Advanced Features in Prolog

Lecture Module 10
Objects in Prolog

- Object is a collection of attributes. Treating related information as a single object is similar to records in conventional programming languages.
- For instance, an object for address can be defined as `address(Number, City, Zipcode, Country).
  - Here entire address is treated as a single object and can be used as an argument of a predicate.
  - `address` is called functor and `Number, Street, City, Zipcode` and `Country` are variables called components which hold actual values.
Example

- Represent a fact “a lecture course on ‘ai’, given on Monday from 9 to 11 in the morning by Saroj Kaushik in block 3, room 21” in Prolog.

- Two representations as follows:
  - $\text{course} (\text{ai}, \text{monday}, 9, 11, \text{saroj}, \text{kaushik}, \text{block3}, \text{room21}).$ (1)
  - $\text{course} (\text{ai}, \text{time}(\text{monday},9,11), \text{lecturer}(\text{saroj}, \text{kaushik}), \text{location}(\text{block3, room21})).$ (2)

- In representation (1), there is a relationship between eight items.

- In (2) there is a relationship between four objects – a course name, a time, a lecturer and a location.
The four-argument version of course enables more concise rules to be written by abstracting the details that are relevant to the query.

Let us define few useful predicates using representation (2) for *course facts*.

/* teacher L teaches a course C.*/
teacher_course(L, C)
/* teacher L teaches a course C on day Day.*/
teacher_on_day(L, Day, C)
/* course C of D duration. */
duration(C, D)
These predicates are defined as follows:

- `teacher_course(L, C) :- course(C,_, lecturer(L,_,_),_)`.
- `teacher_on_day(L, Day, C) :- course(C,time(Day, _, _,),L, _,_)`.
- `duration(C, D) :- course(C,time(_,Start,Finish),_,_), D is Finish – Start`.

Note that the details not relevant in particular rule formation are hidden by putting underscore (_, _). This is called Data abstraction.

No definite rules to decide when to use structured data or not.
Cont…

**Query:** Who teaches ai course?
**Goal:** \(?-\) teacher_course(L, ai).

\(?-\) teacher_course(L, ai).
\(?-\) course(ai, _, lecturer(L, _), _).
\{L = saroj\}

\(\bigg\}\)

**Answer:** \(L = saroj\)

**Query:** Which course does saroj teach?
**Goal:** \(?-\) teacher_course(saroj, C).

\(?-\) teacher_course(saroj, C).
\(?-\) course(C, _, lecturer(saroj, _), _).
\{C = ai\}

\(\bigg\}\)

**Answer:** \(C = ai\)
A binary tree is a finite set of elements that is either empty or is partitioned into two disjoint binary sub trees.

Representation of binary tree in Prolog:

\[ \text{b_node(Left_subtree, Root, Right_subtree)} \]

- \text{b_node} is functor having three arguments.
- The first and last arguments are \textit{left} and \textit{right sub trees} of the original binary tree.
- Middle argument is the \textit{root} of a binary tree.

The empty binary tree is represented by an atom called \textbf{void}. 
b_node(L, 20, R)
L = b_node(b_node(nil, 16, nil), 30, b_node(nil, 14, nil));
R = b_node(nil, 47, b_node(nil, 10, nil))
Traversals of Binary Tree

- Binary trees are traversed using recursion, mainly, in preorder, inorder and postorder.
  - **Preorder Traversal**: visit root, traverse left subtree in preorder, right subtree in preorder.
  - **Inorder Traversal**: traverse left subtree in inorder, visit root and right subtree in inorder.
  - **Postorder Traversal**: traverse left subtree in postorder, right subtree in postorder and visit root.
<table>
<thead>
<tr>
<th>Preorder Traversal</th>
<th>Rule No</th>
</tr>
</thead>
<tbody>
<tr>
<td>preorder(B_tree, L) – succeeds by unifying L with the list of elements obtained in preorder traversal of B_tree.</td>
<td>2</td>
</tr>
<tr>
<td>preorder(nil, []). preorder(b_node(L, V, R), X) :- preorder(L, L1), preorder(R, R1), append([V</td>
<td>L1], R1, X).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inorder Traversal</th>
<th>Rule No</th>
</tr>
</thead>
<tbody>
<tr>
<td>inorder(B_tree, L) – succeeds by unifying L with the list of elements obtained in inorder traversal of B_tree.</td>
<td>2</td>
</tr>
<tr>
<td>inorder(nil, []). inorder(b_node(L, V, R) :- inorder(L, L1), inorder(R, R1), append(L1, [V</td>
<td>R1], X).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Postorder Traversal</th>
<th>Rule No</th>
</tr>
</thead>
<tbody>
<tr>
<td>postorder(B_tree, L) – succeeds by unifying L with the list of elements obtained in postorder traversal of B_tree.</td>
<td>2</td>
</tr>
<tr>
<td>postorder(nil, []). postorder(b_node(L, V, R) :- postorder(L, L1), postorder(R, R1), append(R1, [V], Y), append(L1, Y, X).</td>
<td>1</td>
</tr>
</tbody>
</table>
The System predicate "cut"

- Prolog is non deterministic in nature.
- Prolog provides a system defined predicate called cut (!) for affecting the procedural behavior of program and to limit the non determinism by preventing interpreter from finding alternative solutions.
  - There are many applications, where the very first solution is of interest, if it exists.
- Cut prunes the search tree and hence shortens the path traversed by Prolog interpreter.
- It reduces the computation time and also saves storage space.
- Semantically 'cut' always succeeds.
Example

Write a program that lists the users and library facilities open to them according to the following scheme.

facilities

- **basic**
  - Reference Enquiry
    - open to all users

- **additional**
  - Borrowing Inter library loan
    - open to only those users who do not have any book overdue
Prolog Program using Cut

list(U, F) :- user(U), facility(U, F). \hspace{1cm} (1)
facility(U, F) :- overdue(U, B), !, basic(F). \hspace{1cm} (2)
facility(U, F) :- general((F)). \hspace{1cm} (3)
basic(‘reference’).
basic(‘enquiry’).
general(F) :- basic(F). \hspace{1cm} (4)
general(F) :- additional(F). \hspace{1cm} (5)
additional (‘borrowing inter library loan’).
overdue (‘S. K. Das’, logic).
user (‘S. K. Das’).
user (‘Rajan’).
Use of Cut

- In Prolog, the rules are of *if-then* type.
- If we are interested in implementing *if-then-else* type of rule in Prolog, then we can make use of cut to do that.
- Define a predicate named as *if_then_else* as follows:
  
  ```prolog
/* if_then_else(U, Q, R)- succeeds by solving Q if U is true else by solving R. */
if_then_else(U, Q, R) :- U, !, Q.
if_then_else(U, Q, R) :- R.
  ```
- Operationally it means that "prove U and if succeeds, then prove Q else prove R".
- Declaratively, the relation if_then_else is true if U and Q are both true or if U is not true and R is true.
Types of Cut

- There are two types of cuts viz., green cut and red cut.

- **Green cut**: Does not affect the solution but affects the efficiency of the Prolog program.
  - Removal of such cut does not change the meaning of the program.

- **Red cut**: The cut whose removal from the program changes the meaning of the program.
Example

- Write Prolog program for merging two ordered lists using green cut.
- If we remove cuts from the rules, then the solutions are not affected and program does not change.

/* merge(X, Y, Z) - Z is obtained by merging ordered lists X and Y. */

merge( [X|X1], [Y|Y1], [X|Z] ) :- X < Y, !, merge(X1, [Y|Y1], Z).
merge( [X|X1], [Y|Y1], [X, Y|Z] ) :- X = Y, !, merge(X1, Y1, Z).
merge( [X|X1], [Y|Y1], [Y|Z] ) :- X > Y, merge([X|X1], Y1, Z).
merge(X, [ ], X).
merge([ ], Y, Y).
Example – Red cut (maximum of two numbers)

<table>
<thead>
<tr>
<th>Version I – Red cut</th>
<th>Version 2 – Green cut</th>
</tr>
</thead>
<tbody>
<tr>
<td>% max(X, Y, Z) – Z is unified with maximum of X and Y.</td>
<td>max(X, Y, Z):- X ≥ Y, !, Z = X.</td>
</tr>
<tr>
<td>max(X, Y, Z):- X ≥ Y, !, Z = X.</td>
<td>max(X, Y, Y) :- X &lt; Y.</td>
</tr>
<tr>
<td>• Cut’s removal will affect the solution.</td>
<td>• Cut’s removal will not affect the solution.</td>
</tr>
</tbody>
</table>

Verify results using both versions with and without !
?- max(5, 4, 4).
Fail Predicate

- **Fail** predicate is used for forced backtracking by failing a rule.
- All the sub goals defined after fail will never be executed.
- Hence predicate fail should always be used as the last sub goal in a rule.
- It is to be noted that rule containing fail predicate will not produce any solution.
Example

- Consider the following example to illustrate the use of fail predicate.
  
  listing(Name, Address) :- emp(Name, Address).
  emp(ram, cse).
  emp(rita, maths).
  emp(gita, civil).

  Goal: ?- listing(Name, Address).

  All possible solutions obtained on executing above goal by normal backtracking of Prolog are:
  
  Name = ram , Address = cse;
  Name = rita , Address = maths;
  Name = gita , Address = civil;
Example – Cont…

listing :- write('Name'), write(' Address'), nl,
        emp(Name, Address), write (Name),
        write(' '), write(Address), nl, fail.

emp(ram, cse).
emp(rita, maths).
emp(gita, civil).

Goal:  ?- listing

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ram</td>
<td>cse</td>
</tr>
<tr>
<td>rita</td>
<td>maths</td>
</tr>
<tr>
<td>gita</td>
<td>civil</td>
</tr>
</tbody>
</table>
Cut and Fail Combination

- Cut and fail combination is also useful for expressing negative facts.
- For example, "john does not like snakes" could be expressed by the following rule and fact.
  
  ```prolog
  like(john, snake) :- !, fail.
  like(john, X).
  ```
- This coding of rules state that "John likes everything except snake" which implies that "john does not like snakes".

  Goal:  ?- like(john, snake). Answer: no
  Goal:  ?- like(john, dog). Answer: yes
Negation as Failure

- Horn clauses are incomplete version of FOPL because of the limitation of one positive literal in a clause.
  - Here we describe an extension to the LP computation model that allows a limited use of negative information in the program.
- The Prolog rule $P : - Q_1, Q_2, ..., Q_n$ expresses only if condition for $P$ and it says nothing about other conditions under which $P$ can be true.
- Negation as failure introduces a closed world in the limited sense. Every thing not stated is taken to be false.
A goal \textit{not}(G) is said to be a \textit{logical consequence} of a program P if G is not a logical consequence of P.

Therefore, if goal G can not be shown to be true, then infer the truth of \textit{not}(G).

A goal \textit{not}(G) is implied by a program P by the \textit{negation as failure} rule.

Example

\textbf{Goals:}

\begin{verbatim}
?- not(2 < 4).
?- 2 < 4.
Succeeds
Not (2<4) -- Fails
\end{verbatim}

Answer: No
The cut-fail combination can be used to implement a version of negation as failure.

It is difficult to implement negation both efficiently and correctly.

\[
\text{not(G)} \quad :- \quad G, !, \text{fail.}
\]
\[
\text{not(G).}
\]

The cut ensures that if G succeeds, the second clause will not be attempted and if G fails, then cut is not activated and so second clause is tried and it succeeds.
Example - Cont…

- It is a good programming style to replace \textit{cut} by the use of \textit{not} if possible because the programs containing cuts are generally harder to understand.

- The rule using \textit{not} predicate is more readable and gives clear semantic interpretation of the rule but at times could be computationally expensive.

- Predicate \textit{if\_then\_else} can be implemented using \textit{not} predicate instead of cut.
  \[
  \text{if\_then\_else}(P, Q, R) \quad :- \quad P, Q.
  \]
  \[
  \text{if\_then\_else}(P, Q, R) \quad :- \quad \text{not}(P), R.
  \]

- Here the goal \textit{P} have to be computed again while trying second rule of if\_then\_else.
Logical Limitations of Prolog

- Prolog does not allow disjunction ('or') of facts or conclusion such as
  "If car does not start and the light does not come on, then either battery is down or problem with ignition or some electric fault"
- Such rules can not be expressed straight away in Prolog.
- Prolog does not allow negative facts or conclusions e.g., not(a) :- b ; not(c) etc are not valid in Prolog.
- Prolog does not allow facts, rules having existential quantifications.