Programming Techniques in Prolog

Programming Techniques in Prolog

- Recursion is one of the most important execution control techniques in Prolog.
- Iteration is another important programming technique.
  - Prolog does not support it.
  - Iteration can be achieved by making recursive call to be the last sub goal in the rule and by using the concept of accumulators which are special types of arguments used for holding intermediate results.
- The issues of termination, rule order, goal order and redundancy are also important.

Recursion in Prolog

- Let us consider a classical example of computing factorial using recursion.
- Recurrence relation or mathematical definition for computing factorial of positive integer is:

$$\text{FACTORIAL}(n) = \begin{cases} 1, & \text{if } n = 0 \\ n * \text{FACTORIAL}(n-1), & \text{otherwise} \end{cases}$$

- In Prolog, program for computing factorial of a positive number is written as follows:

Prolog Program for Factorial

```
/*    factorial(N, R) – succeeds if R is unified with
                          the factorial of  N.     */
      factorial(0, 1).                                          (1)
      factorial(N, R) :- N1 is N – 1, factorial(N1, R1),
                             R is  N * R1.                      (2)
```

- Rule (1) states that factorial of 0 is 1 which is a terminating clause.
- Rule (2) is a recursive clause which states that factorial of N is calculated by multiplying N with factorial of (N-1).

Goal: ?- factorial(3, R).

Search Tree

$$\begin{tikzcd}[row sep=large]
\text{?- factorial(3, R).} \arrow[d, "(2)"', "\{N = 3\}"] \\
\text{?- N1 is 2, factorial(N1, R1), R is N * R1} \arrow[d, "(2)"', "\{N1 = 2\}"] \\
\text{?-N2 is 1, factorial(N2, R2), R1 is N1 * R2, R is N * R1.} \arrow[d, "(2)"', "\{N2 = 1\}"] \\
\text{?-N3 is 0, factorial(N3, R3), R2 is N2 * R3 , ….} \arrow[d, "(1)"', "\{N3 = 0, R3 = 1\}"] \\
\text{?- R2 is 1 * R3 , R1 is 2 * R2, R is 3 * R1.} \arrow[d, "\text{Succeeds } \{R2 = 1, R1 = 2, \mathbf{R = 6}\}"] \\
\boxed{}
\end{tikzcd}$$

List Manipulation in Prolog

- List is a common and very useful recursive data structure in non numeric programming.
- In Prolog, a list is represented by elements separated by comma and enclosed in square brackets, e.g., A = [2, 4, 6, 8], B = [[a, b], [c], [p, q, t]] are examples of list.
- Empty list with no elements is represented by [ ] and is useful for terminating recursive programs for list.

Membership Program

Solution: An element is a member of a list if it matches with the head or if it is contained in the tail of the list.

- The fact that it matches with the head of a list is written as: mem_list(X, [X|_]).
- The rule that if an element does not match with the head, then it may be in the tail is written as: mem_list(X, [_ | Y]) :- mem_list(X, Y).

- Underscore is used for non care entry. The complete program in Prolog is given below:

Contd..

/* mem_list(X, L) – succeeds if an element X is a member of list L. */

```
mem_list(X, [X | Y ]).                          (1)
mem_list(X, [ Y | T]) :- mem_list(X, T).        (2)
```

Goal: ?- mem_list(d, [a, b, c, d, e, f]).

Proof tree:

$$\begin{tikzcd}
\text{?- mem\_list(d, [a, b, c, d, e, f]).} \arrow[d, "(2)"', "\{X = d,\ T = [b, c, d, e, f]\}"] \\
\text{?- mem\_list(d, [b, c, d, e, f]).} \arrow[d, "(2)"', "\{X = d,\ T1 = [c, d, e, f]\}"] \\
\text{?- mem\_list(d, [c, d, e, f]).} \arrow[d, "(2)"', "\{X = d,\ T2 = [d, e, f]\}"] \\
\text{?- mem\_list(d, [d, e, f]).} \arrow[d, "(1)"'] \\
\Box \ \text{succeeds}
\end{tikzcd}$$

Ans: Yes

Non ground Query

Goal: ?- mem_list(X, [a, b, c]).

Search tree:

$$\begin{tikzcd}
 & \text{?- mem\_list(X, [a, b, c]).} \arrow[dl, "(1)\ \{\mathbf{X = a}\}"'] \arrow[dr, "(2)\ \{T = [b, c]\}"] & & & \\
\text{succeeds} & & \text{?- mem\_list(X, [b, c]).} \arrow[dl, "(1)\ \{\mathbf{X = b}\}"'] \arrow[dr, "(2)\ \{T = [c]\}"] & & \\
 & \text{succeeds} & & \text{?- mem\_list(X, [c]).} \arrow[dl, "(1)\ \{\mathbf{X = c}\}"'] \arrow[dr, "(2)\ \{T = []\}"] & \\
 & & \text{succeeds} & & \text{?- mem\_list(X, []).} \arrow[d] \\
 & & & & \text{fails}
\end{tikzcd}$$

Answer:X = a; X = b; X = c

Append Prolog Program

```
% append(L, M, N)   –   succeeds if N is concatenation  of
   L and M
        append([], L, L).                                  (1)
        append([X | L], M, [X | N])   :- append(L, M, N).  (2)
```

Goals:

- append([2,3], [4,5], [2,3,4,5]).

  Generate Proof Tree

- append([2,3], [4,5], X).

  Generate Search Tree

Termination

- Depth first traversal of Prolog has a serious problem. If the search tree of a goal with respect to a program contains an infinite branch, the computation will not terminate.
- Prolog may fail to find a solution of a goal, even if the goal has a finite computation.
- *Non termination* arises because of the following reasons:
  - *Left recursive rules*: The rule having a recursive goal as the first sub goal in the body.

Example

- Consider acyclic directed graph {edge(p, q), edge(q, r), edge(q, s), edge(s, t)}
  - where edge(A, B) is a predicate indicating directed edge in a graph from a node A to a node B.
- Write Prolog program to check whether there is a route from one node to another node.

Pictorial representation of graph

$$\begin{tikzcd}
& p \arrow[d] & \\
& q \arrow[dl] \arrow[dr] & \\
r & & s \arrow[dl] \\
& t &
\end{tikzcd}$$

- Directed Graph= {edge(p, q),edge(q, r), edge(q, s), edge(s, t)},
- There is a route from node say, A to B if there is a route from A to some node C and an edge from C to B or direct edge from A to B.

Prolog Program for finding Route

The prolog rules for finding route are written as:

```
route(A, B) :-     route(A, C), edge(C, B).          (1)
route(A, B)  :-    edge(A, B).                       (2)
edge(p, q).
edge(q, r).
edge(q, s).
edge(s, t).
```

Query: Check whether there exist a route between nodes p and t .

Proof Tree

Goal: ?- route(p, t).

Proof tree:

$$\begin{tikzcd} \text{?- route(p, t).} \arrow[d, "(1)"'] \\ \text{?- route(p, C), edge(C, t)} \arrow[d, "(1)"'] \\ \text{?- route(p, C1), edge(C1, C), edge(C, t).} \arrow[d, "\textit{Non terminating}"] \\ {} \end{tikzcd}$$

- The above program is non terminating because of left recursion. Better way of writing the above program is to write recursive call as the last call.

Tail recursive Rule

- This type of rule is called tail recursive rule. The rule (1) is modified and is rewritten as

  route(A, B) :- edge(A, C), route(C, B). (1')

Proof tree of the goal

$$\begin{tikzcd}
 & \text{?- route(p, t)} \arrow[d, "(1')"'] & \\
 & \text{?- edge(p, C), route(C, t).} \arrow[d, "\{ C=q\}"] & \\
 & \text{?- route(q, t).} \arrow[d, "(1')"'] & \\
 & \text{?- edge(q, X), route(X, t).} \arrow[dl, "\{X = r\}"'] \arrow[dr, "\{X = s\}"] & \\
\text{?- route(r, t).} \arrow[d, "\text{Fails}"] & & \text{?- route(s, t).} \arrow[d, "\text{Succeeds}"] \\
 {} & & {}
\end{tikzcd}$$

Non Termination

- *Left Recursive* rules are generally non terminating.
- *Circular definition* is another means of generating *non terminating* computation.
- For example,
- Goal *?- teacher(mike, robert)* does not terminate with respect to the following program.

```
teacher(X, Y) :-     student(Y, X).      (1)
student(X, Y) :-     teacher(Y, X).      (2)
student(robert, mike).
student(john, robert).
```

Cintd..

Proof tree:

$$\begin{tikzcd}
\text{?- teacher(mike, robert)} \arrow[d, "(1)"'] \\
\text{?- student(robert, mike)} \arrow[d, "(2)"'] \\
\text{?- teacher(mike, robert)} \arrow[d] \\
\textbf{Non termination}
\end{tikzcd}$$

?? Solution

Rule Order

- A rule order is very important issue in Prolog programs and is irrelevant for logic programs.
- Changing the order of rules in a program, permutes the branches in any search tree for a goal to be solved with that program.
- Since the search tree is traversed using depth first approach, the *order of solutions* will be different.
- It may affect the efficiency of executing goal.
- Let us consider again Prolog program for finding route between any two nodes of acyclic directed graph mentioned earlier.

Contd…

- If we solve a query *?- route(p, X)*, then the answers {X = t; X = s; X = r; X = q} are produced on backtracking (verify yourself).
- Here we travel to the end of search tree and the solutions are obtained on backtracking.
- This program is inefficient with respect to time.
- The order of facts is important. The rules (1) and (2) are interchanged and written as:

```
route(A, B)  :- edge(A, B).                    (1)
route(A, B)  :- edge(A, C), route(C, B).    (2)
```

Contd…

Goal: ?- route(p, X).

Search tree: ?- route(p, X).

$$\begin{tikzcd}
 & \text{?- route(p, X).} \arrow[dl, "(1)"'] \arrow[dr, "(2)"] & & \\
\text{?- edge(p, X).} \arrow[d, "\{\ \mathbf{X=q}\}"'] & & \text{?- edge(p, Y), route(Y, X).} \arrow[d, "\{Y = q\}"] & \\
\text{succeeds} & & \text{?- route(q, X).} \arrow[dl, "(1)"'] \arrow[dr, "(2)"] & \\
 & \text{?- edge (q, X).} \arrow[d] & & \text{?- edge(q, Y), route(Y, X).} \arrow[d] \\
 & \text{continue} & & \text{continue}
\end{tikzcd}$$

Answer:{X = q; X = r; X = s; X = t}

Goal Order

- In Prolog, goal order is more significant than rule order. This also is irrelevant for logic programs.
- It may affect the termination because subsequent sub goals have different bindings and hence different search trees. Consider the following recursive rules.

```
ancester(X, Y)   :-  parent(X, Z), ancester(Z, Y).
route(X, Y)      :-  edge(X, Z), route(Z, Y).
```

- If the sub goals in the body of above rules are swapped, then the rules becomes left recursive and all the computations will become non terminating.
- It also affects the amount of search the program does in solving a query .

Redundancy

- Redundancy of the solution to the queries is another important issue in Prolog programs.
- In Prolog, each possible deduction means an extra branch in the search tree.
- The bigger the search tree, higher the computation time.
- It is desirable in general to keep the size of the search tree as small as possible.

Contd…

- Redundant program at times may cause exponential increase in runtime in the event of backtracking.
- There are various reasons by which redundancy gets introduced.
- The same case is covered by several rules e.g.,

```
        maximum(X, Y, X) :- X  ≥ Y.                (1)
        maximum(X, Y, Y) :- Y  ≥  X.               (2)
```

Goal: ?- maximum(2, 2, X).

Search tree:

$$\begin{tikzcd}
 & \text{?- maximum(2, 2, X).} \arrow[dl, "(1)"', "\{X = 2\}"] \arrow[dr, "(2)"', "\{X = 2\}"] & \\
\text{succeeds} & & \text{succeeds}
\end{tikzcd}$$

Contd…

- Here we get the same solution twice. We can avoid such redundancy by putting unique conditions.
- The rule (2) can be rewritten as

```
maximum(X, Y, Y) :- Y > X .
```

- *Redundancy* in computation appears in the program by not handling special cases separately.
- Consider a program *append* for concatenation of two lists.
- The computation time for the following *goal* is proportional to the number of elements in the first list even though the second list is empty.

```
?- append([2,4,6,7,8], [], N)
```

- If we add another rule in the program for this special case i.e., append(N, [], N) , then we get answer immediately and hence the redundancy in computation is reduced.

Iterative Programming in Prolog

- Prolog does not have storage variables that can hold intermediate results of the computation and can get modified as the computation progress.
- To implement iterative algorithm one requires the storage of intermediate results.
  - Prolog predicates are augmented with additional arguments called *accumulators*.
  - These are similar to local variables in procedural languages.
  - The programs written using accumulators are *iterative* in nature.
- A Prolog program is iterative if it contains only unit clauses and iterative clauses.

Contd…

- In iterative form of a program, make the recursive call as the last call if possible.
- If the recursive call is the last sub goal in a rule, then the same space is used for recursive call.
- Let us write Prolog program for factorial using accumulators.

/* Here I and T are accumulators used for storing the counter value and intermediate computations */

```
fact (N, R)        :-     fact(N, R, 0, 1).              (1)
fact(N, R, I, T)   :-     I < N, J is I + 1, T1 is J * T,
                          fact(N, R, J, T1).             (2)
fact(N, R, N, R).                                        (3)
```

Contd…

- In rule (1), the goal *fact* calls *fact* with four arguments that corresponds to initialization stage.
- Rule (2) is the basic iterative loop, performed by fact.
- The computation terminates when the accumulator I becomes equal to N.
- Rule (3) is a base rule used to terminate the computation.
- The value of the factorial is returned as a result of the unification of second argument with the fourth argument.
- In Prolog, predicates can be overloaded (i.e., same name can be used with different number of arguments).

Search Tree

Goal: ?- fact (3, R).

Search tree: ?- fact(3, R).

(1) ↓

?- fact(3, R, 0, 1).

(2) ↓

?- 0 < 3, J is 1, T1 is 1 * 1, **fact(3, R, 1, 1).**

(2) ↙

?- 1 < 3, J is 2, T1 is 2 * 1, **fact(3, R, 2, 2).**

(2) ↙

?- 2 < 3, J is 3, T1 is 3 * 2, **fact(3, R, 3, 6).**

(2) ↙ (3) ↘ {R = 6}

fails succeeds

Answer: R = 6