Module 7

Knowledge Representation and Logic – (Rule based Systems)

7.1 Instructional Objective

- The students should understand the use of rules as a restricted class of first order logic statements
- The student should be familiar with the concept of Horn clause
- Students should be able to understand and implement the following reasoning algorithm
    - Forward chaining
    - Backward chaining
- Students should understand the nature of the PROLOG programming language and the reasoning method used in it.
- Students should be able to write elementary PROLOG programs
- Students should understand the architecture and organization of expert systems and issues involved in designing an expert system

At the end of this lesson the student should be able to do the following:

- Represent a knowledge base as a set of rules if possible
- Apply forward/backward chaining algorithm as suitable
- Write elementary programs in PROLOG
- Design expert systems

Lesson 17

Rule based Systems - I

Version 1 CSE IIT, Kharagpur

7.2 Rule Based Systems

Instead of representing knowledge in a relatively declarative, static way (as a bunch of things that are true), rule-based system represent knowledge in terms of a bunch of rules that tell you what you should do or what you could conclude in different situations. A rule-based system consists of a bunch of IF-THEN rules, a bunch of facts, and some interpreter controlling the application of the rules, given the facts. Hence, this are also sometimes referred to as production systems. Such rules can be represented using Horn clause logic.

There are two broad kinds of rule system: forward chaining systems, and backward chaining systems. In a forward chaining system you start with the initial facts, and keep using the rules to draw new conclusions (or take certain actions) given those facts. In a backward chaining system you start with some hypothesis (or goal) you are trying to prove, and keep looking for rules that would allow you to conclude that hypothesis, perhaps setting new subgoals to prove as you go. Forward chaining systems are primarily data-driven, while backward chaining systems are goal-driven. We'll look at both, and when each might be useful.

7.2.1 Horn Clause Logic

There is an important special case where inference can be made substantially more focused than in the case of general resolution. This is the case where all the clauses are *Horn clauses.*

Definition: A *Horn clause* is a clause with at most one positive literal.

Any Horn clause therefore belongs to one of four categories:

- A *rule*: 1 positive literal, at least 1 negative literal. A rule has the form "~P1 V ~P2 V ... V ~Pk V Q". This is logically equivalent to "[P1^P2^ ... ^Pk] => Q"; thus, an if-then implication with any number of conditions but one conclusion. Examples: "~man(X) V mortal(X)" (All men are mortal); "~parent(X,Y) V ~ancestor(Y,Z) V ancestor(X,Z)" (If X is a parent of Y and Y is an ancestor of Z then X is an ancestor of Z.)
- A *fact* or *unit*: 1 positive literal, 0 negative literals. Examples: "man(socrates)", "parent(elizabeth,charles)", "ancestor(X,X)" (Everyone is an ancestor of themselves (in the trivial sense).)
- A *negated goal* : 0 positive literals, at least 1 negative literal. In virtually all implementations of Horn clause logic, the negated goal is the negation of the statement to be proved; the knowledge base consists entirely of facts and goals. The statement to be proven, therefore, called the goal, is therefore a single unit or the conjuction of units; an *existentially* quantified variable in the goal turns into a *free* variable in the negated goal. E.g. If the goal to be proven is "exists (X) male(X) ^ ancestor(elizabeth,X)" (show that there exists a male descendent of Elizabeth) the negated goal will be "~male(X) V ~ancestor(elizabeth,X)".

- The null clause: 0 positive and 0 negative literals. Appears only as the end of a resolution proof.

Now, if resolution is restricted to Horn clauses, some interesting properties appear. Some of these are evident; others I will just state and you can take on faith.

I. If you resolve Horn clauses A and B to get clause C, then the positive literal of A will resolve against a negative literal in B, so the only positive literal left in C is the one from B (if any). Thus, the resolvent of two Horn clauses is a Horn clause.

II. If you resolve a negated goal G against a fact or rule A to get clause C, the positive literal in A resolves against a negative literal in G. Thus C has no positive literal, and thus is either a negated goal or the null clause.

III. Therefore: Suppose you are trying to prove Phi from Gamma, where ~Phi is a negated goal, and Gamma is a knowledge base of facts and rules. Suppose you use the set of support strategy, in which no resolution ever involves resolving two clauses from Gamma together. Then, inductively, every resolution combines a negated goal with a fact or rule from Gamma and generates a new negated goal. Moreover, if you take a resolution proof, and trace your way back from the null clause at the end to ~Phi at the beginning, since every resolution involves combining one negated goal with one clause from Gamma, it is clear that the sequence of negated goals involved can be linearly ordered. That is, the final proof, ignoring dead ends has the form

```
~Phi resolves with C1 from Gamma, generating negated goal P2
 P2 resolves with C2 from Gamma, generating negated goal P3
...
 Pk resolves with C2 from Gamma, generating the null clause.
```

IV. Therefore, the process of generating the null clause can be viewed as a state space search where:

- A state is a negated goal.
- A operator on negated goal P is to resolve it with a clause C from Gamma.
- The start state is ~Phi
- The goal state is the null clause.

V. Moreover, it turns out that it doesn't really matter which literal in P you choose to resolve. All the literals in P will have to be resolved away eventually, and the order doesn't really matter. (This takes a little work to prove or even to state precisely, but if you work through a few examples, it becomes reasonably evident.)

7.2.2 Backward Chaining

Putting all the above together, we formulate the following non-deterministic algorithm for resolution in Horn theories. This is known as backward chaining.

```
bc(in P0 : negated goal;
      GAMMA : set of facts and rules;)
{ if P0 = null then succeed;
  pick a literal L in P0;
  choose a clause C in GAMMA whose head resolves with L;
  P := resolve(P0,GAMMA);
  bc(P,GAMMA)
}
```

If bc(~Phi,Gamma) succeeds, then Phi is a consequence of Gamma; if it fails, then Phi is not a consequence of Gamma.

Moreover: Suppose that Phi contains existentially quantified variables. As remarked above, when ~Phi is Skolemized, these become free variables. If you keep track of the successive bindings through the successful path of resolution, then the final bindings of these variables gives you a *value* for these variables; all proofs in Horn theories are constructive (assuming that function symbols in Gamma are constructive.) Thus the attempt to prove a statement like "exists(X,Y) p(X,Y)^q(X,Y)" can be interpreted as "*Find* X and Y such that p(X,Y) and q(X,Y)."

The succcessive negated goals Pi can be viewed as negations of *subgoals* of Phi. Thus, the operation of resolving ~P against C to get ~Q can be interpreted, "One way to prove P would be to prove Q and then use C to infer P". For instance, suppose P is "mortal(socrates)," C is "man(X) => mortal(X)" and Q is "man(socrates)." Then the step of resolving ~P against C to get ~Q can be viewed as, "One way to prove mortal(socrates) would to prove man(socrates) and then combine that with C."

Propositional Horn theories can be decided in polynomial time. First-order Horn theories are only semi-decidable, but in practice, resolution over Horn theories runs much more efficiently than resolution over general first-order theories, because of the much restricted search space used in the above algorithm.

Backward chaining is complete for Horn clauses. If Phi is a consequence of Gamma, then there is a backward-chaining proof of Phi from Gamma.

7.2.3 Pure Prolog

We are now ready to deal with (pure) Prolog, the major Logic Programming Language. 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 if it is unable to prove (NOT L)
- Terms can be set equal to variables but not in general to other terms. For example, we can say that x=A and x=F(B) but we cannot say that A=F(B).

- Resolvents are added to the bottom of the list of available clauses.

These rules make for very rapid processing. Unfortunately:

The Pure Prolog Inference Procedure is Sound but not Complete

This can be seen by example. Given

- P(A,B)
- P(C,B)
- P(y,x) < = P(x,y)
- P(x,z) < = P(x,y),P(y,z)

we are unable to derive in Prolog that P(A,C) because we get caught in an ever deepening depth-first search.

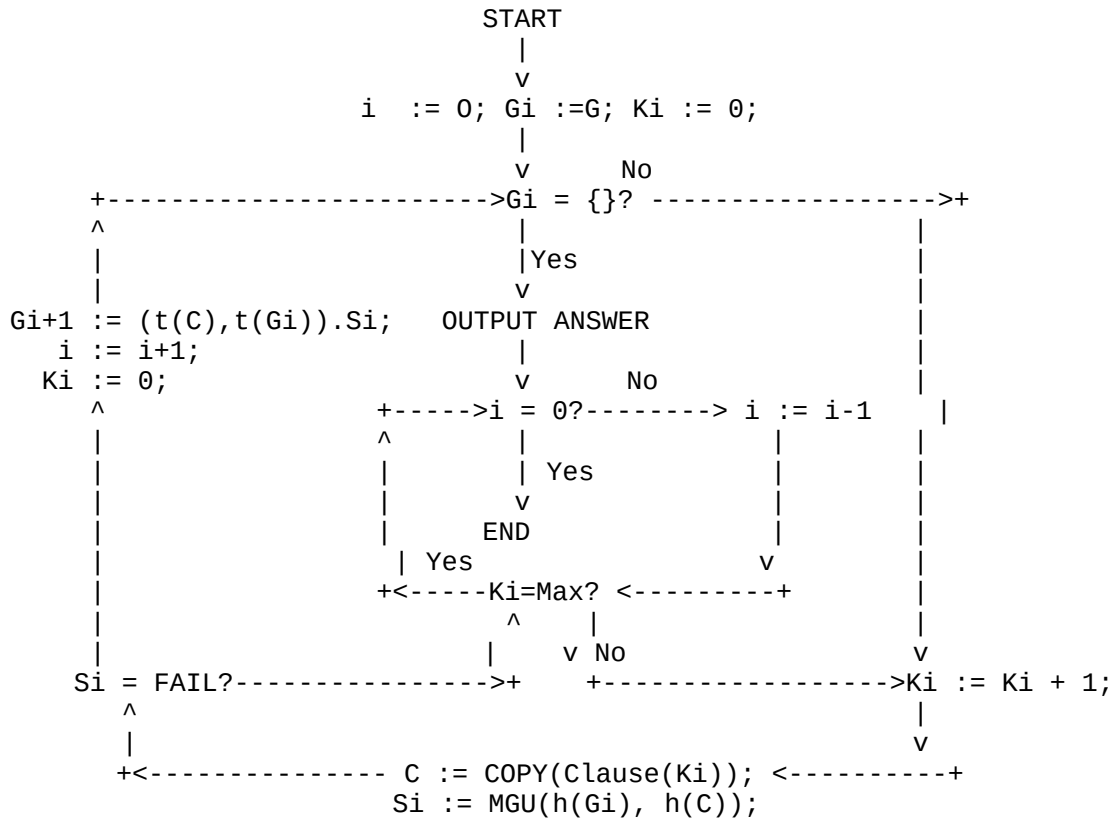
A Prolog "program" is a knowledge base Gamma. The program is invoked by posing a query Phi. The value returned is the bindings of the variables in Phi, if the query succeeds, or failure. The interpreter returns one answer at a time; the user has the option to request it to continue and to return further answers.

The derivation mechanism of Pure Prolog has a very simple form that can be described by the following flow chart.

Interpreter for Pure Prolog

Notational conventions:

- i: used to index literals in a goal
- K_i: indexes the clauses in the given program (i.e. set of clauses) P
- Max: the number of clauses in P
- h(G): the first literal of the goal G
- t(G): the rest of goal G, i.e. G without its first literal
- clause(K_i): the kith clause of the program



7.2.3.1 Real Prolog

Real Prolog systems differ from pure Prolog for a number of reasons. Many of which have to do with the ability in Prolog to modify the control (search) strategy so as to achieve efficient programs. In fact there is a dictum due to Kowalski:

Logic + Control = Algorithm

But the reason that is important to us now is that Prolog uses a Unification procedure which does not enforce the **Occur Test**. This has an unfortunate consequence that, while Prolog may give origin to efficient programs, but

Prolog is not Sound

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 -> Y, Y -> f(X).)

Notation: The clause "~p V ~q V r" is written in Prolog in the form "r :- p,q."

Example: Let Gamma be the following knowledge base:

```
1. ancestor(X,X).
2. ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
3. parent(george,sam).
4. parent(george,andy).
5. parent(andy,mary).
6. male(george).
7. male(sam).
8. male(andy).
9. female(mary).
```

Let Phi be the query "exists(Q) ancestor(george,Q) ^ female(Q)." (i.e. find a female descendant of george.) Then the Skolemization of Phi is "~ancestor(george,Q) V ~female(Q)." A Prolog search proceeds as follows: (The indentation indicates the subgoal structure. Note that the variables in clauses in Gamma are renamed each time.)

```
G0:  ~ancestor(george,Q) V ~female(Q).   Resolve with 1: Q=X1=george.
  G1: ~female(george)                   Fail. Return to G0.
G0:   ~ancestor(george,Q)  V  ~female(Q).    Resolve  with  2:  X2=george.
Z2=Q.
  G2:  ~parent(george,Y2) V ~ancestor(Y2,Q) V ~female(Q).
                                        Resolve with 3: Y2=sam.
    G3: ~ancestor(sam,Q) V ~female(Q).   Resolve with 1: Q=X3=sam.
       G4: ~female(sam).                 Fail. Return to G3.
    G3: ~ancestor(sam,Q) V ~female(Q).   Resolve with 2: X4=sam, Z4=Q
       G5: ~parent(sam,Y2) V ~ancestor(Y2,Q) V ~female(Q).
                                        Fail. Return to G3.
    G3: ~ancestor(sam,Q) V ~female(Q).   Fail. Return to G2.
  G2: ~parent(george,Y2) V ~ancestor(Y2,Q) V ~female(Q).
                                        Resolve with 4: Y2=andy.
    G6: ~ancestor(andy,Q) V ~female(Q).  Resolve with 1: X5=Q=andy
       G7: ~female(andy).                Fail. Return to G6.
    G6: ~parent(andy,Y6) V ~ancestor(Y6,Q) V ~female(Q).
                                        Resolve with 5: Y6=mary.
       G8:   ~ancestor(mary,Q)   V   ~female(mary).   Resolve   with   1:
X7=Q=mary.
         G9: ~female(mary)              Resolve with 9.
            Null.
```

Return the binding Q=mary.

7.2.4 Forward chaining

An alternative mode of inference in Horn clauses is *forward chaining* . In forward chaining, one of the resolvents in every resolution is a fact. (Forward chaining is also known as "unit resolution.")

Forward chaining is generally thought of as taking place in a dynamic knowledge base, where facts are gradually added to the knowledge base Gamma. In that case, forward chaining can be implemented in the following routines.

```
procedure add_fact(in F; in out GAMMA)
   /* adds fact F to knowledge base GAMMA and forward chains */
if F is not in GAMMA then {
   GAMMA := GAMMA union {F};
   for each rule R in GAMMA do {
      let ~L be the first negative literal in R;
      if L unifies with F then
       then { resolve R with F to get C;
              if C is a fact then add_fact(C,GAMMA)
              else /* C is a rule */ add_rule(C,GAMMA)
            }
      }
   }
end add_fact.

procedure add_rule(in R; in out GAMMA)
   /* adds rule R to knowledge base GAMMA and forward chains */
if R is not in GAMMA then {
   GAMMA := GAMMA union {R};
   let ~L be the first negative literal in R;
   for each fact F in GAMMA do
     if L unifies with F
       then { resolve R with F to get C;
              if C is a fact then add_fact(C,GAMMA)
              else /* C is a rule */ add_rule(C,GAMMA)
            }
   }
end add_fact.

procedure answer_query(in Q, GAMMA) return boolean /* Success or
failure */
{ QQ := {Q} /* A queue of queries
  while QQ is non-empty do {
      Q1 := pop(QQ);
      L1 := the first literal in Q1;
      for each fact F in GAMMA do
        if F unifies with L
          then { resolve F with Q1 to get Q2;
                 if Q2=null then return(true)
                            else add Q2 to QQ;
               }
      }
  return(false)
```

The forward chaining algorithm may not terminate if GAMMA contains recursive rules.

Forward chaining is complete for Horn clauses; if Phi is a consequence of Gamma, then there is a forward chaining proof of Phi from Gamma. To be sure of finding it if Gamma contains recursive rules, you have to modify the above routines to use an exhaustive search technique, such as a breadth-first search.

In a forward chaining system the facts in the system are represented in a *working memory* which is continually updated. Rules in the system represent possible actions to take when specified conditions hold on items in the working memory - they are sometimes called condition-action rules. The conditions are usually *patterns* that must *match* items in the working memory, while the actions usually involve *adding* or *deleting* items from the working memory.

The interpreter controls the application of the rules, given the working memory, thus controlling the system's activity. It is based on a cycle of activity sometimes known as a *recognise-act* cycle. The system first checks to find all the rules whose conditions hold, given the current state of working memory. It then selects one and performs the actions in the action part of the rule. (The selection of a rule to fire is based on fixed strategies, known as *conflict resolution* strategies.) The actions will result in a new working memory, and the cycle begins again. This cycle will be repeated until either no rules fire, or some specified goal state is satisfied.

Conflict Resolution Strategies

A number of conflict resolution strategies are typically used to decide which rule to fire. These include:

- Don't fire a rule twice on the same data.
- Fire rules on more recent working memory elements before older ones. This allows the system to follow through a single chain of reasoning, rather than keeping on drawing new conclusions from old data.
- Fire rules with more specific preconditions before ones with more general preconditions. This allows us to deal with non-standard cases. If, for example, we have a rule ``IF (bird X) THEN ADD (flies X)'' and another rule ``IF (bird X) AND (penguin X) THEN ADD (swims X)'' and a penguin called tweety, then we would fire the second rule first and start to draw conclusions from the fact that tweety swims.

These strategies may help in getting reasonable behavior from a forward chaining system, but the most important thing is how we write the rules. They should be carefully constructed, with the preconditions specifying as precisely as possible when different rules should fire. Otherwise we will have little idea or control of what will happen. Sometimes special working memory elements are used to help to control the behavior of the system. For example, we might decide that there are certain basic stages of processing in doing some task, and certain rules should only be fired at a given stage - we could have a special working memory element `(stage 1)` and add `(stage 1)` to the preconditions of all the relevant rules, removing the working memory element when that stage was complete.

Choice between forward and backward chaining

Forward chaining is often preferable in cases where there are many rules with the same conclusions. A well-known category of such rule systems are taxonomic hierarchies. E.g. the taxonomy of the animal kingdom includes such rules as:

```
animal(X) :- sponge(X).
animal(X) :- arthopod(X).
animal(X) :- vertebrate(X).
...
vertebrate(X) :- fish(X).
vertebrate(X) :- mammal(X)
...
mammal(X) :- carnivore(X)
...
carnivore(X) :- dog(X).
carnivore(X) :- cat(X).
...
```

(I have skipped family and genus in the hierarchy.)

Now, suppose we have such a knowledge base of rules, we add the fact "dog(fido)" and we query whether "animal(fido)". In forward chaining, we will successively add "carnivore(fido)", "mammal(fido)", "vertebrate(fido)", and "animal(fido)". The query will then succeed immediately. The total work is proportional to the height of the hierarchy. By contast, if you use backward chaining, the query "~animal(fido)" will unify with the first rule above, and generate the subquery "~sponge(fido)", which will initiate a search for Fido through all the subdivisions of sponges, and so on. Ultimately, it searches the entire taxonomy of animals looking for Fido.

In some cases, it is desirable to *combine* forward and backward chaining. For example, suppose we augment the above animal with features of these various categories:

```
breathes(X) :- animal(X).
...
backbone(X) :- vertebrate(X).
has(X,brain) :- vertebrate(X).
...
furry(X) :- mammal(X).
warm_blooded(X) :- mammal(X).
...
```

If all these rules are implemented as forward chaining, then as soon as we state that Fido is a dog, we have to add all his known properties to Gamma; that he breathes, is warm-blooded, has a liver and kidney, and so on. The solution is to mark these property rules as backward chaining and mark the hierarchy rules as forward chaining. You then implement the knowledge base with both the forward chaining algorithm, restricted to rules marked as forward chaining, and backward chaining rules, restricted to rules marked as backward chaining. However, it is hard to guarantee that such a mixed inference system will be complete.

AND/OR Trees

We will next show the use of AND/OR trees for inferencing in Horn clause systems. The problem is, given a set of axioms in Horn clause form and a goal, show that the goal can be proven from the axioms.

An AND/OR tree is a tree whose internal nodes are labeled either "AND" or "OR". A *valuation* of an AND/OR tree is an assignment of "TRUE" or "FALSE" to each of the leaves. Given a tree T and a valuation over the leaves of T, the values of the internal nodes and of T are defined recursively in the obvious way:

- An OR node is TRUE if at least one of its children is TRUE.
- An AND node is TRUE if all of its children are TRUE.

The above is an *unconstrained* AND/OR tree. Also common are *constrained* AND/OR trees, in which the leaves labeled "TRUE" must satisfy some kind of constraint. A *solution* to a constrained AND/OR tree is a valuation that satisfies the constraint and gives the tree the value "TRUE".

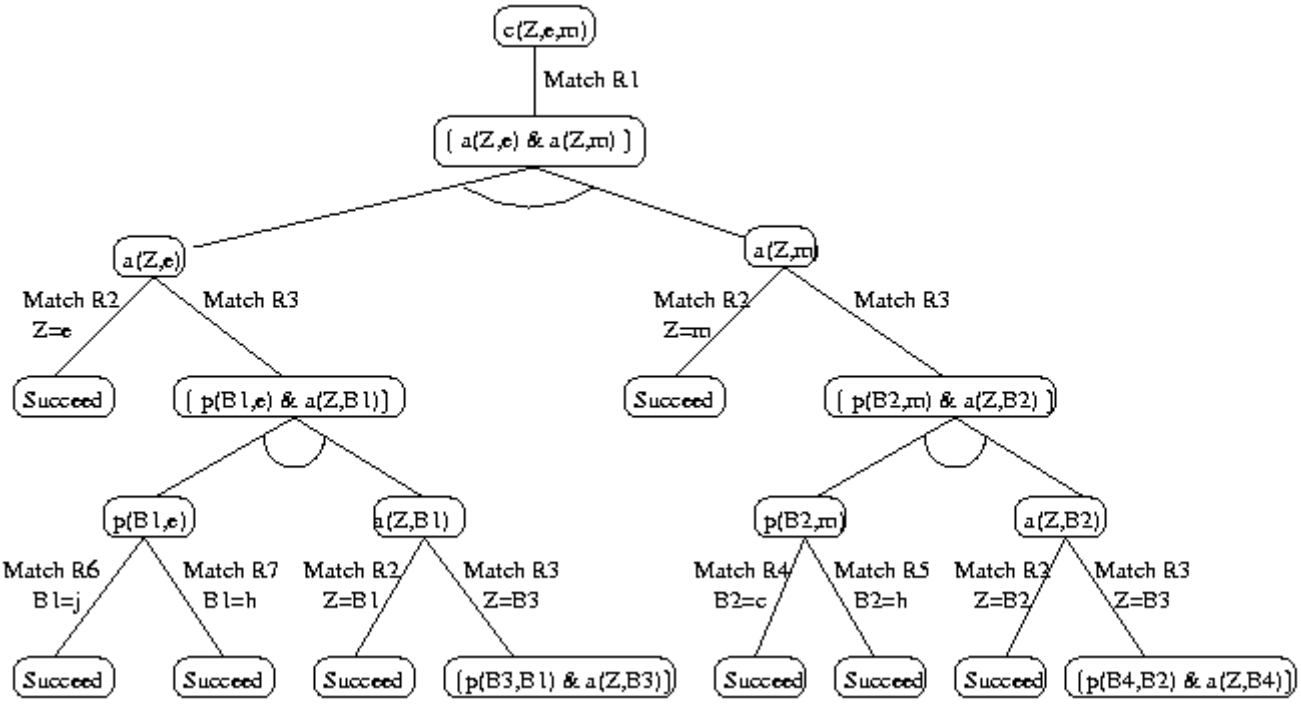
An OR node is a goal to be proven. A goal G has one downward arc for each rule R whose head resolves with G. This leads to an AND node. The children in the AND node are the literals in the tail of R. Thus, a rule is satisfied if all its subgoals are satisfied (the AND node); a goal is satisfied if it is established by one of its rules (the OR node). The leaves are unit clauses, with no tail, labeled TRUE, and subgoals with no matching rules, labeled FALSE . The constraint is that the variable bindings must be consistent. The figure below show the AND/OR tree corresponding to the following Prolog rule set with the goal "common_ancestor(Z,edward,mary)"

```
Axioms:

/* Z is a common ancestor of X and Y */
R1: common_ancestor(Z,X,Y) :- ancestor(Z,X), ancestor(Z,Y).

/* Usual recursive definition of ancestor, going upward. */
R2: ancestor(A,A).
R3: ancestor(A,C) :- parent(B,C), ancestor(A,B).

/* Mini family tree */
R4: parent(catherine,mary).
R5: parent(henry,mary).
R6: parent(jane,edward).
R7: parent(henry,edward).
```


c(Z,e,m)
Match R1
(a(Z,e) & a(Z,m))
a(Z,e)
a(Z,m)
Match R2
Z=e
Match R3
Match R2
Z=m
Match R3
Succeed
(p(B1,e) & a(Z,B1))
Succeed
(p(B2,m) & a(Z,B2))
p(B1,e)
a(Z,B1)
p(B2,m)
a(Z,B2)
Match R6
B1=j
Match R7
B1=h
Match R2
Z=B1
Match R3
Z=B3
Match R4
B2=c
Match R5
B2=h
Match R2
Z=B2
Match R3
Z=B3
Succeed
Succeed
Succeed
(p(B3,B1) & a(Z,B3))
Succeed
Succeed
Succeed
(p(B4,B2) & a(Z,B4))