A Tutorial on Prolog

Russell C. Bjork

I. Background

PROLOG is a programming language based on the use of mathematical logic--specifically the first order predicate calculus. The name is a contraction for "Programming in Logic". PROLOG was developed in 1972 by Philippe Roussel of the AI Group (Groupe d'Intelligence Artificielle) of the University of Marseille. Specifically, it is an outgrowth of research there on automatic theorem proving. Prolog has been widely used by AI researchers in Europe and Japan. In fact, the Japanese have made it the basis for the software side of their "Fifth Generation" computer project. PROLOG is currently used in a wide variety of areas, not just for automatic theorem proving. It is an excellent tool when one wants to do symbolic (as opposed to numerical) computation. Until recently, Prolog has been less widely known in this country--perhaps due to the "not invented here" syndrome.

The recognized standard definition of PROLOG is a book by W.F. Clocksin and C.S. Mellish: Programming in Prolog (published by Springer Verlag). Quite a number of PROLOG implementations are available for machines ranging from mainframes to microcomputers. For this workshop, we will be using a public domain implementation for IBM PC's known as PD Prolog. However, for serious work with the language a commercial implementation is preferable.

Most implementations realize an extended implementation, embodying all of the features described in Clocksin and Mellish plus additional ones. This means, of course, that programs written for one implementation may not run under another if they use any extended features. One implementation to be aware of--and to avoid--is Turbo PROLOG produced by Borland. Borland has produced a number of fine and reasonably-priced implementations of various programming languages; but in the case of Prolog their implementation is severely deficient on one count, to the point where it hardly qualifies as being Prolog.

PROLOG is utterly different from any other programming language you are likely to have studied. The differences can be summed up in two key words: PROLOG is higher-level than other programming languages you may know. PROLOG is non-procedural. MacLennan says the following about non-procedural programming in his book Principles of Programming Languages (p. 486). A non-procedural language is one "in which the programmer state[s] only what [is] to be accomplished and [leaves] it to the computer to determine how it [is] to be accomplished." For example, to sort an array non-procedurally,

> we might say that B is a sorting of A if and only if B is a permutation of A and B is ordered. We might also have to describe what is meant by a permutation and what we meant by an array being ordered. For the latter we might say that B is

ordered if B[i] ≤ B[j] whenever i < j.

It [is] the responsibility of the nonprocedural programming system to determine how to create an array B that is an ordered permutation of a given array A.

In order to learn PROLOG, you must consciously lay aside some of the thought habits you have formed from working with procedural languages. In some respects, it may be easier for a philosopher to learning PROLOG than for a computer scientist to do so!

II. An overview of the language

A PROLOG program consists of a database of facts and rules. These play a role similar to those of the axioms of a formal system. Facts take the form of predicate calculus predicates--e.g.,

```
derivative(V, V, 1).
```
"The derivative of any variable with respect to itself is 1"

Rules take the form of implications - a given predicate may be inferred if certain other predicates are satisfied--e.g.,

```
derivative(E, V, 0) :- number(E).
```
"The derivative of some expression E with respect to any variable is 0 if E is a number."

The predicate calculus equivalent would be

number(E) => derivative(E, V, 0).

Note that in PROLOG the conclusion is written first, followed by the symbol :- (read "if"), followed by the antecedents--the exact opposite of the way predicate calculus implications are written.

The left hand side of a PROLOG rule (called the "head") must consist of a single predicate. The right hand side (called the "body") may consist of any number of predicates, joined by the logical connectives "and" and "or"--which are written , and ;, respectively. For example:

```
derivative(E1 + E2, V, DE1 + DE2) :-
      derivative(E1, V, DE1),
      derivative(E2, V, DE2).
```
"The derivative of an expression of the form E1 + E2 with respect to some variable V is an expression of the form DE1 + DE2, where DE1 is the derivative of E1 with respect to V and DE2 is the derivative of E2 with respect to V."

```
partial_derivative(E, V, 0) :-
      number(E);
      ( variable(E), E \= V).
```

"The partial derivative of an expression E with respect to some variable V is zero if E is a number or E is a variable other than V."

(As an aside, note that a fact is the special case of a rule where the body of the rule contains no predicates.)

PROLOG rules are actually a form of predicate calculus Horn clause. A Horn clause is a disjunction of literals in which all but one literal is negated.

A rule using only conjunction (,) can easily be transformed into a single Horn clause. For example, our statement of the sum rule for derivatives is equivalent to the predicate calculus formulation

derivative(E1, V, DE1) ^ derivative(E2, V, DE2) = >
derivative(E1 + E2, V, DE1 + DE2)

Recall that an implication of the form A = > B is equivalent to ~A v B. Using this plus DeMorgan's theorem gives

~derivative(E1, V, DE1) v ~derivative(E2, V, DE2) v
derivative(E1 + E2, V, DE1 + DE2)

A rule that uses disjunction can be converted into a set of Horn clauses. We leave this as an exercise to the student.

Two final notes on facts and rules. First, the choice of the names and structures of the predicates used to formulate the database is up to the user. Prolog is a formal system that manipulates predicates without regard to their semantic content. Thus, with regard to our derivative examples I chose to use the predicate name "derivative". I could have chosen to use the name "integral" or "square_root"--but if the rules were the rules for differentiation, I would still get derivatives. Secondly, I chose to make the predicate derivative a three place predicate, with the first argument being an expression, the second the variable of differentiation, and the third the derivative. I could have used another order; I could even have chosen to make the variable of differentiation implicit, in which case I would have ended up with two-place predicates.

As our examples have shown, both facts and rules may contain logical variables. Such variables are always implicitly universally quantified. Thus, our first fact is equivalent to the predicate calculus formulation

forall V: derivative(V, V, 1).

Given a database of facts and rules, computation is initiated by presenting a QUERY or GOAL. The goal represents a theorem to be proved, using the facts and rules

in the database. Goals may also contain logical variables. Variables appearing in a goal are implicitly existentially quantified.

A by-product of the discovery of a proof is the discovering of values for the variables which are needed to make the theorem provable.

For example, given a database containing the following three rules:

```
derivative(V, V, 1).
derivative(E, V, 0) :- number(E).
derivative(E1 + E2, V, DE1 + DE2) :-
      derivative(E1, V, DE1),
      derivative(E2, V, DE2).
```

we could initiate computation with the query

```
derivative(x + 3, x, D).
```

"There exists a D such that the derivative of x + 3 with respect to x is D."

The PROLOG interpreter would discover that this theorem can be proved; in particular, the "D" for which it holds is 1 + 0.

Note for the purposes of this computation, only D is regarded as a logical variable. The variable of differentiation, x, is regarded as a constant symbol. As we will see when we get to PROLOG syntax, the distinction is made by case--logical variables are written starting with upper case letters, and constants with lower case.

With regard to proofs, PROLOG uses what is called in AI the "closed world assumption." If a given theorem cannot be proved using the data available, then it is assumed to be false. That is, given the above database, if we entered the query:

```
derivative(x - 3, x, D).
```

"There exists a D such that the derivative of x - 3 with respect to x is D"

the PROLOG interpreter would answer "No."

PROLOG proves theorems using a method called resolution refutation. The database is regarded as a set of clauses. The theorem to be proved is negated and added to this set. The PROLOG interpreter then attempts to derive a contradiction. If this is possible, then the negation of the theorem must be false, in which case the theorem must be true. If the interpreter cannot derive a contradiction, then the negation of the theorem may be true, in which case the theorem is assumed false.

Resolution is based on the following rule of inference:

Given:	A v B
	~A v C
we may conclude:	B v C
Hence also, given:	A
	~A v B
we may conclude:	B
and, given:	~A
	A v B
we may also conclude:	B

Each new clause derived by resolution is then added to the database as a candidate for participation in further resolutions.

When resolving clauses containing variables, PROLOG finds the most general unifier of the two clauses, and retains this information, so that it can finally report the desired values for the existentially quantified variables of the original query.

For example, our goal `derivative(x + 3, x, D)` would be handled as follows. The original database in clause form will be:

```
derivative(V1, V1, 1).
~number(E) v derivative(E, V2, 0).
~derivative(E1, V3, DE1) v ~derivative(E2, V3, DE2) v
        derivative(E1 + E2, V3, DE1 + DE2)
```

Note that in putting the database in clause form, I have used different names for the "V" variables in each clause to avoid confusion when doing the resolution. This is not necessary in Prolog, because the scope of a variable name is a single clause--thus the "V's" in each of the three clauses of our Prolog database are distinct. I have made this explicit here.

Now add the negation of our goal to the database:

```
~derivative(x + 3, x, D).
```

This can be resolved with our third clause if we make the substitutions:

x/E1, 3/E2, x/V3, DE1 + DE2/D

We add the following resolvent to the database:

```
~derivative(x, x, DE1) v ~derivative(3, x, DE2)
```

Now this can be resolved with our first clause, with substitutions: x/V1, 1/DE1, yielding

the new resolvent:

~derivative(3, x, DE2)

This can in turn be resolved with our second clause, with substitutions:

3/E, x/V2, 0/DE2

yielding any empty resolvent; that is, we have derived a contradiction. Therefore, the negation of our theorem is false, so our theorem is true.

As a by-product, putting our substitutions together, we have discovered that the substitution

1 + 0 / D

is needed to make our theorem hold.

Note that the clauses derived during resolution are not permanently added to the database. In fact, in general, some of them will be untrue, since they are part of the chain of reasoning that leads to a contradiction.

III. Lexical structure of PROLOG

Programs in any programming language are built up out of certain lexical elements. For example, Pascal programs are built out of keywords (**program, const, type, var, begin** etc.); identifiers, constants, and operators. PROLOG programs are built out of atoms, integers, variables, and punctuation.

A PROLOG atom may take one of several forms:

1. A name composed of letters (either case), digits, and the underscore character. The first character must be a lower case letter.

Examples: `a, apple, a_red_apple, all, aBC`

But not: `ABC, _a, 1a`

2. A sequence of one or more special characters from the following list:

`+ - * / ~ < > = \ ' ^ : . ? @ # $ & %`

Examples: `+, \=, :-`

But not: `$a, a$`

3. The following one or two character strings:

`!` `;` `[]` `{}` (in some contexts:) ,

4. Any string of characters enclosed by single quotes. (A ' may be embedded by doubling.)

Examples: `'animals.pro'` -- a file name
`'$op'` -- a "hidden" predicate.

A PROLOG integer is written in the standard way, except that in some versions tilde (~) is used for negative numbers. (However, PD Prolog uses the conventional - sign.)

Examples: `1, 127, -127`

A PROLOG variable is a sequence of letters (either case), digits, and underscores. The first character must be an upper case letter or an underscore.

Examples: `X, XYZ, _XYZ, _xyz, This_is_a_long_name`

Variable names of the form `_1`, `_2`, etc. are generated internally by PROLOG during unification. A single underscore, by itself, is an anonymous variable. If it occurs several times in a given formula, each occurrence is treated as different. For example in the formula

```
p(X) :- q(X)
```

the two occurrences of `X` must unify to the same object. But in the formula `p(_) :- q(_)`, the two variables could unify differently.

PROLOG punctuation marks follow.

1. A period (.) is used to terminate each PROLOG formula.

2. Parentheses are used in constructing predicates and other terms, and for grouping.

3. Square brackets and the vertical bar are used in constructing lists.

4. Double quotes (" ") enclosing a string of characters cause PROLOG to construct a list whose elements are the ASCII codes of the characters.

5. Curly braces are used in writing grammar rules. We will not discuss these.

IV. Syntactic structure of PROLOG

In any programming language, the lexical elements are combined in certain specified

ways to produce more complex constructs. PROLOG has basically two syntactic constructs: structures and lists. A structure may be written as an atom, followed immediately by a left parenthesis (no space allowed), followed by one or more terms separated by commas, and closed by a right parenthesis.

Example: `dog(X), +(1,2), book(programming_in_prolog, clocksin).`

Structures can be used in several ways in PROLOG programs:

1. As the PROLOG equivalent of predicate calculus predicates. When used in this way, the structure is assumed to have the value true or false. For example,

```
dog(X).
```

2. As the PROLOG equivalent of a predicate calculus function. When used in this way, the structure may have a numeric or other kind of value. Note that only built-in functions are available; the user cannot define his own (without modifying the interpreter). For example,

`+(1,2)` could be evaluated (in the right context) to 3.

3. As the PROLOG equivalent of what is called a "record" in other programming languages. For example,

```
book(programming_in_prolog, clocksin, springer_verlag, 1984).
```

The interpretation of a structure depends on the context in which it occurs, as we shall see.

Some terminology: The atom before the left parenthesis is called the functor of the structure. The terms enclosed in parentheses are called the arguments of the structure. The number of arguments is the arity of the structure. For example in:

```
derivative(V, V, 1).
```

The functor is derivative, the arguments are `V`, `V`, and `1`, and the arity is 3.

Note that PROLOG structures, like variables etc. are classed as terms. Thus, a structure may appear as an argument of another structure. For example, in

```
derivative(sin(E), V, cos(E)*DE)
```

the first argument of `derivative` is `sin(E)`, which is itself a structure. This provides a limited capability to have "predicate variables" in PROLOG.

You will note that, PROLOG uses a form of prefix notation for structures. Unlike LISP, however, PROLOG also allows the use of infix notation for certain functors. It is possible to define a PROLOG atom to be an infix operator. Quite a number of these are

defined in the standard PROLOG library, and you can define more using the op built-in predicate. An infix operator can appear with arguments on either or both sides, and the PROLOG reader will convert it internally to the equivalent prefix form. For example, since + is predefined as a built in operator, one can write `1 + 2` as the exact equivalent of writing `+(1,2)`. The internal form is always +(1,2). But the printed form will normally be `1 + 2`. We have made use of this already in our examples.

The following is a list of the more important built-in operators. Many also can behave as functions that can compute values.

1. Logical connectives: `,` `;` `not`

2. General comparison operators `=` `\=` `==` `\==`

Note: "==" is a stricter test than "=". "=" will match an uninstantiated variable to anything by instantiating it, while "==" will only match a variable to another variable it is already sharing with.

3. Arithmetic operators: `+` `-` `*` `/` `mod` `is` `=:=` `=\=` (More on these later.)

4. Arithmetic comparison operators: the above equality operations plus `<` `>` `>=` `=<` (Note the peculiar form: =< not <=).

5. Miscellaneous (to be discussed later): `?-` `:-` `.` `=..`

One limitation of PROLOG structures is that a given structure has a fixed arity. Therefore, PROLOG also provides for lists of arbitrary length. (For those familiar with LISP, the PROLOG list facility is very similar.) The empty list is written `[]`. The notation `[a, b, c]` denotes a list whose elements are `a`, `b`, and `c`. The notation `[H | T]` denotes a list with head (first element) `H` and tail `T`. Note that H is a list element, while `T` is a list.

If we tried to unify `[H | T]` with the list `[a, b, c]`, `H` would be unified with the element `a`, and `T` with the list `[b, c]`.

The notation of surrounding characters with double quotes as in `"anything"` produces a list each of whose elements is the ASCII code of one of the characters in the string--e.g. `"123"` is the same as `[48, 49, 50]`.

The operator =.. which we read univ can be used to transform structures into lists and vice versa. The mapping used is that the head of the list is the functor of the structure and the tail of the list is a list of the structure's arguments. For example:

```
derivative(V, V, 1) =.. L
```

would succeed by instantiating `L` to `[derivative, V, V, 1]`, and

`S =.. [sin, X]` would succeed by instantiating `S` to `sin(X)`.

V. Using the PD PROLOG interpreter

To enter the interpreter, type the command `PDPROLOG` at the `A>` prompt. Typically, PROLOG interpreters operate in two modes:

1. In question answering mode, anything typed is taken as a goal to be satisfied. This is denoted by the `?-` prompt. Try this:

```
?- Y is 2 + 2.
```

Note that all PROLOG formulas must be terminated with a period and a final carriage return. Carriage returns may appear anywhere a space may occur within a formula. Try this:

```
?- Y <CR>
  is <CR>
2 + 2 <CR>
<CR>
.<CR>
```

You may think of input to this mode as being interpreted as if it were prefixed by "Please tell me ...". When PROLOG answers a question, it prints the associated values of the variables. (e.g. `Y` in the above.)

Note that, in some cases, it may be possible to find more than one set of values for the variables that would satisfy the goal. In this case, the interpreter will print the first set of values found, and will then prompt to see if you want more. We will see an example of this shortly.

2. In consultation mode, PROLOG treats all input as facts/rules to be entered in the database for later use in answering questions. You may think of input to this mode as being interpreted as if it were prefixed by "The following is true ...".

Most PROLOG implementations allow you the choice of either consulting a disk file, or consulting you interactively from the keyboard. In the latter case, though, there is no permanent record of the facts and rules you entered. PDPROLOG does not have the capability of interactive consultation, so all facts and rules must be entered into a disk file using a text editor, and can then be read into PDPROLOG by using the system predicate:

```
consult(Filename).
```

In either mode, comments may be entered enclosed in `/*  */`.

To exit the interpreter, type `exitsys` at the `?-` prompt. Don't forget an ending period.

The PDProlog package includes an editor which can be entered from Prolog by typing exec. . The editor will prompt for a file name, which can be either an existing file or a new one. Upon exiting the editor, control will return to Prolog--at which time the edited file can be consulted or reconsulted.

VI. The Search Process

We have said that Prolog satisfies a goal by developing a resolution refutation proof for it. As we all know, given a set of axioms and a theorem to prove, one of the hardest tasks is knowing which axioms to use, and in what order. PROLOG's method of satisfying a goal uses a recursive process that is essentially depth first search. At any moment, there is a current goal that is being satisfied (initially the query that initiated the computation.)

A search is made of the database to find any clauses whose head will unify with the goal literal. If none is found, the goal fails and PROLOG responds no. If more than one is found, PROLOG first tries the one nearest the beginning of the database.

If the selected clause is a fact, then the goal succeeds and PROLOG reports back the unifications used for any variables appearing in the original goal. For example, with the derivative database, if we typed

```
derivative(x, x, D)
```

as a goal, PROLOG would immediately respond

```
D = 1
```

If the selected clause has a body composed of a series of literals, then PROLOG treats each in turn as a subgoal to be satisfied. When one is satisfied, PROLOG moves on to the next. If all are satisfied, then PROLOG reports back as for the bodyless clause case.

If at any time a subgoal fails, but there is another way to satisfy a previously satisfied goal, PROLOG will backtrack to try the alternate approach. Backtracking can go as far back as to select a totally new clause if more than one would unify with the original goal. For example, suppose we had:

```
dog(X) :- barks(X).
dog(X) :- chases_cats(X).
chases_cats(sandy).
```

The goal `dog(A)` would match two clauses. The first would generate the subgoal `barks(X)`, which would fail, so PROLOG would backtrack and try the second clause, leading to the

subgoal `chases_cats(X)` which can be satisfied by giving A the value `sandy`.

Whenever PROLOG tries to satisfy a newly generated subgoal, it begins considering candidates from the beginning of the database. When it backtracks to retry a previously satisfied goal, it begins its search just after the approach that ultimately failed.

Where Prolog departs from being pure logic programming is that the programmer must, in general, give attention to the structuring his program so as to cause the search order to be efficient, and to avoid infinite loops in the search. This can be tricky. There are two basic tools available:

1. The ordering of clauses.

One handle on controlling the search is the relative order of clauses within the database and of subgoals within a clause. For example, suppose I wanted to do some genealogical reasoning, using rules like those in the following biblical genealogy database:

```
ancestor(X,Y) :- father(X,Y).
ancestor(X,Y) :- ancestor(X,Z), father(Z,Y).
father(adam, seth).
father(seth, enosh).
father(enosh, kenan).
```

This database explicitly lists father-son relationships, but not ancestor relationships. Suppose I wanted to generate a list of such relationships. Try this, using the file `fathers1.pro`:

```
consult(fathers1).
ancestor(X,Y).
```

The program succeeds repeatedly by responding `y` to more, and quits when it finds `adam, kenan`.

Now suppose I use exactly the same rules, but with the order of the first two clauses reversed. Try this using the file `fathers2.pro`: [`recon` or reconsult overrides previously defined predicates of the same name and arity.]

```
recon(fathers2)
ancestor(X,Y).
```

This version goes into an infinite loop because of the order of the two clauses: the recursive case is tried before the trivial case!

Actually, the original version has a problem, too. Try this:

```
recon(fathers1).
ancestor(X,Y).      --keep stepping until it loops forever.
```

The problem here is the ordering of the subgoals in the recursive clause. A much better version is found in the file `fathers3.pro`, which is like `fathers1.pro` except that a non-recursive "father" goal precedes the recursive "ancestor" goal in the recursive clause:

```
ancestor(X,Y) :- father(X,Y).
ancestor(X,Y) :- father(X,Z), ancestor(Z,Y).
father(adam, seth).
father(seth, enosh).
father(enosh, kenan).
```

Now try this:

```
recon(fathers3).
ancestor(X,Y).      --it halts when all cases have been found.
```

It should be clear that ordering of goals, both within a single clause and among several clauses within the database, has a strong effect on the operation of PROLOG programs. This is a weakness in the language that is basically a consequence of its very simple, uninformed depth first search strategy. In essence, the programmer must order the goals to make the search more informed.

2. The "cut" operator (!):

The second search control tool is the cut operator. Ordinarily, when a subgoal fails, PROLOG backs up to the last successful goal for which there remain untried alternatives. Sometimes, however, we want to preclude consideration of alternative solutions for a particular goal once a successful one has been found. The cut operator will do this for us. The cut operator has two important properties:

1. Regarded as a subgoal, it always succeeds.

2. As a side-effect, it precludes any consideration of alternative ways of satisfying the goal of which it is a subgoal during backtracking.

The cut operator has three major uses:

1. To confirm the choice of a particular way of satisfying a goal, shutting off alternatives which might lead to problems.

2. To force the failure of a goal without any possibility of satisfying it some other way.

3. To say that the one answer to the question that has been found is satisfactory and no others need be tried.

Here is one example. (We'll get to better ones later.) Suppose we are interested in taking partial derivatives. Then we might have rules like these:

```
partial(V, V, 1) :- !.
partial(E, V, 0) :- variable(E).
```

The second rule says that the partial of some variable with respect to another variable is 0. But this rule would also succeed if we tried to take the partial of a variable with respect to itself. The first rule, with the cut, precludes this.

VII. Built-in predicates

PROLOG has a significant number of built-in predicates. Some of these behave like ordinary predicate calculus predicates; but others are engineered to have useful and important side effects that are needed to make PROLOG a useful programming tool. The built-in predicates fall into a number of categories.

A. Arithmetic operators. We have already noted that PROLOG has a number of built-in operators for doing arithmetic: `+ - * / mod`. We have seen that, while these are normally written using infix notation, the internal representation is actually prefix. Thus, for example, if we type `A + B`, PROLOG treats it as `+(A, B)` (and we may type it this way in the first place if we wish.)

But what is the meaning of something like `+(A, B)`? Clearly, this is not a predicate in the strict sense of the word, since it is not a true/false expression. The answer is that it makes no sense as a predicate (and will always fail if we try to use it as one.)

Recall that in the syntax of PROLOG, our compound terms are called structures, which may be used in several ways--including as a predicate. The arithmetic operators are structures which make no sense as predicates, but which do make sense as functions or general symbolic structures like

```
book(programming_in_prolog, clocksin, springer_verlag, 1984)).
```

But how do we do arithmetic, then? The answer is that PROLOG has a special operator `is` which forces the evaluation of arithmetic operators. `Is` is used in a manner like the following:

<variable or expression> `is` <expression>

The right hand expression must be composed only of valid arithmetic operators, integers, and variables already instantiated to integers. Otherwise an error will occur. Its value is computed using the rules of arithmetic.

If the left hand side is an uninstantiated variable, then it is instantiated to the computed value of the right hand side and the goal succeeds. In effect, this is like the Pascal:

<variable> := <expression>

Otherwise, the left-hand side must be either an integer or a variable instantiated to

an integer. The right-hand side is evaluated, and the "is" goal succeeds only if the two values are the same. Try this:

```
X is 2 + 2.
2 is 1 + 1.
3 is 1 + 1.
1 + 1 is 2. -- fails because left-hand side contains an expression.
1 is X.     -- error because right-hand side contains an uninstantiated
```
variable.

B. PROLOG also has a number of arithmetic comparison operators. Unlike the arithmetic computation operators, these make sense as predicates--e.g. `X < Y` (or `<(X,Y)`) is clearly a true/false expression. When used in conjunction with arithmetic expressions, these force the evaluation of the expressions and compare the results. Try this:

```
1 + 1 < 2 + 1.
1 + 1 > 2 + 1.
X + 1 < 3.  -- fails since X is uninstantiated.
```

When it comes to equality, one must be careful what operator one uses. PROLOG has multiple concepts of "equal."

1. The ordinary equality operators = and \= work on any objects, not just numbers. When used with expressions, they test for identical patterns, not values. Try this:

```
1 + 1 = 2.
1 + 1 = 1 + 1.
```

2. For numerical comparison in our PROLOG,, use the special operators =:= and =\= . Try this:

```
1 + 1 =:= 2.
1 + 1 =:= 1 + 1.
```

In effect, these combine "is" with ordinary equality.

C. Database manipulation predicates.

1. `asserta(clause)` adds a new clause to the front of the database. Try this:

```
dog(snoopy).
asserta(dog(snoopy)).
dog(snoopy).
```

Note: the clause may be a rule as well as a fact, in which case double parentheses are needed:

```
asserta((dog(X) :- beagle(X)).
asserta(beagle(sandy)).
dog(sandy).
```

2. `assertz` is like `asserta` but it puts the clause at the end of the database.

3. `retract(clause)` removes the first matching clause from the database. The clause may contain variables. Try this:

```
retract((dog(X) :- Y)).
dog(snoopy).
dog(sandy).
retract((dog(X) :- Y)).
dog(snoopy).
```

4. `retractall(clause)` removes all matching clauses from the database, like repeated execution of retract.

5. Other database manipulation predicates are somewhat implementation dependent. For these, see Clocksin and Mellish plus the manual for whatever version you use.

6. Note that these predicates allow a Prolog program to be modified as it executes--an essential attribute for AI work. AI programs written in Prolog can acquire and use new knowledge as they run! For example, a derivative-taking program can learn:

```
derivative(V,V,1).
derivative(E,V,0) :- number(E).
derivative(E,V,0) :- variable(E), E \= V.
derivative(E1 + E2, V, DE1 + DE2) :-
      derivative(E1, V, DE1),
      derivative(E2, V, DE2).

derivative(E1 - E2, V, DE1 - DE2) :-
      derivative(E1, V, DE1),
      derivative(E2, V, DE2).

derivative(E1 * E2, V, E1 * DE2 + E2 * DE1) :-
      derivative(E1, V, DE1),
      derivative(E2, V, DE2).

derivative(E1 / E2, V, (E2 * DE1 - E1 * DE2) / (E2 * E2) ) :-
      derivative(E1, V, DE1),
      derivative(E2, V, DE2).

derivative(E, V, D) :-
      write('I don''t know how to take the derivative of '),
      write(E),
      write(' with respect to '),
      write(V),
```

```
        nl,
        write('Please enter a rule for this, or no if you can''t: '),
        read(NewRule),
        deal_with(NewRule),
        derivative(E, V, D).

deal_with(no) :- !, fail.
deal_with(NewRule) :-
        asserta(NewRule).

number(E) :- integer(E).
variable(E) :- atom(E).
```

Try this using expressions involving `sin(x)`, then `sin(x * x)`.

D. Input-output predicates.

1. `read(variable)` reads a term from input and unifies it with variable. Try this:

```
read(X).          -- then type some input.
```

2. `write(term)` writes a term to output. Infix operators are written in infix form. Try this:

```
write(dog(sandy)).
```

3. `nl` writes a newline.

4. `tab(integer)` writes the specified number of spaces.

There are many more predicates in this category. See Clocksin & Mellish plus the manual for your version.

E. Metalogical predicates

1. `var(variable)` is true iff variable is currently uninstantiated or is unified with another variable.

2. `nonvar(variable)` is true iff variable is currently instantiated to something other than another variable.

3. `atom(variable)` is true iff variable is currently instantiated to an atom.

4. `integer(variable)` is true iff variable is currently instantiated to an integer.

5. `atomic(variable)` is true iff variable is currently instantiated to either an atom or an integer.

6. `functor(T,F,N)` succeeds if `T` is a structure, `F` is its principal functor, and `N` is its arity. Note that either `F` and `N` can be variables or `T` can be a variable--but not both.

7. `arg(N,T,A)` instantiates `A` to the `N`th argument of a structure `T`.

8. `X =.. L` (pronounced "univ") converts between a structure `X` and a list `L` of the form [functor, arg1, arg2 ...]

9. `name(A,L)` converts between an atom A and a list of characters comprising its name.

F. Control structures

1. `call(G)` --where `G` is instantiated to a goal--attempts to satisfy `G`, and succeeds iff `G` succeeds.

2. `repeat` followed by any number of clauses causes the clauses it to be retried whenever backtracking occurs. For example, the following loop would accept and echo input until the user types quit:

```
repeat, read(X), print(X), X = quit.
```

3. `fail` means what it says--the current subgoal fails and backtracking is initiated.

Appendix

```
/* Three versions of a biblical genealogy database.  Only the third one behaves
correctly: */

/* 1 */     ancestor(X,Y) :- father(X,Y).
            ancestor(X,Y) :- ancestor(X,Z), father(Z,Y).
            father(adam, seth).
            father(seth, enosh).
            father(enosh, kenan).

/* 2 */     ancestor(X,Y) :- ancestor(X,Z), father(Z,Y).
            ancestor(X,Y) :- father(X,Y).
            father(adam, seth).
            father(seth, enosh).
            father(enosh, kenan).

/* 3 */     ancestor(X,Y) :- father(X,Y).
            ancestor(X,Y) :- father(X,Z), ancestor(Z,Y).
            father(adam, seth).
            father(seth, enosh).
            father(enosh, kenan).

/* The following program knows how to take the derivative of some functions, and
can be taught how to take others. */

      main :-
          repeat,
            write('Enter function: '),
            read(F),
            handle(F),
          F = end.

      handle(end) :- !.
      handle(F) :-
          write('Enter variable: '),
          read(V),
          derivative(F, V, D),
          write('Derivative is:  '),
          write(D),
          nl,
          !.
      handle(F) :- write('I can''t do this one.'), nl.

      derivative(V,V,1).
      derivative(E,V,0) :- number(E).
      derivative(E,V,0) :- variable(E), E \= V.

      derivative(E1 + E2, V, DE1 + DE2) :-
          derivative(E1, V, DE1),
          derivative(E2, V, DE2).
```

```
derivative(E1 - E2, V, DE1 - DE2) :-
    derivative(E1, V, DE1),
    derivative(E2, V, DE2).

derivative(E1 * E2, V, E1 * DE2 + E2 * DE1) :-
    derivative(E1, V, DE1),
    derivative(E2, V, DE2).

derivative(E1 / E2, V, (E2 * DE1 - E1 * DE2) / (E2 * E2) ) :-
    derivative(E1, V, DE1),
    derivative(E2, V, DE2).

derivative(E, V, D) :-
    write('I don''t know how to take the derivative of '),
    write(E),
    write(' with respect to '),
    write(V),
    nl,
    write('Please enter a rule for this, or no if you can''t: '),
    read(NewRule),
    deal_with(NewRule),
    derivative(E, V, D).

deal_with(no) :- !, fail.
deal_with(NewRule) :-
    asserta(NewRule).

number(E) :- integer(E).
variable(E) :- atom(E).
```

```
/* An expert system which can learn about animals */

title(Program) :- Program = 'IDENTIFIER system - Winston chapter 6'.

p(X,Y) :- ask(X,Y,Answer), assert_answer(X,Y,Answer), !, Answer = true.
ask(X,Y,Answer) :- print(does), tab(1), print(Y), tab(1), print(X), print('?'),
                   tab(1), read(A), yes_no(Answer,A), !.
yes_no(true,A) :- A = y ; A = yes.
yes_no(fail,C).
assert_answer(X,Y,true) :- asserta((p(X,Y) :- !)).
assert_answer(X,Y,fail) :- asserta((p(X,Y) :- !, fail)).

mammal(X) :- p(have_hair,X), !.
mammal(X) :- p(give_milk,X).
bird(X) :- p(have_feathers,X), !.
bird(X) :- p(fly,X), p(lay_eggs,X).
carnivore(X) :- mammal(X), p(eat_meat,X), !.
carnivore(X) :- mammal(X), p(have_pointed_teeth,X), p(have_claws,X),
                p(have_eyes_that_point_forward,X).
ungulate(X) :- mammal(X), p(have_hoofs,X), !.
ungulate(X) :- mammal(X), p(chew_cud,X).
even_toed(X) :- mammal(X), p(chew_cud,X).
isa(X,cheetah) :- carnivore(X), p(have_tawny_color,X), p(have_dark_spots,X).
isa(X,tiger) :- carnivore(X), p(have_tawny_color,X), p(have_black_stripes,X).
isa(X,giraffe) :- ungulate(X), p(have_long_legs,X), p(have_long_neck,X),
                  p(have_tawny_color,X), p(have_dark_spots,X).
isa(X,zebra) :- ungulate(X), p(have_white_color,X), p(have_black_stripes,X).
isa(X,ostrich) :- bird(X), not(p(fly,X)), p(have_long_legs,X),
                  p(have_long_neck,X), p(have_black_and_white_color,X).
isa(X,penguin) :- bird(X), not(p(fly,X)), p(swim,X),
                  p(have_black_and_white_color,X).
isa(X,albatross) :- bird(X), p(fly,X).
isa(X,dog) :- carnivore(X), p(bark, X), p(chase_cats,X).
identify(X) :- isa(X,Y), print(X), tab(1), print(is), tab(1), print(a),
               tab(1), print(Y), nl.
```