Modern Programming Languages

Lecture 22-26

PROLOG - Programming in Logic

An Introduction

PROLOG stands for PROgramming in LOGic and was design in 1975 by Phillippe Roussell. It is a declarative programming language and is based upon Predicate Calculus. A program in PROLOG is written by defining predicates and rules and it has a built-in inference mechanism. Unfortunately, there is no effective standardization available.

PROLOG Paradigm

As mentioned earlier, PROLOG draws inferences from facts and rules. It is a declarative language in which a programmer only specifies facts and logical relationships. It is non-procedural. That is we only state "what" is to be done and do not specify "how" to do it. That is, we do not specify the algorithm. The language just executes the specification. In other words, program execution is carried out as a theorem proving exercise. It is similar to SQL used in databases with automated search and the ability to follow general rules.

One of the main features of this language is its ability to handle and process symbols. Hence it is used heavily in AI related applications. It is an interactive (hybrid compiled/interpreted) language and its applications include expert systems, artificial intelligence, natural language understanding, logical puzzles and games.

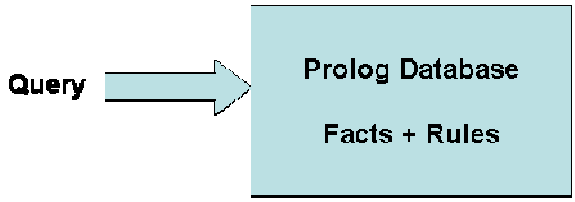
PROLOG Programming

PROLOG programming follows the following steps:

- Declaring some facts about objects and their relationships
- Defining some rules about objects and their relationships
- Asking questions about objects and their relationships

The PROLOG Programmer loads facts and rules into the database and makes queries to the database to see if a fact is in the database or can be implied from the facts and rules therein.

This is depicted in the following diagram.



Facts

- Facts are used to represent unchanging information about objects and their relationships.
- Only facts in the PROLOG database can be used for problem solving.
- A programmer inserts facts into the database by typing the facts into a file and loading (consulting) the file into a running PROLOG system.

Queries

Queries are used to retrieve information from the database. A query is a pattern that PROLOG is asked to *match* against the database and has the syntax of a compound query. It may contain variables. A query will cause PROLOG to look at the database, try to find a match for the query pattern, execute the body of the matching head, and return an answer.

Logic Programming

Logic programming is a form of declarative programming. In this form, a program is a collection of *axioms* where each axiom is a *Horn clause* of the form:

H :- B1, B2, ..., Bn.

where *H* is the *head term* and *Bi* are the *body terms*, meaning *H* is true if all *Bi* are true. A user of the program states a *goal* (a theorem) to be proven. The logic programming system attempts to find axioms using *inference steps* that imply the goal (theorem) is true. The basic proof technique is Modus Ponens i.e.

A -> B (If A is true then B is also true but it does not mean that if B is true then A is also true)

If Fact is given that

A is true

Then

B is also true

Resolution

To deduce a goal (theorem), the logic programming system searches axioms and combines sub-goals. For this purpose we may apply forward (from fact to goal) or backward (from goal to fact) chaining. For example, given the axioms:

C :- *A*, *B*.
D :- *C*.

Given that A and B are true, *forward chaining* (from fact to goal) deduces that *C* is true because *C* :- *A*, *B* and then *D* is true because *D* :- *C*.

Backward chaining (from goal to fact) finds that *D* can be proven if sub-goal *C* is true because *D* :- *C*. The system then deduces that the sub-goal is *C* is true because *C* :- *A*, *B*. Since the system could prove *C* it has proven *D*.

Prolog Uses backward chaining because it is more efficient than forward chaining for larger collections of axioms.

Examples

Let us look at some examples of facts, rules, and queries in Prolog.

Facts

Following table shows some example of facts.

English	PROLOG
"A dog is a mammal"	isa(dog, mammal).
"A sparrow is a bird"	isa(sparrow, bird).

Rules

Here are some examples of rules.

English	PROLOG
"Something is an animal	animal(X) :- isa(X, bird).
if it is a mammal or a bird"	animal(X) :- isa(X,mammal).

Queries

An now some queries:

English	PROLOG
"is a sparrow an animal?" answer: "yes"	?- animal(sparrow). yes
"is a table an animal?" answer: "no"	?- animal(table). no
"what is a dog?" answer: "a mammal"	?- isa(dog, X). X = mammal

----------END OF LECTURE 22----------

PROLOG syntax

Constants

There are two types of constants in Prolog: atoms and numbers.

Atoms:

- Alphanumeric atoms - alphabetic character sequence starting with a lower case letter. Examples: apple a1 apple_cart
- Quoted atoms - sequence of characters surrounded by single quotes. Examples: 'Apple' 'hello world'
- Symbolic atoms - sequence of symbolic characters. Examples: & < > * - + >>
- Special atoms. Examples: ! ; [ ] {}

Numbers:

Numbers include integers and floating point numbers.

Examples: 0 1 9821 -10 1.3 -1.3E102.

Variable Names

In Prolog, a variable is a sequence of alphanumeric characters beginning with an upper case letter or an underscore. Examples: Anything _var X _

Compound Terms (structures)

A compound term is an atom followed by an argument list containing terms. The arguments are enclosed within brackets and separated by commas.
Example: isa(dog, mammal)

Comments

In Prolog % is used to write a comment just like // in C++. That is after % everything till the end of the line is a comment.

Important points to note

- The names of all relationships and objects must begin with a lower case letter.
  - For example ***studies, ali, programming***
- The relationship is written first and the objects are enclosed within parentheses and are written separated by commas.
  - For example ***studies(ali, programming)***
- The full stop character '.' must come at the end of a fact.
- Order is arbitrary but it should be consistent.

An Example Prolog Program

This program has three facts and one rule. The facts are stated as follows:

```
rainy(columbo).
rainy(ayubia).
cold(ayubia).
```

The rule is:

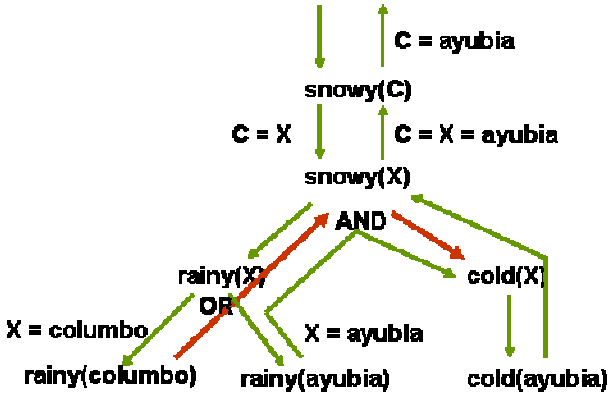
```
snowy(X) :- rainy(X), cold(X).
```

Some queries and responses:

```
?- snowy(ayubia).
yes

?- snowy(columbo).
no

?- snowy(lahore).
No

?- snowy(C).
C = ayubia
```

Because **rainy(ayubia)** and **cold(ayubia)** are sub-goals that are both true facts in the database, **snowy(X)** with **X=columbo** is a goal that fails, because **cold(X)** fails, triggering *backtracking.*

This inference mechanism based upon back tracking is depicted in the following diagram:



Logic Representation : Propositional Calculus

Propositional Logic is a set of Boolean statements. It involves logic connectives such as $\neg \wedge \vee \Leftrightarrow \Rightarrow$.

Example – a family

If we had the following atomic statements:

p: Ahmed is father of Belal
q: Ahmed is brother of Aslam
r : Aslam is uncle of Belal

Then the following table shows the meanings in English for some logical statements in propositional calculus:

Statement	Logical meaning
$\neg p$	Ahmed is not father of Belal
$p \vee q$	Ahmed is father of Belal or Ahmed is brother of Aslam
$p \wedge q \Rightarrow r$	If Ahmed is father of Belal and brother of Aslam then Aslam is uncle of Belal

Statements in predicate calculus can be mapped almost directly into prolog as shown below:

Predicate Calculus

- Predicates
  - man(Ahmed)
  - father(Ahmed, Belal)
  - brother(Belal, Chand)
  - brother(Chand, Delawar)
  - owns(Belal, car)
  - tall(Belal)
  - hates(Ahmed, Chand)
  - family()
- Formulae
  - $\forall X,Y,Z(man(X) \wedge man(Y) \wedge father(Z,Y) \wedge father(Z,X) \Rightarrow brother(X,Y))$
- Variables
  - X, Y and Z
- Constants
  - Ahmed, Belal, Chand, Delawar and car

Prolog code

```
Predicates
        man(symbol)
        father(symbol, symbol)
        brother(symbol, symbol)
        owns(symbol, symbol)
        tall(symbol)
        hates(symbol, symbol)
        family()
Clauses
        man(ahmed).
        father(ahmed, belal).
        brother(ahmed, chand).
        owns(belal, car).
        tall(belal).
        hates(ahmed, chand).
        family().

        brother(X,Y):-
              man(X),
              man(Y),
              father(Z,Y),
              father(Z,X).
Goal
        brother(ahmed,belal).
```

Sections of Prolog Program

Predicates
Predicates are declarations of relations or rules. They are just like function prototypes (declaration). A predicate may have zero or more arguments. Example of predicates is:

```
man(symbol)
family()
a_new_predicate (integer, char)
```

Clauses
Clauses are definition(s) of Predicate sentence or phrase. It has two types: rules and facts. A rule is a function definition. It may have 0 or more conditions. A fact has all parameters as constants and it cannot contain any condition.

Example – Fact:

```
brother(ahmed, chand).
```

Saying that ahmed and chand are brothers.

Example – Rule:

```
brother(X,Y):-
        man(X),
        man(Y),
        father(Z,Y),
        father(Z,X).
```

It says that two symbols, X and Y, are brothers if X is a man and Y is a man and their father is same.

Goal

- Goal is the objective of the program.
- There can be only and exactly one instance of the goal in a program.
- It is the only tentative section of the program.
- It may be thought as the main() function of prolog.
- The truth value of the goal is the output of the program.
- Syntactically and semantically, it is just another **clause.**
- It may be empty, simple (one goal), or compound (sub-goals).

Examples

1. brother(X,chand). % In this case the output will be the name of Chand's brother.
2. brother(ahmed,chand).% The output will be true or false.
3. brother(X,chand), father(X, Y). % The output will be Chand's and his children.

----------END OF LECTURE 23----------

Prolog Variables

Prolog variable are actually constant placeholders and NOT really variables. That is, a value to a variable can be bound only once and then it cannot be changed. These variables are loosely typed. That is, type of the variable is not defined at compile time. As mentioned earlier, a variable name starts with capital letter or underscore. Here are some examples of variable names:

- brother(ahmed, Ahmed) % ahmed is a constant whereas Ahmed is a variable
- brother(ahmed, _x) % _x is a variable
- Brother(ahmed, X) % X is a variable

Anonymous variable:
The _ is a special variable. It is called the anonymous variable. It is used as a place holder for a value that is not required. For example _ is used as a variable in the following clause:

brother(ahmed, _)

Other Syntactic Elements

Prolog has a number of other syntactic elements. Some of these are listed below:

- Special predicates
  - cut <or> !
  - not(predicate)
- Predefined predicates
  - write(arg1[,arg2[,arg3[,arg4[…..]]]]) % for output
  - nl % new line
  - readint(var) % read integer
  - readchar(var) % read char
  - readln(var) % read line into var
  - … many more related to i/o, arithmetic, OS etc
- Operators
  - Arithmetic
    - + - * div / mod
  - Relational
    - < <= => > = <> ><

PROLOG Lists

In Prolog, a list is a sequence of terms of the form

[t1, t2, t3, t4, ..., tn]

where term ti is the ith element of the list

Examples:
1. [a,b,c] is a list of three elements a, b and c.

2. [[a,list,of,lists], and, numbers,[1,2,3]]
is a four element list, the first and last are themselves lists.

3. [ ] is the 'empty list'. It is an atom not a list data type. This is a fixed symbol. And it will always have the same meaning that's why it is treated as atom not list.

Working with Lists

1. Lists are used to build compound data types.
2. A list can have elements of arbitrary type and size.
3. There is no size limits.
4. Representation in prolog is very concise (short).
5. **head – tail separator**
    a. The vertical bar '|' is used to separate the head and tail of a list as shown below:
        [<head>|<tail>]
    b. In this case, head is an element whereas tail is list of the same sort.
    c. The | has dual purpose in Prolog; it is used for list construction as well as list dismantling.

Here as some examples of lists:

```
[a, bc, def, gh, i]   % a simple list of five elements
[ ]                   % an empty list
[1,2,3]               % a simple list of three elements
[1, 2 | [3] ]         % same as above; a list of three elements with 1 and 2 as head and
                      % [3] as tail. When you join head and tail you get [1 2 3].
[1 | [2, 3] ]         % another way of writing the same thing again. This time the head
                      % is only 1 and the tail is [2 3]. Concatenation of these two gives
                      % [1 2 3].
[1, 2, 3 | [ ] ]      % same thing again but with a different head and tail formation
```

Recursion is the basic tool for writing programs in Prolog. Let us now look at some example programs in Prolog using lists.

Example 1: Calculating the length of a list by counting the first level elements

```
Domains
        list = Integer *
Predicates
        length (list, integer)
Clauses
        length([],0).                       %length of an empty list is zero

        length ([_|Tail], Len):-
                Length (Tail, TailLen),     %getting the tail length
                Len = TailLen + 1.          %the length of list is 1 plus the
                                            % length of tail
Goal
        X = [1,2,3],
        length(X,Len).
```

Example 2: Finding the last and second last element in a list

```
lastElement(X,[X]).                         % if the list has only one element
                                            % then it is the last element
lastElement(X,[_|L]) :- lastElement(X,L).   % recursive step

last_but_one(X,[X,_]).                      % similar to the above example
last_but_one(X,[_,Y|Ys]) :- last_but_one(X,[Y|Ys]).
```

----------END OF LECTURE 24----------

Example 3: Eliminate consecutive duplicates of list elements.

If a list contains repeated elements they should be replaced with a single copy of the element. The order of the elements should not be changed.

Example:

```
?- compress([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X).
X = [a,b,c,a,d,e]
```

Here is the code for this function:

```
compress([],[]).                    % there is nothing to be compressed in
compress([X],[X]).                  % an empty list or a list with only one
                                    % element

compress([X,X|Xs],Zs) :-            % if the first element is repeated, copy
        compress([X|Xs],Zs).        % it once in the new list
```

```
compress([X,Y|Ys],[X|Zs]) :-          % inspect the rest of the list and
        X \= Y,compress([Y|Ys],Zs). % repeat the process recursively
```

Sorting

Remember, in Prolog, we only specify what we want and not how to do it. Here is a classical example of this specification. In the following program segment, in order to sort a list, we only specify what is meant by sorting and not how to do it.

```
sorted ([ ]).                % an empty list or list with only one element
sorted ([X]).                % is already sorted
sorted ([X, Y | list]) :-    % recursive step
        X <= Y,              % a list is sorted if every element in the list
        sorted ([Y | list]). % is smaller than the next element in the list.
```

List Membership

```
member(X, [X|T]).            % X is a member of the list if it is the first
                             % element

member(X, [H|T]) :- member(X, T).
                             % otherwise recursively check it in the rest of
                             % the list
```

subset

```
subset( [ ], Y ).            % The empty set is a subset of every set.
subset( [ A | X ], Y ) :-    % recursive step
        member( A, Y ), subset( X, Y ).
```

intersection

```
% Assumes lists contain no duplicate elements.
intersection( [ ], X, [ ] ).
intersection( [ X | R ], Y, [ X | Z ] ) :-
                member( X, Y ), !, intersection( R, Y, Z ).
intersection( [ X | R ], Y, Z ) :- intersection( R, Y, Z ).
```

union

```
union( [ ], X, X ).
union( [ X | R ], Y, Z ) :- member( X, Y ), !, union( R, Y, Z ).
union( [ X | R ], Y, [ X | Z ] ) :- union( R, Y, Z ).
```

Puzzles- Who owns the zebra

According to an e-mail source, this puzzle originated with the German Institute of Logical Thinking, Berlin, 1981. [It's possible.]

1. There are five houses.
2. Each house has its own unique color.
3. All house owners are of different nationalities.
4. They all have different pets.
5. They all drink different drinks.
6. They all smoke different things.
7. The English man lives in the red house.
8. The Swede has a dog.
9. The Dane drinks tea.
10. The green house is on the left side of the white house.
11. In the green house, they drink coffee.
12. The man who smokes cigarette has birds.
13. In the yellow house, they smoke pipe.
14. In the middle house, they drink milk.
15. The Norwegian lives in the first house.
16. The man who smokes cigar lives in the house next to the house with cats.
17. In the house next to the house with the horse, they smoke pipe.
18. The man who smokes hukka drinks soda.
19. The German smokes bedi.
20. The Norwegian lives next to the blue house.
21. They drink water in the house that is next to the house where they smoke cigar.
22. Who owns the zebra?

Try solving this puzzle on your own. Now imagine writing a computer program to solve it. Which was more difficult?

This is an example of a completely specified solution which doesn't appear to be specified at all. The constraints are such that the answer is unique, but they are stated in such a way that it is not at all obvious (to this human, at least) what the answer is.

In Prolog, we could express each of these specifications, then let Prolog's search strategy (database search engine, automated theorem prover) search for a solution. We don't have to worry about *how* to solve the problem -- we only have the specify *what* is to be solved.

```
%  -*-prolog-*-
%
%  The Zebra puzzle:  Who owns the zebra?
%
%  According to a recent e-mail source, this puzzle originated with
%  the German Institute of Logical Thinking, Berlin, 1981. [It's
possible.]
%  The terms of the puzzle are included as comments in the code below.
%
%  Solution by Jonathan Mohr (mohrj@augustana.ca)
%  Augustana University College, Camrose, AB, Canada  T4V 2R3


%  Invoke this predicate if you just want to see the answer to the
%  question posed at the end of the puzzle.
solve :-
    solve(_).


%  Invoke this predicate (with a variable parameter) if you want to
%  see a complete solution.
solve(S) :-

% There are five houses.
% Each house has its own unique color.
% All house owners are of different nationalities.
% They all have different pets.
% They all drink different drinks.
% They all smoke different cigarettes.

% (The constraints that all colors, etc., are different can only be
% applied after all or most of the variables have been instantiated.
% See below.)

% S = [[Color1, Nationality1, Pet1, Drink1, Smoke1] |_]
% The order of the sublists is the order of the houses, left to right.
    S = [[C1,N1,P1,D1,S1],
         [C2,N2,P2,D2,S2],
         [C3,N3,P3,D3,S3],
         [C4,N4,P4,D4,S4],
         [C5,N5,P5,D5,S5]],

% The English man lives in the red house.
    member([red, 'English man', _, _, _], S),
% The Swede has a dog.
    member([_, 'Swede', dog, _, _], S),
% The Dane drinks tea.
    member([_, 'Dane', _, tea, _], S),
% The green house is on the left side of the white house.
    left_of([green |_], [white |_], S),
% In the green house, they drink coffee.
    member([green, _, _, coffee, _], S),
% The man who smokes cigarette has birds.
    member([_, _, birds, _, cigarette], S),
% In the yellow house, they smoke pipe.
    member([yellow, _, _, _, pipe], S),
% In the middle house, they drink milk.
```

```
    D3 = milk,
% The Norwegian lives in the first house.
    N1 = 'Norwegian',
% The man who smokes cigar lives in the house next to the house with
cats.
    next_to([_, _, _, _, cigar], [_, _, cats |_], S),
% In the house next to the house with the horse, they smoke pipe.
    next_to([_, _, _, _, pipe], [_, _, horse |_], S),
% The man who smokes hukka drinks soda.
    member([_, _, _, soda, hukka], S),
% The German smokes bedi.
    member([_, 'German', _, _, bedi], S),
% The Norwegian lives next to the blue house.
    next_to([_, 'Norwegian' |_], [blue |_], S),
% They drink water in the house that is next to the house
% where they smoke cigar.
    next_to([_, _, _, water,_], [_, _, _, _, cigar], S),

%
% The puzzle is so constrained that the following checks are not really
% required, but I include them for completeness (since one would not
% know in advance of solving the puzzle if it were fully constrained
% or not).
%
% Each house has its own unique color.
    C1 \== C2, C1 \== C3, C1 \== C4, C1 \== C5,
    C2 \== C3, C2 \== C4, C2 \== C5,
    C3 \== C4, C3 \== C5, C4 \== C5,
% All house owners are of different nationalities.
    N1 \== N2, N1 \== N3, N1 \== N4, N1 \== N5,
    N2 \== N3, N2 \== N4, N2 \== N5,
    N3 \== N4, N3 \== N5, N4 \== N5,
% They all have different pets.
    P1 \== P2, P1 \== P3, P1 \== P4, P1 \== P5,
    P2 \== P3, P2 \== P4, P2 \== P5,
    P3 \== P4, P3 \== P5, P4 \== P5,
% They all drink different drinks.
    D1 \== D2, D1 \== D3, D1 \== D4, D1 \== D5,
    D2 \== D3, D2 \== D4, D2 \== D5,
    D3 \== D4, D3 \== D5, D4 \== D5,
% They all smoke different cigarettes.
    S1 \== S2, S1 \== S3, S1 \== S4, S1 \== S5,
    S2 \== S3, S2 \== S4, S2 \== S5,
    S3 \== S4, S3 \== S5, S4 \== S5,

% Who owns the zebra?
    member([_, Who, zebra, _, _], S),
    write('The '), write(Who), write(' owns the zebra.\n').

% Or, replace the last line by:
%   format("The ~w owns the zebra.", Who).

left_of(L1, L2, [L1, L2 |_]).
left_of(L1, L2, [_| Rest ]) :- left_of(L1, L2, Rest).

next_to(L1, L2, S) :- left_of(L1, L2, S).
next_to(L1, L2, S) :- left_of(L2, L1, S).
```

Expert system

One of the main application domains for Prolog has been the development of expert systems. Following is an example of a simple medical expert system:

```
% clauses for relieves(Drug, Symptom). That is facts about drugs and relevant symptoms

relieves(aspirin, headache).
relieves(aspirin, moderate_pain).
relieves(aspirin, moderate_arthritis).
relieves(aspirin_codine_combination, severe_pain).
relieves(cough_cure, cough).
relieves(pain_gone, severe_pain).
relieves(anti_diarrhea, diarrhea).
relieves(de_congest, cough).
relieves(de_congest, nasal_congestion).
relieves(penicilline, pneumonia).
relieves(bis_cure, diarrhea).
relieves(bis_cure, nausea).
relieves(new_med, headache).
relieves(new_med, moderate_pain).
relieves(cong_plus, nasal_congestion).

% now we have clauses for side effects in the form of aggravates(Drug, Condition).

aggravates(aspirin, asthma).
 aggravates(aspirin, peptic_ulcer).
aggravates(anti-diarrhea, fever).
aggravates(de_congest, high_blood_pressure).
aggravates(de_congest, heart_disease).
aggravates(de_congest, diabetes).
aggravates(de_congest, glaucoma).
aggravates(penicilline, asthma).
aggravates(de_congest, high_blood_pressure).
aggravates(bis_cure, diabetes).
aggravates(bis_cure, fever).
```

```
% now some rules for prescribing medicine using symptoms and side effects
should_take(Person, Drug) :-
        complains_of(Person, Symptom),
        relieves(Drug, Symptom),
        not(unsuitable_for(Person, Drug)).

unsuitable_for(Person, Drug) :-
        aggravates(Drug, Condition),
        suffers_from(Person, Condition).

% we now have some specific facts about a patient named Ali
complains_of(ali, headache).
suffers_from(ali, peptic_ulcer).

% now the query about the proper medicine and the answer by the expert system
?- should_take(ali, Drug).
Drug = new_med;
```

As mentioned earlier, Prolog has been used to write expert systems and expert system shells. Because of its inference mechanism and independence of rules and clauses it is relatively easy to achieve such task as compared to any imperative language.

Conclusions

Prolog is a declarative programming language where we only specify what we need rather than how to do it. It has a simple concise syntax with built-in tree formation and backtracking which generates readable code which is easy to maintain. It is relatively case and type insensitive and is suitable for problem solving / searching, expert systems / knowledge representation, language processing / parsing and NLP, and game playing. Efficiency is definitely is negative point. While writing programs one has to be aware of the left recursion, failing to do that may result in infinite loops. It has a steep learning curve and suffers from lack of standardization.