1 Answers

2 Recursive clause

- A rule defined in terms of itself.
  i.e. t :- t1,t2, ..., tn
- where t and atleast one ti have the same functor and arity

```
ancestor(Ancestor, Person) :-
  parent(Parent, Person),
  ancestor(Ancestor, Parent).
(tail recursion)
ancestor(Ancestor, Person) :-
 parent(Ancestor, Perso).
(base case)
```

Make *head* and *tail* of the following :

```
[X | [Y]]
[[ a,b,c | d],e | [ f,g | X]]
[ a | [ b | [ c | [d]]]]
```

Similarly with lists :

```
list_length([],0).
list_length([H|T], N) :-
list_length(T,N1),
  N is N1+1.
```

This can be explained as :

- Length of an empty list is zero *(fact)*
- Since each list is composed of a head and a tail, the length of a list is the length of the tail plus one element (the head)
  i.e. Length of list(L) = length of (H—T),
  L is divided into a head and tail,
  Length of (H—T) = Length of T + 1

The *recursive use* of the list_length relationship allows the length to be calculated. Each time, we are adding on one element (the head) to our calculation.

3 So, how do Prolog programs work then?

- Given a *program* and a *query*, one wants to know if the program can *satisfy* the query.
- Based on the *resolution principle*, at the heart of which lies the process of *unification*

Unification

Given 2 terms A and B, determine whether they are equal (not necessarily identical, but involving assignment of values to variables).

- if A and B are constant, then they match if and only if they are identical
- If A is a variable and B anything else, theny they match by A becoming *instantiated* to B. Similarly, if A is anything and B is a variable.
- if A and B are compound terms, then they match if they have the same *main* functor (arity must be the same) ;
  - all their corresponding components match in a pair wise manner, by reapplying the algorithm

4 Examples

- Q : test(a, X) = test(Y, b)
  A : X = b, Y = a
  (this is a unifier (substitution of variables)
- Q : test(a, X)
  = test(Y, b)
  A : no
- Q : test(a,X) = test(Y, f(Z))
  A : X = f(Z), Y = a
- Q : X = f(X)
  A : error - term too deep
  a variable being instantiated by a compound term containing its argument

If matching does not succeed, Prolog *back-tracks*, **but if successfully done, it produces the most general instantiation of variables**

Prolog works by defining the problem in a declarative way - this is easier for us, as it more closely resembles our method of thinking, however, Prolog solves a problem using the procedural style

```
nice :-
 warm.
nice :-
 sunny,
 not_windy.

warm :-
 temperature( T),
 T > 20.

not_windy :-
 wind_speed( S),
 S > 5.

sunny.
temperature(15).
wind_speed( 3).
```

To **prove** *that it is nice, it must first be proved that it is sunny and then that it is not windy*
or
To satisfy the goal nice, first satisfy the goal sunny, and then not_windy - Procedural meaning.

- Prolog tries to match - done by *instantiating* a variable
- **Backtracking** ensures that if a solution exists to a goal, it is found, and if more than one solution exists then they are *all* found.
- *Consider the hifi example to see how multiple results were obtained*

1. The **query** (question) is a conjunction of atoms or compound terms - also called the **goal** (if more than one conjunct exists, they are refered to as the **subgoals**)
2. The **subgoals** are **satisfied** (i.e. get a yes or a value solution), *from left to right*
3. To satisfy a subgoal, look amongst the relations declared in the program for one with same functor name and arity, i.e. the head of the clauses are inspected. If no match is found between the required query and the head, the subgoal fails (i.e. answer no)
4. If a relation with the same functor as the subgoal is found, try to **unify** the subgoal with the head of the relation. If the unification fails, reject this relation and try looking for a second one, i.e. the declarations for a same functor are inspected in *top to bottom order*

1. The above four steps are repeated until :
   either no unifiable relation is found, in whcih the subgoal fails *answer no.*
   or
   a unifiable relation is found; the tne values given (instantiated) to the variables in the unifier are transmitted through to the other subgoals (if any) and to the body (if any) of the relation; once this transmission is completed, the body of the relation replaces the subgoal in the query conjunction
2. BEcause of the other subgoals and/or the body being possibly empty, the query conjunction may eventually be empty : success (answer *yes* or values for initial variables in query)

This strategy of left to right, top to bottom is know as *depth first with backtracking*

- Variables are binded (instantiated) to values while the computation is going *forward* and they retain those values (no possibility to reassign values like in C or Pascal)
- When *backtracking* takes place because of a failure, the computation (or part of it) is undone and some of the bindings of variables becomes undone.

5 Tracing a Prolog program

- When we make the query :

  nice.

- *nice* becomes the goal to be satisfied.
- Next, *warm* becomes the goal to be satisfied, as *warm* is the first clause Prolog finds.
- Prolog again searches the program to find a clause whose head matches the desired goal, *warm*
- Two goals are found in the body of the rule to be satisfied : *temperature (T)* and *T ¿ 20.* Prolog attempts to satisfy them left to right. When fact *temperature(15)* is encountered, Prolog tries to satisfy the goal *15 ¿ 20.* This is not possible, so goal fails and Prolog automatically backtracks to the last goal that was matched, i.e. temperature (T), and tries to perform this match in another way. While backtracking (T) gets uninstantiated.
- To satisfy the goal *temperature (T)*, Prolog does not start search from the beginning but continues from the clause where this goal was matched last. It starts with the fact temperature (15), and searches for another clause with head of the same name (temperature). = it fails.

- Prolog again automatically backtracks to the last goal that was successfully matched i.e. to the goal *warm*. Now it tries to find an alternative clause whose head would match this goal. Search continues from the point where the last match occured, and not from the beginning. Since another clause for warm cannot be found, Prolog backtracks to *nice*
- A new clause is now found !!
- Execution is now performed as above, with the goal to be satisfied now being *sunny* and *not_windy*. The goal *sunny* is satisfied by matching it with the fact *sunny*.
- Similarly the goal *not_windy* is matched with the relevant clause
- Now, the two goals to be satisfied are *wind_speed(S)* and S ¡ 5, which are compared with the fact *wind_speed(3)* and the goal 3 ¡ 5, which is the final goal

The execution of a Prolog program is therefore non-deterministic because of multiple execution paths. Of course, if all procedures in a program consisted of only one clause, then only one execution path would exist

- The *closed world* assumption
- Prolog program as a set of axioms (as facts and rules) from which we can prove theorems (get answer to questions) using *logical deductive reasoning*
- We state a theorem and Prolog tries to *prove* it (derive it) from the given axioms (program). Hence, the concept of *Prolog as a theorem prover.*

Useful to draw **search trees** *to understand the operation of Prolog in backtracking* - Graphical representations always prefered.