pp.59-82, 1987.

Colmerauer, A. "Opening the Prolog-III Universe" Byte Magazine, August, 1987.

Ciepielewski A., Haridi S., and Hausman B. "Or-Parallel Prolog on Shared Memory Multiprocessors" *Journal of Logic Programming* **7**, 2, pp.125-147, 1989.

Dincbas M., Simonis H. and van Hentenryck P., "Soving Large Combinatorial Problems in Logic Programming" *Journal of Logic Programming* **8**, 1&2, pp. 72-93, 1990.

van Hentenryck P. *Constraint Satisfaction in Logic Programming*, MIT Press, pp. 224, 1989.

Jaffar, J. and Michaylov, S. "Methodology and Implementation of a CLP system" *Proc. 4th Int. Conf. on Logic Programming*, J-L. Lassez (Ed), MIT Press, 1987.

Leler, W. *Constraint Programming Languages their specification and generation*, Addison-Wesley pp. 202, 1988.

Older, W. and Vellino, A., "Extending Prolog with Constraint Arithmetic on Real Intervals" CRL Report 89023, 1989.

Shapiro E. "The Family of Concurrent Logic Programming Languages" *ACM Computing Surveys* **21**, pp. 413-510, 1989.

Recommended Prolog Text Books

Bratko, I. *Prolog Programming for Artificial Intelligence*. Wokingham, England, Reading, Mass., Menlo Park, Calif., Don Mills Ont.: Addison-Wesley, 1986.

Clocksin, W. F., and Mellish, C. S. *Programming in Prolog.* 3rd ed. Berlin, Heidelberg, New York, London: Springer-Verlag, 1984.

Covington, M. A., Nute, D., and Vellino, A. *Prolog Programming in Depth.* Glenview, Ill., London: Scott, Foresman and Company, 1988.

O'Keefe, R. A. *The Craft of Prolog* Cambridge, Mass., London England: The MIT Press, 1990.

Pereira, F. C. N., and Shieber, S. M. *Prolog and Natural-Language Analysis.* CLSI Lecture Notes, 10. Stanford: Center for the Study of Language and Information, University of Chicago Press, 1987.

Sterling, L., and Shapiro, E. *The Art of Prolog.* Cambridge, Mass., London England: The MIT Press, 1986.

Declarative vs. Imperative Programming (Box 1)

The difference between declarative and imperative programming can be illustrated by comparing the declarative and procedural instructions for reaching a given destination.

To get to some destination with the declarative model one needs a map that describes the relative locations of the starting point, the destination, and all of the relevant streets in between. Add to this a generic algorithm for finding a path between any two points on a map and the problem is solved. In the procedural model the problem is solved by providing a set of instructions on what to *do* to get from the starting point to the destination by following a sequence of steps.

For example, consider the following map:
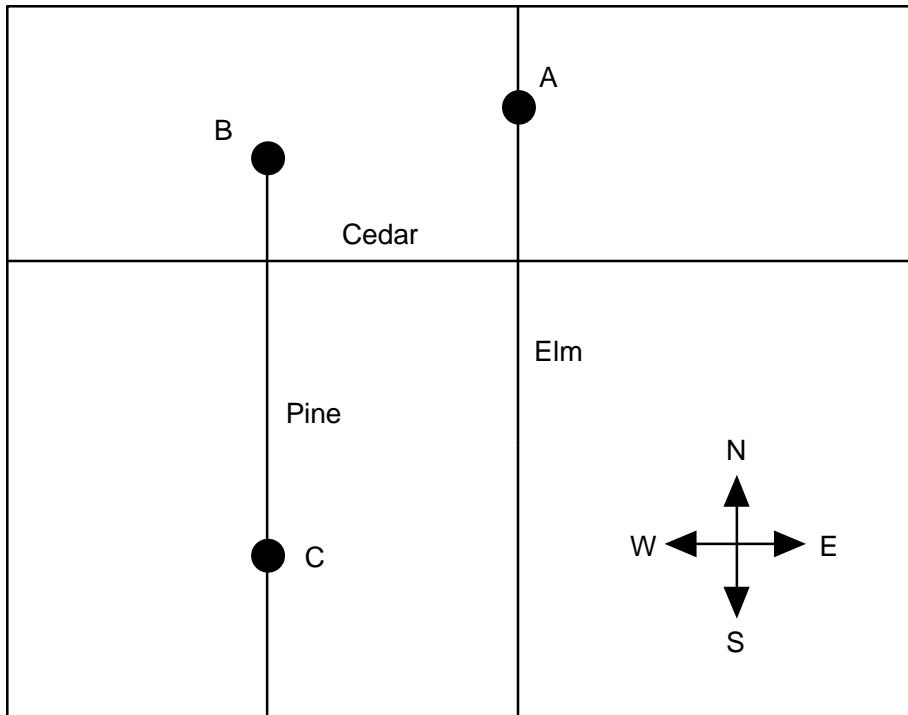
The imperative method for getting from A to C consists of a simple sequence of instructions. The declarative method consists of set of facts that describe the topology of the map, coupled with an inference engine (not shown) that enables you to infer the path to be followed.

To get from A to C

IMPERATIVE

Go south on Elm to Cedar,
then turn West on Cedar to Pine,
then turn south on Pine.

DECLARATIVE

A is on Elm north of Cedar.
C is on Pine south of Cedar.
Cedar intersects Pine and Elm.

One difference between the two methods is that the instructions in the imperative recipe need to be followed in sequential order whereas the declarative facts are order independant. This can be important if we now want to modify these "programs" to get from A to B.

To get from A to B	
IMPERATIVE	*DECLARATIVE*
Follow instructions to get from A to C except turn north on Pine.	B is on Pine north of Cedar.

In the imperative case we can either copy the instructions to get from A to C and them modify the last line or else attempt to re-use part of the A to C procedure and introduce a dependancy between them. On the other hand, the declarative approach is simpler and more modular since we need only add (not modify) a new fact about the map.

There are other advantages and disadvantages to both methodologies. If a traveller doesn't know how to read a map but can easily follow instructions like "turn west on Cedar to Pine" the procedural method has a lot going for it: the instructions are simple, they are easy to follow, they don't take up much memory and they tend to be reliable. But a disadvantage with the procedural approach is that for each significantly different starting point you need a separate set of instructions.

On the other hand, if you can read a map, and you are able to locate both your current position, and your destination, and you have the ability to search for a path on the map, then the declarative approach may be preferable. For example, suppose you are travelling toward the desired destination and you find yourself at a road block on the path that is prescribed by the sequence of imperative instructions. Without explicit instructions on what to do in case an abnormal situation arises, the imperative instructions might make you miss a good party. But with a declarative map and a generic path-finding algorithm there is no need for explicit instructions since you can always construct them dynamically when necessary.

Prolog at BNR (Box 3)

Work with Prolog at BNR began in 1984 as part of a two man exploratory program in Computing Technology investigating fifth generation languages (LISP and Prolog). Several generations of Prolog implementations, each with improved performance and feature complements were developed on XMS workstations. Between 1985 and 1987 XMS Prolog was used by several groups in BNR.

By late 1988 BNR Prolog had become a full-fledged, commercial quality implementation of Prolog for the Macintosh. Its features include high level interfaces to the Mac operating system, facilities for developing stand-alone applications, a foreign-language interface and an integrated constraint and relational arithmetic package. Recently, several new features have been added to BNR Prolog (networking, multitasking, etc.) to permit development of voice/telephony applications on distributed Macs.

Early experience with BNR Prolog concentrated on those applications which conventional wisdom indicated were good fits for a high level symbolic language. More recently, BNR Prolog has been used to implement a Norstar functional terminal, to simulate protocols and new architectures for telephony systems, and to interactively construct user interfaces. One particularly successful application is PRISM (PRolog Interpretive Structural Modeling), a decision support system used for strategic planning (see *Matrix* Vol. 2 Num. 2). In several cases, BNR Prolog has been used in areas which are not representative of its traditional strength. The main advantages seem to be in fast prototyping/iterative designs where the advantages of a symbolic, interactive language and environment outweigh the disadvantages of slower execution speed and initially higher memory costs.

Future plans for BNR Prolog include the development of a portable compiler based on Warren Abstract Machine (WAM) which will run on 32-bit Unix workstations (Sun and HP, for example).

Available Prolog Interpreters and Compilers

Product	Hardware Platform	Cyclic Terms	Freeze[1]	Variable Functors[2]	Cons-traints	Comments
Applied Logic Systems	Unix, PC, Mac, 88K	No	No	No	No	Very fast, generic.
Arity /Prolog	PC	No	No	No	No	Good database/SQL features.
ExperTelligence/Prolog-II	Mac	Yes	Yes	Yes	No	Feature rich, unusual syntax.
Logic Programming Associates	PC/Mac	No	No	Yes	No	Quintus-compatible for Micro-computers.
BIM	Unix	No	No	No	No	Fast, native-code compiler.
Quintus	Unix	No	No	No	No	Industry leader, robust.
SICSTUS	Unix	Yes	Yes	Yes	No	Portable, Quintus look-like, all written in C.
BNR Prolog	Mac (Unix)	Yes	Yes	Yes	Yes[3]	Feature rich, research product.
IBM	VM	Yes	Yes	Yes	No	New high-speed, integrated mainframe pro

All systems have a Module system, Definite Clause Grammar (DCG), Debugger &Foreign Language Interface.

1 Freeze allows the evaluation of an expression to be delayed until its logic variables have been bound.
2 Variable functors permit the expression of F(X) where both F and X are logic variables.
3 Arithmetic constraints on intervals.