Meta Logical Predicates

- Meta predicates are those which are programmed using user's and Prolog system's defined predicates.
- Meta program treats other program as data.
- Meta rules enhance general purpose control structures and are not a control structures by themselves.
- We will describe some of important and useful meta predicates.
- These may be implementation specific
Basic predicates

\textbf{var}(X):\ succeeds if X is a variable
\textbf{nonvar}(X): succeeds if X is not a variable.
\textbf{number}(X): succeeds if X is integer or real.
\textbf{atomic}(X): succeeds if X is a number, symbol or string.
\textbf{integer}(X): succeeds if X is integer.
\textbf{compound}(X): succeeds if X is a compound term. A compound term is of the form \( f(t_1, \ldots, t_n) \), where \( f \) is n-place functor and \( t_1, \ldots, t_n \) are simple terms. Simple term in Prolog is either a constant or a variable.
Functor and Arg predicates

functor(Term, F, N): succeeds if Term is a compound term with name of function as F having arity N.

Examples
- functor(max(4,5), F, N) succeeds with bindings F = max and N = 2
- functor(father(john, mike), father, N) succeeds with N = 2
- functor(T, min, 2) succeeds and T is unified with min(_, _)
- functor(f(4, 5, 6), f, 3) succeeds & functor(g(4), g, 2) fails.

arg(N, Term, Arg): succeeds by unifying Nth argument of Term with Arg.

Examples
- arg(1, f(ram, shyam), ram) succeeds.
- arg(2, f(ram, shyam), A) succeeds with A = shyam.
Note that predicate \textit{functor} and \textit{arg} fail if goal does not unify with appropriate fact or if the type restrictions are violated.

\textbf{Term} = \ldots [F|L] : succeeds if F is unified with function name of Term and L is unified with the list of arguments of F.

\textbf{Examples}

- father(ram, shyam) = \ldots [F | L] succeeds with bindings F = father and L = [ram, shyam].
- father(ram, shyam) = \ldots [father | [ram, shyam]] succeeds.
**Clause predicate**

`clause(Head, Body):` succeeds, if *Head* and *Body* are matched with the head and body of an existing rule in the Prolog program. It is very important and useful while developing interpreters.

1. `append([], X, X).`  
2. `append([H|X], Y, [H|Z]) :- append(X, Y, Z).`

?- clause(append(A, B, [2,3]), D).

\[\begin{align*}
\text{(1)} & : \{A = [], B = [2, 3], D = true\} \\
\text{(2)} & : \{A = [H|X], H = 2, Z = [3], \quad D = \text{append}(X,B,[3])\}
\end{align*}\]
Call predicate

**call(X):** succeeds if goal X succeeds and fails if goal X fails.

- At the first glance, this predicate seems to be redundant because effect of, for example, `call(member(3, L))` and `member(3, L)` is same.
- It is not same.
  - When we have to construct goal X dynamically during execution of a program, then we *call* predicate with variable goal as an argument is used whose value keeps on changing at run time.
- The *call* predicate is also widely used while writing various interpreters.
**Findall predicate**

```prolog
findall(X, G, L): succeeds by constructing a list L of all the objects X for which the goal G is satisfied.
```

**Examples**

?- findall(X, son(X, john), L)  - L is a list of all X such that son(X, john) is satisfied.

?- findall(X, son(mike, john), L)  - fails

?- findall([X,Y], son(X, Y), L)  - L is a list of list [X,Y] such that son(X,Y) is satisfied.
Input / Output predicates

- Input/output predicates are not part of pure Prolog.
- \( \text{read}(X) \): succeeds by getting the value of \( X \) from terminal.
- \( \text{write}(X) \): succeeds by printing the value of \( X \) on terminal.
- \( \text{nl} \): succeeds by creating a new line.
- \( \text{writeln}(X) \): succeeds after writing \( X \) on terminal and creating new line.
- \( \text{listing} \): succeeds by listing all the predicates in the current database.
**Prolog Program**

**Example:** Define a predicate $ground(X)$ which succeeds if $X$ is a ground term (i.e., a term without variables).

**Solution:** We can write program for $ground(X)$ in Prolog using some system defined predicates as:

```
ground(T) :- nonvar(T), atomic(T), !.
ground(T) :- nonvar(T), compound(T), functor(T, _, N), ground1(T, N).
ground1(T, N) :- N > 0, arg(N, T, A), ground(A), N1 is N – 1, ground1(T, N1).
ground1(T, 0).
```

Query: `?- ground(4)` and `?- ground(f(2, 3))` succeeds
Interactive Interpreter for Prolog

- One can write interactive meta interpreter that accepts commands from terminal and executes them using Prolog interpreter and terminates its execution when 'exit' command is given.

  
  ```prolog
  interpreter :- interpreter_prompt, read(Goal),
    interpreter(Goal). \(1\)
  interpreter(exit) :- !.
  interpreter(Goal) :- ground(Goal), !,
    solve_ground(Goal), interpreter. \(3\)
  interpreter(Goal) :- solve(Goal), interpreter. \(4\)
  interpreter_prompt :- writeln('Next Command').
  ```
solve_ground(Goal) :- call(Goal), !, writeln('Yes'). (5)
solve_ground(Goal) :- writeln('No'). (6)
solve(Goal) :- call(Goal), writeln(Goal), fail. (7)
solve(Goal) :- writeln('No more Solutions'). (8)

**Interpretation of the rules:**

- Rule(1) displays message 'Next Command' and asks user to input the *Goal* to be interpreted. The Goal is passed as an argument of predicate *interpreter*.
- Rule(2) states that if Goal is *exit*, then quit the program.
- Rule(3) checks whether *Goal* is ground or not. If *Goal* is ground, then solve ground goal using rules (5) and (6) otherwise rule(4) is tried and *Goal* is solved using rules (7) and (8). The predicate *interpreter* is again initiated and the process repeats interaction with user till 'exit' goal is given.
Meta Interpreter

- *Meta interpreter* for a language is an interpreter or a meta program that is written in a language itself. *Meta program* treats other programs as data.
- Prolog has a powerful features for writing meta programs because Prolog treats program and data both as terms.
- One can write meta interpreters for various applications.
Applications of Interpreter

Following are various applications for which meta interpreters can be developed.

- exploring different execution strategies based on breadth first, combination of depth first and breadth first searches etc.
- generating proof trees
- developing expert system shell, editor etc.
- debugging programs i.e., to identifying errors in a program. (Bug may be of any type viz., program is returning false result, fails to return some true solution or fails to terminate etc.)
Meta Interpreter for Prolog Language

Let us write simple meta interpreter that simulates the computational model of Prolog program using depth first (DF) control strategy.

/* execute(G) – succeeds if G succeeds with respect to program being interpreted. */

execute([true]) :- !.
(1)
execute([P]) :- clause(P), execute([P]), execute([P]), execute([P]).
Query for Meta Interpreter

Query:  
?- execute([mem (X,[3,4])]).

Answer:  X = 3;  X = 4
Interpreter for Building Proof Trees

- Add an extra parameter for storing proof tree.
- The basic relation is \( \text{gen}(G, \ Proof) \), where \( \Proof \) is a proof tree for solving a goal \( G \).
- Let us represent proof trees by a structure \( (G \Rightarrow \Proof) \), where \( \Proof \) is a conjunction of the branches proving the goal \( G \).
- It must be noted that 'G' has to be ground.
/* gen(G, Proof) - succeeds if G succeeds with respect to program being interpreted and Proof is unified with its proof structure. */

gen([true], true). (1)
gen([G], (G => P)) :- clause(G, Body), gen([Body], P). (2)
gen([G1|T], (P1, T1)) :- gen(G1, P1), gen(T, T1). (3)

Interpretation of above rules:

– Rule (1) states that if a goal is true, then proof tree is represented by an atom true.
– Rule (2) builds an actual proof tree structure (G => P) for the goal G, where P is a proof recursively built by solving the body of G.
– Rule (3) states that a proof tree of a conjunctive goals stored in list [G1|T] is a conjunction of the proof trees of P1 and T1 as (P1, T1).
Cont...

- Goal: ?- gen([mem(4,[3,4])], Proof).
- Proof = (mem(4, [3,4]) ⇒ (mem(4,[4]) ⇒ true))

?- gen([mem (4,[3,4])], Proof).
   (3)  { Proof = (mem (4,[3,4]) ⇒ P1) }
?- clause(mem (4,[3,4]), Q), gen([Q], P1).
   (2)  { Q = mem (4, [4])}
?- gen([mem(4, [4])], P1).
   (3)  { P1 = (mem (4, [4]) ⇒ P2)
?- clause(mem (4, [4]), Q), gen([Q], P2).
   (m1)  { Q = [true] }
?- gen([true], P2).
   (1)  { P2 = true}
succeeds

Proof = (mem (4,[3,4]) ⇒ (mem (4, [4]) ⇒ true) )
Iteration Looping

- Iterative loops are important facility and can simulate them in Prolog easily.
- The predicate \textit{repeat} (user defined predicate) creates a loop (similar to the loops in imperative programming language)
- Even though it is not a system defined predicate but is quite useful predicate and is defined as follows:
  \begin{verbatim}
  repeat.
  repeat :- repeat.
  \end{verbatim}
Login Module – asking password till it is correct

```
login :- getdata(_, _), write("You are logged on "), nl.  (1)
login :- repeat, write("Sorry, you are not permitted"), nl,  (2)
        write("Try again: "), getdata(_, _),
        write("You are now logged on"), nl.
getdata(N, P) :- write("Enter your login name: "),
                readln(N), nl
                write("Enter your password: "),
                readln(P), nl, user(N, P).  (3)
user(john, john1).
user(mike, mike1).
user(mary, mary1).
user(jack, jack1).
```
Higher Order Predicate

- Define a predicate which checks whether some predicate S succeeds for all possible variables that satisfy some other predicate R.
- In other words, we want to say that S never fails when R has succeeded previously.

\[\text{for\_all}(R, S) :- \neg\text{failure\_exist}(R, S).\]
\[\text{failure\_exist}(R, S) :- \text{call}(R), !, \neg\text{call}(S).\]

- This is basically a universal quantification of S with respect to R i.e., \(\forall(R) S\).
  - Predicate defined above is an example of second order predicate where quantification is over predicate and not on variables.