A Concise Introduction to Prolog

A first look at Prolog

Prolog is a *logic language,* not an algorithmic language, and one therefore has to learn to think about programs in a somewhat different way. The terminology is also somewhat different.

The following is a simple Prolog program:

```
man(socrates).
mortal(X) :- man(X).
```

The first line can be read, "Socrates is a man." It is a *base clause,* which represents a simple fact.

The second line can be read, "X is mortal if X is a man;" in other words, "All men are mortal." This is a *clause,* or rule, for determining when its input X is "mortal." (The symbol ":-", sometimes called a *turnstile,* is pronounced "if".) We can test the program by asking the question:

```
| ?- mortal(socrates).
```

that is, "Is Socrates mortal?" (The "`| ?-`" is the computer's prompt for a question.) Prolog will respond "`yes`". Another question we may ask is:

```
| ?- mortal(X).
```

That is, "Who (X) is mortal?" Prolog will respond "`X = socrates`".

Prolog syntax (approximate)

A program, or *database,* in Prolog consists of one or more *predicates;* each predicate consists of one or more *clauses.* A clause is a *base clause* if it is unconditionally true, that is, it has no "if part."

```
<program> ::= <predicate> | <program><predicate>

<predicate> ::= <clause> | <predicate><clause>

<clause> ::= <base clause> | <nonbase clause>
```

Two clauses belong to the same predicate if they have the same *functor* (name) and the same *arity* (number of arguments). Thus, `mother(jane)` and `mother(jane, jim)` are different predicates.

```
<base clause> ::= <structure> .

<nonbase clause> ::= <structure> :- <structures> .

<structures> ::= <structure> | <structure> , <structures>
```

A *structure* is a functor followed by zero or more arguments; the arguments are enclosed in parentheses and separated by commas. *There must be no space between the functor and the opening parenthesis!* If there are no arguments, the parentheses are omitted.

```
<structure> ::= <name> | <name> ( <arguments> )

<arguments> ::= <argument> | <argument> , <arguments>
```

Arguments may be any legal Prolog values or variables.

A *variable* is written as a sequence of letters and digits, beginning with a capital letter. The underscore (_) is considered to be a capital letter.

An *atom* is any sequence of letters and digits, beginning with a lowercase letter. Alternatively, an atom is any sequence of characters, enclosed by single quotes ('); an internal single quote must be doubled. Examples: `cat`, `r124c41`, `max_value`, `maxValue`, `'max value'`, `'Don''t go'`.

As syntactic sugar, Prolog allows certain infix operators: ',' (comma), ';' (semicolon), ':-' (turnstile), +, -, *, /, =, ==, and many others. These are the same as the operator written as the functor of a structure; for example, `2+2 is the same as '+'(2,2)`.

Comments begin with the characters `/*` and end with `*/`. Comments are not restricted to a single line, but may not be nested.

Example: `/* This is a comment. */`

Unification and instantiation

Unification and instantiation can be performed explicitly with the '=' operator, or implicitly via parameter transmission. Unification is a symmetric operation (X=Y is the same as Y=X), and is not the same as assignment.

1. Any value can be unified with itself. (This is normally useful only as a test.)

   Example: `mother(john) = mother(john).`

2. A variable can be unified with another variable. The two variable names thereafter reference the same variable.

   Example: `X = Y, X = 2, write(Y). /* Writes the value 2. */`

3. A variable can be unified with any Prolog value; this is called *instantiating* the variable. A variable is *fully instantiated* if it is unified with a value that does not itself contain variables.

   Example: `X = foo(bar, [1, 2, 3]). /* X is fully instantiated. */`
   Example: `Pa = husband(Ma). /* Pa is partially instantiated. */`

4. Two different values can be unified if there are unifications for the constituent variables which make the values the same.

   Example: `mother(mary, X) = mother(Y, father(Z)).`
   [Also results in the unifications `mary=Y` and `X=father(Z)`].

5. It is legal to unify a variable with an expression containing itself; however, the resultant value cannot be printed, and must otherwise be handled with extreme care.

   Example: `X = foo(X, Y).`

Variables

A *variable* is written as a sequence of letters and digits, beginning with a capital letter. The underscore (_) is considered to be a capital letter. Examples: `X2`, `Max`, `This_node`, `ThisNode`.

Prolog variables are similar to "unknowns" in algebra: Prolog tries to find values for the variables such that the entire clause can succeed. Once a value has been chosen for a variable, it cannot be altered by subsequent code; however, if the remainder of the clause cannot be satisfied, Prolog may backtrack and try another value for that variable.

The *anonymous variable* consists of a single underscore. Each occurrence of the anonymous variable is considered to be a new, distinct variable (i.e. different occurrences may have different values).

The scope of a variable name is the clause in which it occurs. There are no "global" variables.

Prolog execution

Most Prolog clauses have both a *declarative reading* and a procedural reading. Whenever possible, the declarative reading is to be preferred.

```
mother(X, Y) :- parent(X, Y), female(X).
```

Declarative reading: `X` is the mother of `Y` *if* `X` is a parent of `Y` *and* `X` is female.

Approximate procedural reading: To show that `X` is the mother of `Y`, first show that `X` is a parent of `Y`, then show that `X` is female.

Suppose we have the additional base clauses:

```
parent(john, bill).
parent(jane, bill).
female(jane).
```

Now if we inquire:

```
| ?- mother(M, bill).
```

the clause of `mother/2` will be located, and the unifications `X=M, Y=bill` will occur. (Parameter transmission is by unification.) Then `parent(M, bill)` will be attempted, resulting in the unification `M=john`. Next, `female(john)` will be attempted, but will fail. Prolog will backtrack to `parent(M, bill)` and look for another solution for this; it will succeed and unify `M=jane`. Finally, it will attempt `female(jane)`, and succeed; so the inquiry will succeed, having performed the unification `M=jane`.

Typically Prolog predicates work regardless of which arguments are instantiated, and may instantiate the others. Thus `mother/2` works equally well for the calls `mother(jane,C)`, `mother(M,C)`, and `mother(jane,bill)` [but the procedural reading is different in each case.] Injudicious use of control predicates, particularly "cut,", can destroy this property.

Lists and strings

`[]` is the empty list; `[a, 2+2, [5]]` is the list containing the three elements `a`, `2+2`, and `[5]`. `[Head | Tail]` is the list whose first element is `Head` and whose *tail* (list of remaining elements) is `Tail`. `[a, X, c | Tail]` is the list whose first three elements are `a`, `X`, and `c`, and whose remaining elements are given by the list `Tail`. Only one term may follow the "|": `[a | X, Y]` and `[a | X | Y]` are syntactic nonsense.

Unification can be performed on lists:

```
[a, b, c] = [Head | Tail].      /* a = Head, [b, c] = Tail. */
[a, b] = [A, B | T].    /* a = A, b = B, [] = Tail. */
[a, B | C] = [X | Y].   /* a = X, [B | C] = Y. */
```

In most (but not all) Prolog systems, the list notation is syntactic sugar for the '.' functor, with the equivalence: `'.'(Head, Tail) = [Head | Tail]`.

Two useful predicates are `member/2`, which succeeds when its first argument is a member of the list that is its second argument, and `append/3`, which is true when the third argument is the list formed by appending the first two lists. *Neither is predefined.* Definitions are:

```
member(A, [A | _]).
member(A, [_ | B]) :- member(A, B).

append([A | B], C, [A | D]) :- append(B, C, D).
append([], A, A).
```

The operator "`=..`", pronounced "univ," succeeds when its left argument is a structure and its right argument is the corresponding list [Functor | Args].

Example: `mother(M, bill) =.. [mother, M, bill]`.

A double-quoted character string is syntactic sugar for a list of the ASCII codes for those characters.

Example: `"abc" = [97,98,99]`.

The predicate `name/2` succeeds if its first argument is the atom formed from the string that is its second argument.

Example: `name(abc, "abc")`.

How to run Prolog

How you start Prolog depends on your operating system; try typing `prolog` at the top-level prompt. Prolog should respond with the "`| ?-`" prompt.

To load in predicates from file, use `reconsult(`*`FileName`*`) or reconsult([`*`FileName1, FileName2, ...`*`])`. If one of the files contains a predicate that already occurs in the database, it replaces the old definition. [The similar predicate `consult` adds the new clauses after the existing predicates, rather than replacing them.]

To type in predicates directly, use `reconsult(user)`. The prompt will change from "`| ?-`" to "|". Enter predicates as you would on a file. To quit this mode, enter an end-of-file (probably ^D).

"Run" the program by typing in inquiries at the Prolog prompt. You may call any predicate with any arguments, and you may make multiple calls in one inquiry by separating them with commas. Use a period at the end of each inquiry. There is no "main" program.

When Prolog does not give you a new prompt after it answers an inquiry, that means there may be other answers. Enter

```
;<return>
```

to tell it to get the next answer, or just `<return>` to tell it you have seen enough.

To begin tracing, use `trace`; to end tracing, use `notrace`. To exit Prolog, use `halt`.

How to write Prolog programs

Prolog is a notation for stating logical relations that happens to be executable. It has few control structures, because it is very difficult to assign meanings to control structures. Accordingly, you should try to learn how to write declarative programs. Avoid using the control structures listed below. If you find Prolog frustrating and difficult, you are probably still programming procedurally.

A Prolog predicate consists of multiple clauses. It is useful to think of each clause as coding a separate "case"--first describe what constitutes the case, then describe the result for that case. For example, the absolute value of a number is (1) the negation of the number, if it is negative, or (2) the number itself, if it isn't negative.

```
abs(X, Y) :- X < 0, Y is -X.
abs(X, X) :- X >= 0.
```

If a case has subcases, feel free to invent another predicate to deal with the subcases.

Remember that parameter transmission is by unification. You can do a lot of your work right in the parameter list. For example:

```
second([_, X | _], X).  /* 2nd argument is 2nd element of list. */
```

You can often simplify code by writing parameters that match only the specific case you are interested in. See `member` and `append` for examples. (Note: when you must program procedurally, by convention the "result" is the last argument.)

Recursion is fully supported. In other languages you must always test for the base case first; in Prolog, the base case can (and should) go last, if it is such that the more general clauses will fail--see `append`. If the predicate is to fail in the base case, it can (and should) be omitted; for example, the base case for `member` is the unnecessary clause:

```
member(_, []) :- fail.  /* No 1st parameter is a member of [] */
```

You can't keep a value for future use by "assigning it to a variable." If it is temporary, local information, you can pass it around as a parameter; if it is relatively permanent information that should be globally accessible, you can

put it in the database (see `assert` and `retract` below). Prolog has no functions, so "results" must be returned by instantiating one or more parameters.

Many predicates can be used as generators, to generate one solution after another until later conditions are met. For example,

```
member(X, [1, 2, 3, 4, 5]), X > 3.
```

succeeds and instantiates `X` to `4`. If backed into, it re-instantiates `X` to `5`. (But if you think declaratively, this just says "`X` is a member of the list `[1, 2, 3, 4, 5]` that is greater than `3`.")

When one clause fails, the next one is tried. If you want the failure of a clause to cause the failure of the entire predicate, you can use a cut-fail combination:

```
sqrt(X, RootX) :- X < 0, !, fail.
(more clauses of sqrt should follow)
```

This is a procedural shortcut that avoids the necessity of having `X <= 0` in every clause; it is justified only if the test is complex and there are many clauses.

Arithmetic is performed only upon request. For example, `2+2=4` will fail, because `4` is a number but `2+2` is a structure with functor '+'. Prolog cannot work arithmetic backwards; the following definition of square root ought to work when called with `sqrt(25, R)`, but it doesn't.

```
sqrt(X, Y) :- X is Y * Y.       /* Requires that Y be instantiated. */
```

Arithmetic is procedural because Prolog isn't smart enough to solve equations, even simple ones. This is a research area.

It is possible to build a so-called *fail loop* in Prolog. Such a loop has the form *generate-process-test;* the loop repeats if the test fails. For example, the following will print the elements of a list, one per line:

```
print_elements(List) :- member(Element, List), write(Element), nl, fail.
```

However, if the processing is at all complex, it may be difficult to backtrack over it safely. Tail recursion is safer, cleaner, and usually more efficient:

```
print_elements([Head | Tail]) :- write(Head), nl, print_elements(Tail).
```

Both of these fail after printing the list. If this is undesirable (and it probably is), a simple idiom is to add another clause whose purpose is to unconditionally succeed after the first clause is done:

```
print_elements(_).
```

Built-in Predicates

Prolog has a large number of built-in predicates. The following is a partial list of predicates which should be present in all implementations.

Input predicates

`read(X)`
: Read one clause from the current input and unify it with `X`. If there is no further input, `X` is unified with `end_of_file`.

`get(X)`
: Read one printing character from the current input file and unify the ASCII code of that character (an integer) with `X`.

`get0(X)`
: Read one character from the current input file and unify the ASCII code of that character with `X`.

`see(File)`
: Open `File` as the current input file.

`seen`
: Close the current input file.

Output predicates

`write(X)`
: Write the single value `X` to the current output file.

`writeq(X)`
: Write `X` with quotes as needed so it can be read in again.

`tab(N)`
: Write `N` blanks to the current output file.

`nl`
: Write a newline to the current output file.

`put(X)`
: Write the character whose ASCII value is `X` to the current output file.

`tell(File)`
: Open `File` as the current output file.

`told`
: Close the current output file.

Control predicates

`X ; Y`
: `X` or `Y`. Try `X` first; if it fails (possibly after being backtracked into), try `Y`.

`(X -> Y)`
: If `X`, then try `Y`, otherwise fail. `Y` will not be backtracked into.

`(X -> Y ; Z)`
: If `X`, then try `Y`, else try `Z`. `X` will not be backtracked into.

`not X`
: (Sometimes written `\+X` or `not(X)`) Succeed only when X fails.

`true`
: Succeed once, but fail when backtracked into.

`repeat`
: Always succeed, even when backtracked into.

`fail`
: Never succeed.

`!`
: (Pronounced "cut".) Acts like `true`, but cannot be backtracked past, and prevents any other clauses of the predicate it occurs in from being tried.

`abort`
: Return immediately to the top-level Prolog prompt.

Database manipulation predicates

`assert(X)`
: Add `X` to the database. For syntactic reasons, if `X` is not a base clause, use `assert((X))`.

`asserta(X)`
: Add `X` to the database in front of other clauses of this predicate.

`assertz(X)`
: Add `X` to the database after other clauses of this predicate.

`retract(X)`
: Remove X from the database. For syntactic reasons, if `X` is not a base clause, use `retract((X))`.

`abolish(F,A)`

Remove all clauses with functor `F` and arity `A` from the database.

`clause(X,V)`
: Find a clause in the database whose head (left hand side) matches `X` and whose body (right hand side) matches `V`. To find a base clause, use `true` for `V`.

`save(F)`
: Save the entire program state on File `F (usu. as a binary image)`.

`restore(F)`
: Replace the program state with the one on File `F`.

Arithmetic predicates

`X is E`
: Evaluate `E` and unify the result with `X`.

`X + Y`
: *When evaluated,* yields the sum of `X` and `Y`.

`X - Y`
: *When evaluated,* yields the difference of `X` and `Y`.

`X * Y`
: *When evaluated,* yields the product of `X` and `Y`.

`X / Y`
: *When evaluated,* yields the quotient of `X` and `Y`.

`X mod Y`
: *When evaluated,* yields the remainder of `X` divided by `Y`.

`X =:= Y`
: Evaluate `X` and `Y` and compare them for equality.

`X =\= Y`
: Evaluate `X` and `Y` and succeed if they are not equal.

...and similarly for `>`, `<`, `>=`, `=<`.

Listing and debugging predicates.

`listing(P)`
: Display predicate `P`. `P` may be a predicate name, a structure of the form `Name/Arity`, or a bracked list of the above.

`trace`
: Turn on tracing.

`notrace`
: Turn off tracing.

`spy P`
: Turn on tracing when predicate `P` is called. `P` may be a predicate name, a structure of the form `Name/Arity`, or a non-bracked list of the above.

`nospy P`
: Turn off spying for `P`.

`nospyall`
: Turn off all spypoints.

`debug`
: Enable spypoints (allow them to initiate tracing.).

`nodebug`
: Disable spypoints (without removing them).

Tracing

Control over tracing is very system-dependent, but is probably like this:

Enter key

Single-step to next line of trace.

`h`
Provide help on tracing commands.

`s`
(On a CALL) Skip over this call.

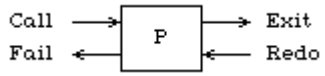
`l`
Leap without tracing to the next spypoint.

`n`
Turn off tracing.

`+`
Set a spypoint here.

`-`
Remove the spypoint here.

Trace terminology: CALL is the initial entry to a predicate; EXIT is a successful return; REDO is when it is backed into for another answer; FAIL is when it finds no more solutions. A sequence of calls may be viewed as forming a chain.



Tests

`atom(X)`
Succeed if `X` is an atom (an empty list is considered an atom).

`atomic(X)`
Succeed if `X` is an atom or number.

`number(X)`
Succeed if `X` is a number.

`integer(X)`
Succeed if `X` is an integer.

`float(X)`
Succeed if `X` is a real number.

`var(X)`
Succeed if `X` is unbound (a non-instantiated variable).

`nonvar(X)`
Succeed if `X` is bound.

`X == Y`
Succeed if `X` and `Y` are identical (but do not unify them).

`X \== Y`
Succeed if `X` and `Y` are not identical.