32002: AI (Prolog and Rule Chaining) **Spring 2014**

Lecture 15-17: February 11-13, 2014

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

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

15.1 Logic Programming

The PROLOG (logic programming) is useful in problem areas, such as artificial intelligence, natural language processing, databases, . . . , but pretty useless in others, such as graphics or numerical algorithms.

A prolog program is implemented in two parts:

1. *logic*, which describes the problem, and
2. *control*, provides the solution method.

This is in contrast to other programming languages, where description and solution go together. This separation has been expressed by R.A. Kowalski in the following equation:

$$\boxed{\textit{algorithm} = \textit{logic} + \textit{control}} \tag{15.1}$$

Specifying a logic program amounts to:

- Specifying the *facts* concerning the objects and relations between objects relevant to the problem at hand;
- Specifying the *rules* concerning the objects and their interrelationships;
- Posing *queries* concerning the objects and relations.

Prolog is a declarative (or descriptive) language. Programming in Prolog means describing the *world*. Using such programs means asking Prolog questions about the previously described world. The simplest way of describing the world is by stating facts, like this one:

```
bigger(elephant, horse).
```

This states, quite intuitively, the fact that an elephant is bigger than a horse. Let's add a few more facts to our little program:

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
```

This is a syntactically correct program, and after having compiled it we can ask the Prolog system questions (or *queries* in proper Prolog-jargon) about it.

```
?- bigger(donkey, dog). <enter>
Yes
```

The query 'bigger(donkey, dog)'(i.e. the question "Is a donkey bigger than a dog?") succeeds, because the fact 'bigger(donkey, dog)'was previously communicated to the Prolog system. Our next query is, "is a monkey bigger than an elephant?"

```
?- bigger(monkey, elephant).
No
```

The reason of 'No'is that, the program says nothing about the relationship between elephants and monkeys.

Solution would be to define a new relation, which we will call *isbigger*, as the *transitive closure*. Animal X is bigger than animal Y either if this has been stated as a *fact* or if there is an animal Z, for which it has been stated as a fact that animal X is bigger than animal Z and it can be shown that animal Z is bigger than animal Y. In Prolog such statements are called *rules* and are implemented as follows:

```
isbigger(X, Y) :- bigger(X, Y).                  %rule1
isbigger(X, Y) :- bigger(X, Z), isbigger(Z, Y). %rule2
```

As query is can be given now:

```
?- isbigger(elephant, monkey).
Yes
```

In the rule1 above, the predicate 'isbigger(X, Y)' is called *goal*, and 'bigger(X, Y)' is called *sub-goal.* In the rule2 'isbigger(X,Y)' is goal and the expressions after the sign ':-' are called sub-goals. The goal is also called *head* of the predicate, and the set expressions after sign ':-' is called *body* of the prolog statement. In fact, the rule1 above has corresponding predicate

$$if\ bigger(X, Y)\ then\ isbigger(X, Y)$$

or

$$bigger(X, Y) \rightarrow isbigger(X, Y),$$

Similarly, predicate expression for rule2 is

$$bigger(X, Z) \wedge isbigger(Z, Y) \rightarrow isbigger(X, Y).$$

The expressions which are not conditionals, i.e., like,

```
bigger(elephant, horse).
```

are called *facts*(or assertion). The facts and rules, together, make the knowledge-base in a program.

For the query 'isbigger(elephant, monkey)' the Prolog still cannot find the fact 'bigger(elephant, monkey)' in its database, so it tries to use the second rule instead. This is done by matching the query with the head of the rule, which is 'isbigger(X, Y)'. When doing so the two variables get bound: X = elephant, and Y = monkey. The rule says that in order to prove the goal 'isbigger(X,Y)' (with the variable bindings that's equivalent to isbigger(elephant, monkey)), Prolog needs to prove the two subgoals 'bigger(X, Z)' and 'isbigger(Z, Y)', with the same variable bindings. Hence, the rule2 gets transformed to:

$$isbigger(elephant, monkey) : -bigger(elephant, Z),$$
$$isbigger(Z, monkey).$$

By repeating the process *recursively*, the facts that make up the chain between *elephant* and *monkey* are found and the query ultimately succeeds.

Of course, we can do slightly more exciting stuff than just asking yes/no-questions. Suppose we want to know, what animals are bigger than a donkey? The corresponding query would be:

```
?- isbigger(X, donkey).
```

We could also have chosen any other name in place of X for it as long as it starts with an uppercase letter. The Prolog interpreter replies as follows:

```
X = horse;    % press here ';' to get another match
X = elephant ;                              if exists.
No
```

There are many more ways of querying the Prolog system about the contents of its database.

Try to find out the answer for:

```
?- bigger(Who, Whom).
```

You will get many answers ! The prolog treats an argument as variable when started with uppercase letter, which both the *Who* and *Whom* are.

As a final example we ask whether there is an animal X that is both smaller than a donkey and bigger than a monkey:

```
?- isbigger(donkey, X), isbigger(X, monkey).
No
```

Consider writing the two rules rule1, rule2 above in reverse order.

```
isbigger(X, Y) :- bigger(X, Z), isbigger(Z, Y). %rule2
isbigger(X, Y) :- bigger(X, Y).                 %rule1
```

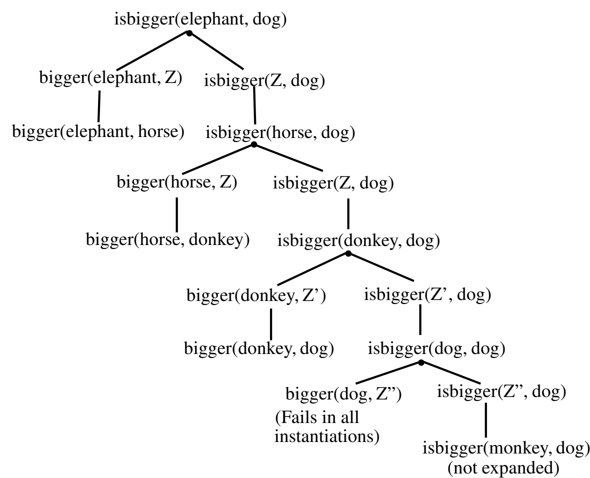

Figure 15.1: Inference Tree.

Let us try to put the query $isbigger(elephant, dog)$. This results to loop as shown in the figure 15.1. On failure at the recursive rule2, it tries at rule1.

Many a times, when started from goal, it may not be possible to reach to facts available. This shows that prolog is *incomplete* in theorem proving even for definite clauses, as it fails to prove facts that can be concluded from knowledge-base.

15.2 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 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 its value will be printed out. 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.
```

Consulting program files. Program files can be compiled using the predicate *consult.* The argument has to be a Prolog atom denoting the particular program file. For example, to compile the file 'big-animals.pl', submit the following query to swi-Prolog:

```
?- consult(big-animals.pl).
```

If the compilation is successful, Prolog will reply with Yes. Otherwise a list of errors is displayed. The 'swi-prolog' can also run in GUI environment in Windows.

The gnu-prolog running on linux can run a prolog program 'big-animals' as follows:

```
$ gprolog <enter>
GNU Prolog 1.3.0
Copyright ....
| ?-[big-animals]. % without extensions.
```

Matchings. Following examples for matchings. If two expressions matches, the output is 'Yes' otherwise it is 'No'.

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

Which shows that they cannot be matched.

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 explains this.

Example 15.1 *demonstrating of Backtracking.*

All men are mortal.
Socrates is a man.
―――――――――――――
Hence, Socrates is mortal.

In Prolog terms, the first statement represents a rule: X is mortal, if X is a man (for all X). The second one constitutes a fact: 'Socrates is a man'. This can be implemented in Prolog as follows:

```
mortal(X) :- man(X).
man(socrates).
```

Note that X is a variable, whereas 'socrates' is constant. The conclusion of the argument, "Socrates is mortal", can be expressed through the predicate 'mortal(socrates)'. After having run the above program we can submit this predicate to Prolog as a query, resulting to following results:

```
?- mortal(socrates).
Yes
```

Prolog agrees with our own logical reasoning. But how did it come to its conclusion? Let's follow the goal execution step by step.

1. The query mortal(socrates) is designated the initial goal.
2. Scanning through the clauses of this program, Prolog tries to match $mortal(socrates)$ with the first possible fact or head of rule. It finds $mortal(X)$ - head of the first (and only) rule. When matching the two socrates is bound to X, with unifier $\{socrates/X\}$.
3. The variable binding is extended to the body of the rule, i.e. $man(X)$ becomes $man(socrates)$.
4. The newly instantiated body becomes our new goal: $man(socrates)$.
5. Prolog executes the new goal by again trying to match it with a rule-head or a fact.
6. Obviously, the goal man(socrates) matches the fact $man(socrates)$, because they are identical. This means the current goal succeeds.
7. This, again, means that also the initial goal succeeds, and prolog responds with 'YES'.

15.3 Recursive Programming

Recursive definition of functions. 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):

Example 15.2 *Factorial Program.*

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

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

Example 15.3 *Membership Program.*

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

```
% membership program
ismember(X, [X|R]).
ismember(X, [Y|R]) :- ismember(X, R).
```

On the similar lines, a recursive algorithm for GCD (greatest common divisor) can be constructed.

```
%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 15.4 *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 A to B, as per the rules of this game.

```
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).
```

15.4 List Manipulation

The lists are 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. This may include also other lists. 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, [])))
```

Head and Tail. The first element of a list is called its *head* and the remaining list is called the *tail*. An empty list does not have a head. A list just containing a single element has a head (namely that particular single element) and its tail is the empty list. A variant of the list notation allows for convenient addressing of both head and tail of a list. This is done by using the separator | (bar).

```
?- [1, 2, 3, 4, 5] = [Head | Tail].
Head = 1
Tail = [2, 3, 4, 5]
Yes
```

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

Example 15.5 *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)
```

The search, along with unifications for appending two lists is shown in figure 15.2. The goal search shows alternate cycles of *unification* and calling of sub-goal. As a result of the recursion, the *append* operation can be realized as follows. The terminal node is matched with the *fact*: $append([], L, L)$. Consequently, $L4 = L5 = [3, 4]$. On back substitution,

$$\begin{aligned} L3 &= [2|L4] \\ &= [2|[3, 4]] \\ &= [2, 3, 4]. \end{aligned}$$

$$\begin{aligned} Z &= [1|L3] \\ &= [1|[2, 3, 4]] \\ &= [1, 2, 3, 4]. \end{aligned}$$

□

The more examples are as follows, which are self explanatory.

```
?- append(X, Y, [a, b, c, d]).
X = []
Y = [a, b, c, d] ;
X = [a]
Y = [b, c, d] ;
```

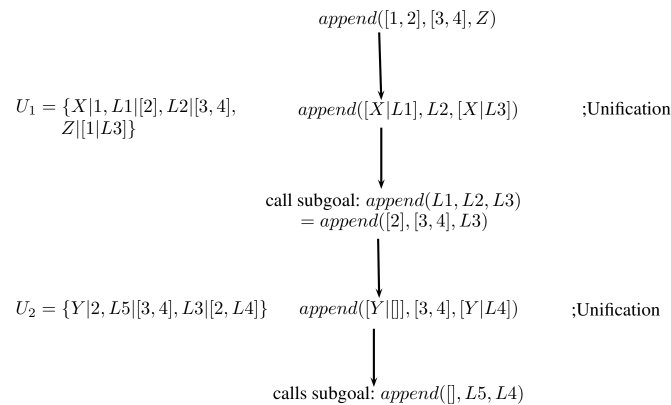

Figure 15.2: Prolog search for appending two lists.

```
X = [a, b]
Y = [c, d] ;
X = [a, b, c]
Y = [d] ;
X = [a, b, c, d]
Y = [] ;
No
```

```
?- member(dog, [elephant, horse, donkey, dog, monkey]).
Yes
```

```
?- reverse([1, 2, 3, 4, 5], X).
X = [5, 4, 3, 2, 1]
Yes
```

15.5 Arithmetic Expressions

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

15.6 Backtracking, Cuts and Negation

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 15.6 *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. □

15.7 Efficiency Considerations for Prolog Programs

For a given goal, prolog explores the premises for rules in the knowledge-base, making the goal true. If there are premises $p_1 \wedge p_2 \wedge \cdots \wedge p_n$, it fully explores the premise (called choice point) p_i before proceeding to p_{i+1}.

The solution through prolog is unification, and binding of variables, pushing and retrieving the stack, associated with backtracking. When a search fails, prolog will backtrack to the previous choice point, followed with possibly unbounding of some of the variables. It always keeps track of all the bound variables at any moment, and those kept in the stack. In addition, it has to manage the index for fast searching of predicates. This is called trail. Accordingly, even the most efficient prolog interpreters consume thousands of machine instructions for even the simple unifications and matching.

For the huge task, and due to nature of computing, required, the normal processors give very poor performance to prolog programs. Hence, the prolog programs are compiled into intermediate programs, called WAM (Warren Abstract machine). WAM helps prolog running faster as well as making it parallel.

Prolog may some times lead to incomplete loops.

The true version of prolog is called *pure Prolog.* It is obtained from a variation of the backward chaining algorithm that allows Horn clauses with the following rules and conventions:

- The Selection Rule is to select the leftmost literals in the goal.

- The Search Rule is to consider the clauses in the order they appear in the current list of clauses, from top to bottom.
- Negation as Failure, that is, Prolog assumes that a literal L is proven if it is unable to prove $\neg L$.
- Terms can be set equal to variables but not in general to other terms. For example, we can say that $x = A, x = F(B)$ but we cannot say that A=F(B).

These rules makes fast processing. But, unfortunately, the Pure Prolog inference Procedure is *Sound* but *not Complete*. This can be seen by the following example. Using this we are unable to derive in Prolog that $P(a, c)$ because we get caught in an ever deepening depth-first search.

$P(a, b).$
$P(c, b).$
$P(Y, X) : -P(X, Y).$
$P(X, Z) : -P(X, Y), P(Y, Z).$

Actual Prolog: Actual Prolog differs from pure Prolog in three major respects:

- There are additional functionalities besides theorem proving, such as functions to assert statements, functions to do arithmetic, functions to do I/O.
- The "cut" operator allows the user to prune branches of the search tree.
- The unification routine is not quite correct, in that it does not check for circular bindings e.g. $X \to Y, Y \to f(X)$.

References

[BRT07] I. Bratko, "PROLOG - Programming for Artificial Intelligence," *3rd Edition, Pearson Education, India*, 2007.