CS323: AI (Hands on with Prolog) **Spring 2016**

Lecture 11: Feb. 10, 2016

Lecturer: K.R. Chowdhary *: Professor of CS (VF)*

11.1 Some Built-in Predicates

The built-ins can be used in a similar way as user-defined predicates. The important difference between the two is that a built-in predicate is not allowed to appear as the principal function in a *fact* or in the *head of a rule*. This must be so, because using them in such a position would effectively mean changing their definition.

Equality. We write $X = Y$. Such a goal succeeds, if the terms X and Y can be matched.

Output. Besides Prolog's replies to queries, if you wish your program to have further output, you can use the *write* predicate. The argument can be any valid Prolog term. In the case of a variable argument, its value will be printed. Execution of the predicate causes the system to skip a line, as in the following cases.

```
?- write(Hello World!), nl.
Hello World!
Yes

?- X = elephant, write(X), nl.
elephant
X = elephant
Yes

read(N).
write('the number is'), write(N), nl.
the number is 5
N = 5
Yes
```

Matchings. Following are the examples for matchings. If two expressions matches, the output is 'Yes' otherwise it is 'No'. The query, to prolog shows that the two expressions cannot be matched.

```
?- p(X, 2, 2) = p(1, Y, X).
No
```

Sometimes there is more than one way of satisfying the current goal. Prolog chooses the first possibility (as determined by the order of clauses in a program), but the fact that there exists alternatives, is recorded. If at some point, Prolog fails to prove a certain subgoal, the system can go back and try an alternative left behind in the of executing of the goal. This process is known as *backtracking*. The following example demonstrates backtracking.

11.2 Recursive Programming

Using recursive programs, we can provide recursive definition of functions. We know that the factorial $n!$ of a natural number n is defined as the product of all natural numbers from 1 to n. Here's a more formal, recursive definition (also known as an inductive definition), and the code in prolog.

Example 11.1 *Factorial Program.*

Recall the definition of factorial in equations (**??**) and (**??**) in section (**??**).

$0! = 1$, (base case)
$n! = (n1)! * n$, for $n \geq 1$ (Recursion rule)

```
%finding factorial.
fact(0, 1).              % base case

fact(N, R) :-  N >= 1,      % recursion step
               N1 is N - 1,
               fact(N1, R1),
               R is R1 * N.
```

□

For a recursive program to test the membership of an element in a set, if the element is not as *head* of the list, then it is in the *tail*. The process is recursively called. The membership algorithm is built-in feature of prolog, as well as it can be user-defined.

A recursive algorithm for GCD (greatest common divisor) based on Euclid's Algorithm can be constructed as follows.

Example 11.2 *Program for Greatest Common Divisor (GCD).*

```
%gcd
gcd(X, X, X).
gcd(X, Y, Z) :- X > Y, D is X - Y, gcd(D, Y, Z).
gcd(X, Y, Z) :- X < Y, D is Y - X, gcd(X, D, Z).
```

□

Example 11.3 *Towers of Hanoi Problem.*

Given three stacks A (source), B (destination), and I (intermediate), the towers of Hanoi problem is to move N number of disks from stack A to B using I as temporary stack. The disks are originally on stack A such

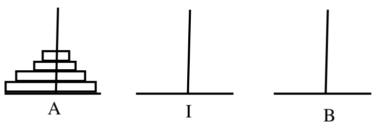

Figure 11.1: Towers of Hanoi Problem.

that larger diameter disks are below the smaller diameter disks, and no two disks have equal diameters. The movement is to be done following the rules of this game, which states that only one disk is to be moved at a time, and at no time the bigger diameter disk shall come over a smaller diameter disk (figure 11.1).

```
move(A,B):- nl,
            write('move top from '),
            write(A),
            write(' to '),
            write(B).
transfer(1,A,B,I) :- move(A,B).
transfer(N, A, B, I):- N > 1,
                       M is N -1,
                       transfer(M, A, I, B),
                       move(A, B),
                       transfer(M, I, B, A).
```

The algorithm uses the strategy: move $n-1$ disks from A to I, then move a single disk from A to B, finally move $n-1$ disks from I to B. For $n-1$, it recursively calls the algorithm. The predicate *nl* stands for new-line.

□

11.3 List Manipulation

Prolog represents the lists contained in square brackets with the elements being separated by commas. Here is an example:

```
[elephant, horse, donkey, dog]
```

Elements of lists could be any valid Prolog terms, i.e. atoms, numbers, variables, or compound terms. A term may also be other list. The empty list is denoted by '[]'. The following is another example for a (slightly more complex) list:

```
[elephant, [], X, parent(X, tom), [a, b, c], f(22)]
```

Internally, lists are represented as compound terms using the function . (dot). The empty list '[]'is an atom and elements are added one by one. The list $[a, b, c]$, for example, corresponds to the following term:

```
.(a, .(b, .(c, [])))
```

We discussed in section (**??**) about lists. A list is a recursive definition, consisting of a head and a tail. The tail also comprises of head and rest of the elements as tail, and so on, until the tail is empty list.

Example 11.4 *Membership Program.*

```
% membership built-in
?-member(2, [a, b, c, 2, 4, 900]).
Yes.

% membership program
ismember(X, [X|R]).       % matches with 1st element
ismember(X, [Y|R]) :- ismember(X, R). % try for next
                                       % element
```

□

The built-in program *append*, appends two lists.

```
?append([1, 2, 3], [a, b, c], X). % This is buit-in
X=[1, 2, 3, a, b, c]
```

Example 11.5 *Appending of lists.*

$append([], L, L).$
$append([X|L1], L2, [X|L3]) : -append(L1, L2, L3).$

For a query, we write,

```
? append([1, 2], [3, 4], Z)
Z = [1, 2, 3, 4]
Yes
?-
```

11.4 Arithmetic Expressions

Prolog is not designed to handle arithmetics efficiently. Hence, it handles expressions and assignment operations in some different way.

```
?3 + 5 = 8.
No

?X is 3 + 5.
X = 8
Yes
```

The terms $3 + 5$ and 8 do not match as the former is a compound term, whereas the latter is a number.

The following are arithmetical relational predicates:

```
X > Y
X < Y
X >= Y
X =< Y
X \= Y
X = Y
```

The last two predicates express inequality and equality, respectively.

11.5 Backtracking, Cuts and Negation

PROLOG offers the programmer a number of predicates for explicitly controlling the back-tracking behaviour of the interpreter. Note that here PROLOG deviates from the logic programming idea. The predicate *True* takes no arguments, and it always succeeds.

11.5.1 Fail

The predicate *Fail* also has no arguments, the condition *fail* never succeeds. The general application of the predicate fail is to enforce *backtracking*, as shown in the following clause:

$$a(X)\ \text{:-}\ b(X), fail.$$

When the query $a(X)$ is entered, the PROLOG interpreter first tries to find a match for $b(X)$. Let us suppose that such a match is found, and that the variable X is *instantiated* to some term. Then, in the next step $fail$, as a consequence of its failure, enforces the interpreter to look for an alternative instantiation to X. If it succeeds in finding another instantiation for X, then again $fail$ will be executed. This entire process is repeated until no further instantiations can be found. This way all possible instantiations for X will be found. Note that if no side-effects are employed to record the instantiations of X in some way, the successive instantiations leave no trace. It will be evident that in the end the query $a(X)$ will be answered by *no*. But, we have been successful in backtracking, i.e., going back and trying all possible instantiations for X, which helps in searching all the values.

The negation in prolog is taken as failure as shown in the following program.

Example 11.6 *Negation as failure.*

```
bachelor(P) :- male(P), not(married(P)).
male(rajan).
male(rajam).
married(dicken).
```

When run, the queries responded as obvious. In the third case, married(Who) succeeds, so the negation of goal fails.

```
?bachelor(rajan).
 yes
```

```
?bachelor(dicken).
no
bachelor(Who).
Who = dicken
no
```

11.5.2 Cut

Some times it is desirable to selectively turn off backtracking. This is done by *cut* (!). The cut, denoted by !, is a predicate without any arguments. It is used as a condition which can be confirmed only once by the PROLOG interpreter: on backtracking it is not possible to confirm a cut for the second time. Moreover, the cut has a significant side effect on the remainder of the backtracking process: it enforces the interpreter to reject the clause containing the cut, and also to ignore all other alternatives for the procedure call which led to the execution of the particular clause.

Example 11.7 *backtracking.*

```
a :- b,c,d.
c :- p,q,!,r,s.
c.
```

Suppose that upon executing the call a, the successive procedure calls b, p, q, the *cut* and r have succeeded (the *cut* by definition always succeeds on first encounter). Furthermore, assume that no match can be found for the procedure call s. Then as usual, the interpreter tries to find an alternative match for the procedure call r. For each alternative match for r, it again tries to find a match for condition s. If no alternatives for r can be found, or similarly if all alternative matches have been tried, the interpreter normally would try to find an alternative match for q. However, since we have specified a cut between the procedure calls q and r, the interpreter will not look for alternative matches for the procedure calls preceding r in the specific clause. In addition, the interpreter will not try any alternatives for the procedure call c; so, clause 3 is ignored. Its first action after encountering the cut during backtracking is to look for alternative matches for the condition preceding the call c, that is, for b. □