Prolog Programming

Derived from PROgramming in LOGic (1972)

Good at expressing logical relationships between concepts

Prolog and LISP - two most popular AI languages

Execution of a Prolog program is a process of inference, inferring new facts from old facts

Prolog programs based on predicate logic using Horn clauses

A Horn clause is an implication in which the consequent does not contain connectives
Eg,
   \[ R \leftarrow P \land Q \]
Clauses (Rules) in Prolog

A clause has the following structure:

```
conclusion :- condition(s)
```

:- read as IF

The L.H.S is a single concept (logical atom)
The R.H.S of a clause might be constructed like:

```
condition1 AND condition2 AND condition3
```

or like:

```
condition1 AND condition2 OR condition3
```

In Prolog

- AND is represented by a comma (,)
- and OR by a semi-colon (;) or a separate line.

Facts represent statements of truth and are expressed as predicates, eg,

```
elective_unit(m219).
core_unit(m209, cs).
cs(m104, core_unit, part1).
cs(m219, elective, part2).
functor(relation) objects(arguments)
```
A simple Prolog program

Prolog programs consist of clauses used to make inferences
Example:

```
is_a(dog, butch).
is_a(cat, cassidy).
person(anne, female, 23).
person(tom, male, 19).
```

Prolog programs are executed by posing questions, or queries:

```
?- is_a(cat, cassidy).
   Yes
```

The Prolog interpreter simply scans the list of facts (top to bottom) until it finds a matching one. Once this happens it displays the "Yes".

```
?- person(anne, female, 25).
   No
```

The program has not been able to match the query with any fact, since we know that Anne is in fact 23 years old, so it concludes that the query, or proposition, is false.
Prolog and its “universe of discourse”

Consider this query;

?- is_a(dog, arthur).
No

As far as the program is concerned there is no dog named Arthur, and since the program cannot know everything, it must give a "No" result.

There may well be a dog name Arthur, it is just that this fact is unknown to the program (outside its universe of discourse).

A “No” result may therefore mean one of two things:

1. The proposition is indeed false, or
2. There is insufficient information to prove it true.
A slightly more complex program involving a rule:

\[
\begin{align*}
\text{is\_a(dog, butch).} \\
\text{is\_a(cat, cassidy).} \\
\text{is\_a(cat, ginger).} \\
\text{person(anne, female, 23).} \\
\text{person(tom, male, 19).} \\
\text{likes(tom, cassidy):- person(tom, male, 19), is\_a(cat, cassidy).}
\end{align*}
\]

We can now pose the query:

?- likes(tom, cassidy).

The Prolog interpreter now tries to match this query against the facts or with the conclusion parts of the rules in the program.

In this case a match occurs with the rule. For this rule to be proven true its right hand side must first be proven true.

We now have two new propositions to prove:

Is Tom a male person of age 19? AND Is Cassidy a cat?
Queries involving variables

Prolog enables us to make queries using predicates containing variables.

If we ask the query

?- is_a(cat, X).

then Prolog will respond firstly with

X = Cassidy

and then (by typing a ;) with

X = Ginger

X is instantiated to each of the above values for the query to succeed.

Thus Prolog can do more than just simply CONFIRM or DENY whether a particular proposition is a logical consequence of a set of premises.
Prolog - Structures and Operations

Data objects in Prolog

A Prolog program consists of terms

A term can be

- a constant
- a variable
- a structure

Constants

Represent specific objects or relationships
Can be numbers or atoms

Atoms

Can be names of objects or relationships (predicates)

eg,

joanne, plato, m219 (objects)
mathmajor, hastaken, is_a (relationships)

Must begin with lower case letters unless inside quotes, eg, ‘Joanne Bloggs’

Can also be special symbols,

eg,

:-, ?, !, ^, ;

**Prolog variables**

Represent some object to be named  
Must begin with an uppercase letter or ‘_’  
eg,  
X, Person, Which, _alpha, Gross_Pay

*Anonymous* variables  
Written as an underscore  
Represent objects which will never be named, eg,  
hasachild( X ) :- parent( X, _ ) .

**Overview of Prolog data objects**

```
terms
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>constants     variables structures</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>atoms           numbers</td>
</tr>
</tbody>
</table>
```

**Prolog Structures**

A structure is a single object which consists of other objects (components). Eg,

- a date composed of day, month and year

A structure in Prolog is specified by its functor and components,

Eg,

```
date (11, february, 1997 )
```

Components may themselves be structures,

Eg,

```
owns( john, book( hamlet, author( william, shakespeare )))
```

is a fact consisting of a structure (author) within another structure (book)

Note: Syntax of a structure is the same as that of a fact

Write a Prolog query to list John’s Shakespeare collection

Example: Representation of a specific triangle using a structure

```
triangle1 (vertex(2,3), vertex(15, 19), vertex(27, 3))
```
Lists

A list is a data structure that contains one or more elements.

Prolog lists are denoted by square brackets
Eg,

```
[murdoch, curtin, uwa, ecu]
```

The order of the elements is important.
The list `[a, b, c]` is distinct from the list `[b, c, a]`

Unlike sets, a list may contain several instances of the same element.

Elements of a list may themselves be lists
```
[a, b, [c, d, [e, f]]]
```

The *length* of a list is the number of elements contained in it.

The *empty list* is a list of length 0 and is denoted by `[ ]`
The head and tail of it..

The leftmost element of a list is its head, The remainder of the list is its tail

Eg, the list

\[ \text{murdoch} \mid \text{curtin, uwa, ecu} \]

has

element murdoch as its head, and the list

\[ \text{curtin} \mid \text{uwa, ecu} \]
as its tail

In Prolog, we denote a list with head \( H \) and tail \( T \) as \([H \mid T]\)

For example, we may write the list

\[ \text{[a, b, c]} \]

alternatively as

\[ \text{[a} \mid \text{[b, c]]} \]
Non-example: \([a \mid b] \) is not the same as \([a, b]\)

In fact, in this case, \([a \mid b]\) is not even a list since the \(TAIL\) is not a list but an element.

Note however that \([a, b]\) is equivalent to \([a \mid [b]]\)

In fact a list may be constructed recursively as follows:

(i) the empty list \([\ ]\) is a list.

(ii) if \(H\) is an element and \(T\) is a list, then \([H\mid T]\) is a list.

Thus the following are all equivalent

\([a, b, c]\)
\([a \mid [b, c]]\)
\([a \mid [b \mid [c]]]\)
\([a \mid [b \mid [c \mid []]]]\)

As a further notational convention

\([a, b \mid T]\) is an abbreviation for \([a \mid [b \mid T]]\)

\([a, b, c \mid T]\) is an abbreviation for \([a \mid [b \mid [c \mid T]]]\)

..

.. etc.
Calculating the length of a list

We can calculate the length of a list via a predicate

\[
\text{length}(X,N)
\]

defined to be true precisely when the list \( X \) has length \( N \).

Again, a recursive definition follows easily and naturally from the following observation

The length of the tail of a list is 1 less than the length of the whole list.

or, in other words

The length of a list is 1 more than the length of its tail.
We make use of the property,

The length of a list is 1 more than the length of its tail.

To write our list as \([H|T]\), and obtain the rule for its length as

\[
\text{length}([H|T],N) :\text{-} \text{length}(T,M) \text{, } N \text{ is } M+1.
\]

Thus, including an appropriate terminating condition, the final definition becomes

\[
\text{length}([ ], 0).
\]

\[
\text{length}([H|T],N) :\text{-} \text{length}(T,M) \text{, } N \text{ is } M+1.
\]

Example:

?- length( [howard, keating, hawke], Length ).

Length = 3
List examples:

family ( [chris, janet, martin, david] ).

?- family ( All ). (query)
All = [ chris, janet, martin, david ] (response)

?- family ( [ Head | Rest ] ).
Head = chris
Rest = [ janet, martin, david ]

?- family ( [ Dad, Mum | Kids ] ).
Dad = chris
Mum = janet
Kids = [ martin, david ]

?- family ( [ Head | _ ] ).
Head = chris
Arithmetic using Prolog

Arithmetic operators in Prolog:

+ - * /

// (integer division)

mod (remainder after division
eg, 9 mod 2 equals 1)

^ (exponentiation , I ^ J equivalent to I^J)

Predicate ‘is’
Used to perform calculations, Eg,

?- X is 3 + 5. (query)

X = 8 (Prolog’s response)

?- A is 3,
    B is 2,
    C is 6,
    E is ((A * B + C) mod 5).

E = 2 (Prolog’s response)
Calculation using Prolog - an example

/* Program to convert seconds into days, hours, minutes and seconds. */

Convert(Input, Days, Hours, Minutes, Seconds) : -
    Seconds is ( Input  mod  60),
    MinutesTot is ( Input  //  60),
    Minutes is ( MinutesTot  mod  60),
    HoursTot is ( MinutesTot  //  60),
    Hours is (HoursTot mod 24),
    Days is ( HoursTot  //  24).

?- Convert(90061, Days, Hours, Minutes, Seconds).

Days = 1
Hours = 1
Minutes = 1
Seconds = 1
Recursion

Defining something in terms of itself with some boundary condition, eg,
The \( n \)th natural number is the \((n-1)\)th natural number plus 1. The first natural number is 1.

Rules can be defined recursively in Prolog
We’ve already defined the length of a list recursively:

The length of a list is 1 more than the length of its tail (which itself is a list).

The length of an empty list is 0 (boundary condition)

\[
\text{length([ ]}, 0).
\]

\[
\text{length([H|T],N) :- length(T,M) , N is M + 1.}
\]

Example: Rules to define ancestry

\[
\text{ancestor_of ( Older, Younger):- parent_of( Older, Younger).}
\]

\[
\text{ancestor_of ( Older, Younger):- parent_of( Older, In_between), ancestor_of (In_between, Younger).}
\]
More examples of Recursion

The *FACTORIAL* of a (positive) integer *N* is defined as follows

\[ N! = 1 \times 2 \times \ldots \times N \]

We wish to define a predicate `fact(N,X)` which shall be true precisely when

\[ X = N! \]

Thus, if we were to make the query

\[ ?- \ text{fact}(4,X). \]

then Prolog’s unification mechanism would instantiate *X* to the value 24 in order for the query to succeed.
Consider the following important property of the factorial function

\[ N! = N \times (N - 1)! \]

This relation enables us to define the factorial function in terms of itself (i.e., recursively).

Thus we can calculate the factorial of any positive integer \( N \) provided we can calculate the factorial of its predecessor \( N - 1 \).

But the factorial of \( N - 1 \) can be calculated in an entirely analogous way by first calculating the factorial of its predecessor \( N - 2 \).

This process can be repeated until we reach a number whose factorial is already known (or can be calculated by some other means).

In this case we know, trivially, that the factorial of \( 1 \) is \( 1 \).
We might try to express the above (recursive) property as the following Prolog clause

\[ \text{fact}(N,X) :- \text{fact}(N-1,Y), X = N \times Y. \]

There are two arithmetic expressions in this clause that need to be evaluated.

The second is easily dealt with by using the \texttt{is} operator in place of \texttt{=}

In order to evaluate the first expression, we introduce another variable \( M \) and instantiate it to the value of the expression with an extra sub-goal as follows:

\[ \text{fact}(N,X) :- M \text{ is } N - 1, \text{fact}(M,Y), X \text{ is } N \times Y. \]
Our final definition is then

\[
\text{fact}(1,1).
\]

\[
\text{fact}(N,X) :\ M \text{ is } N - 1, \ \text{fact}(M,Y), \ X \text{ is } N \times Y.
\]

The first clause is called the \textit{TERMINATING CONDITION}. It is necessary to prevent the recursion from continuing on forever.

The terminating condition must appear before the second clause in the database.

If it doesn't, then the second clause will always be tried first (creating an infinite number of sub-goals).
Now consider what happens when Prolog attempts to prove the query

?- fact(3,Y).

We proceed as follows, again viewing the resolution process as *GOAL DIRECTED SEARCH*

We attempt to *UNIFY* the (current) *GOAL* with the *HEAD* of the first clause in the database that will match (in this case the second clause in the database).

?- fact(3,Y).

fact(N,X) :- M is N - 1, fact(M,Y), X is N * Y.

Unfortunately, this causes a problem with conflicting variable names.

In particular, the variable *Y* in the goal is *NOT* the same *Y* that appears in the other clause shown above.
In order to avoid such conflicts, we employ the following convention:

- Each time a clause from the database is invoked, as above, we associate a (different) number with it.
- All variables appearing in the clause are subscripted with that number.
- Variables with the same name but a different subscript are thus \textit{DISTINCT} variables.

Thus we rewrite the above as

\[ \texttt{?- fact(3,Y).} \]
\[ \texttt{fact(N1,X1) :- M1 is N1 - 1 , fact(M1,Y1) , X1 is N1 \times Y1.} \]

A match will occur if we \textit{INSTANTIATE}

\[
\begin{align*}
\text{n1} & \text{ to } 3 \\
\text{x1} & \text{ to } Y
\end{align*}
\]

to give the clause

\[ \texttt{fact(3,Y):- M1 is 3 - 1 , fact(M1,Y1) , Y is 3 \times Y1.} \]

This creates 3 sub-goals that need to be satisfied in order for the original goal to be true.
The first is satisfied by instantiating $M_1$ to 2

To prove the sub-goal

? - fact(2,Y1).

we attempt to UNIFY it with the HEAD of the first clause in the database that will match (which is again the second clause) as follows

? - fact(2,Y1).

\[
\text{fact}(N_2,X_2) \iff M_2 \text{ is } N_2 - 1 \land \text{fact}(M_2,Y_2) \land X_2 \text{ is } N_2 \times Y_2.
\]

Note that all the variables in the clause are now subscripted with a 2 since this is the second time the clause has been invoked.

A match will occur if we INSTANTIATE

\[
N_2 \text{ to } 2 \\
X_2 \text{ to } Y_1
\]

to give the clause

\[
\text{fact}(2,Y_1) \iff M_2 \text{ is } 2 - 1 \land \text{fact}(M_2,Y_2) \land Y_1 \text{ is } 2 \times Y_2.
\]

This creates 3 further subgoals [level 2] to be proved

The first will be satisfied by instantiating $M_2$ to 1
The second [level 2] sub-goal then becomes

? - fact(1, Y2)

and this will be satisfied if we instantiate Y2 to 1

[so that it matches the first clause in the database – i.e. the terminating condition]

The third [level 2] sub-goal thus becomes

? - Y1 is 2 * 1

which will be satisfied if we instantiate Y1 to 2

The next (and final) outstanding sub-goal to prove is the third sub-goal which becomes

? - Y is 3* 2

This is satisfied if we instantiate Y to 6 and thus, with no more outstanding sub-goals left to prove, our original goal is proved.
More examples with Lists

Testing for list membership

Example:

Is lisa in the list [homer, marge, bert, lisa, maggie]?

In Prolog, we can express this problem using a relationship member

\[
\text{member (X, [X | Rest]).}
\]

If X matches the head of the list, then predicate member is true.

If not, we test for membership of X in Rest

Thus the membership rules can be written as

\[
\text{member (X, [X | _].}
\]

\[
\text{member (X, [\_ | Rest]):- member (X, Rest).}
\]

Adding an item to a list

The new item is put in front of the list and becomes the new head

\[
\text{add(X, L, [X|L]).}
\]

Write a query to add item ‘a’ to the list [b,c,d]
Deleting an item from a list

Deleting item \( X \) from list \( L \) can be programmed as a relation

\[
\text{del}(X, L, L1)
\]

where \( L1 \) is the list \( L \) with item \( X \) removed

Two cases:

1. \( X \) is the head of the list \( L \)
2. \( X \) is in the tail of \( L \)

Taken care of by the clauses

\[
\text{del}( X, [X|Tail], \text{Tail} ).
\]

\[
\text{del}( X, [Y|Tail], [Y|\text{NewTail}] ) :-
\]

\[
\text{del}( X, \text{Tail}, \text{NewTail} ).
\]

Examples:

?- del(a, [a, b, c, d], L).
\[
L = [b, c, d]
\]

?- del(c, [a, b, c, d], L).
\[
L = [a, b, d]
\]
Joining two lists (Concatenating)

\textbf{conc( L1, L2, L3)}

where \quad L3 = L1 + L2, \; \text{eg,}

? - conc([a,b], [c,d], NewList).
NewList = [a,b,c,d]

Two cases
1. First argument an empty list
\textbf{conc( [ ], L, L)}

2. First argument has a head and a tail
\textbf{conc( [ X | L1], L2, [X | L3] ) :-}
\textbf{conc( L1, L2, L3).}

\begin{array}{c|c|c}
[X | L1] & X & L1 \\
\hline
 & L2 & \\
\hline
[X | L3] & X & L3
\end{array}