5. Predicate Logic

Introduction

Predicate logic is used to represent Knowledge. Predicate logic will be met in Knowledge Representation Schemes and reasoning methods. There are other ways but this form is popular.

Propositional Logic

It is simple to deal with and decision procedure for it exists. We can represent real-world facts as logical propositions written as well-formed formulas.

To explore the use of predicate logic as a way of representing knowledge by looking at a specific example.

$$It\ is\ raining. \rightarrow RAINING$$
$$It\ is\ sunny. \rightarrow SUNNY$$
$$It\ is\ windy. \rightarrow WINDY$$
$$If\ it\ is\ raining\ then\ it\ is\ not\ sunny.\ :\ RAINING \rightarrow \neg SUNNY$$

$$Socrates\ is\ a\ man \rightarrow SOCRATESMAN$$
$$Plato\ is\ a\ man \rightarrow PLATOMAN$$

The above two statements becomes totally separate assertion, we would not be able to draw any conclusions about similarities between Socrates and Plato.

$$MAN(SOCRATES)$$
$$MAN(PLATO)$$

These representations reflect the structure of the knowledge itself. These use predicates applied to arguments.

$$All\ men\ are\ mortal \rightarrow MORTALMAN$$

It fails to capture the relationship between any individual being a man and that individual being a mortal.

We need variables and quantification unless we are willing to write separate statements.

Predicate:

A Predicate is a truth assignment given for a particular statement which is either true or false. To solve common sense problems by computer system, we use predicate logic.

Logic Symbols used in predicate logic

$\forall$ – *For all*

$\exists$ – *There exists*

$\rightarrow$ – *Implies*

$\neg$ – *Not*

$\vee$ – *OR*

$\wedge$ – *AND*

Predicate Logic

- Terms represent specific objects in the world and can be constants, variables or functions.
- Predicate Symbols refer to a particular relation among objects.
- Sentences represent facts, and are made of terms, quantifiers and predicate symbols.
- Functions allow us to refer to objects indirectly (via some relationship).
- Quantifiers and variables allow us to refer to a collection of objects without explicitly naming each object.
- Some Examples
    - Predicates: Brother, Sister, Mother , Father
    - Objects: Bill, Hillary, Chelsea, Roger
    - Facts expressed as atomic sentences a.k.a.

    literals:
        - Father(Bill,Chelsea)
        - Mother(Hillary,Chelsea)
        - Brother(Bill,Roger)
        - Father(Bill,Chelsea)

Variables and Universal Quantification

Universal Quantification allows us to make a statement about a collection of objects:

- $\forall$x Cat(x) $\Rightarrow$ Mammel(x) : All cats are mammels
- $\forall$x Father(Bill,x) $\Rightarrow$ Mother(Hillary,x) : All of Bill's kids are also Hillary's kids.

Variables and Existential Quantification

Existential Quantification allows us to state that an object does exist (without naming it):

- $\exists$x Cat(x) $\wedge$ Mean(x) : There is a mean cat.
- $\exists$x Father(Bill,x) $\wedge$ Mother(Hillary,x) : There is a kid whose father is Bill and whose mother is Hillary

Nested Quantification

- $\forall x,y\ Parent(x,y) \rightarrow Child(y,x)$
- $\forall x\ \exists y\ Loves(x,y)$
- $\forall x\ [Passtest(x) \vee (\exists x\ ShootDave(x))]$

Functions

- Functions are terms - they refer to a specific object.
- We can use functions to symbolically refer to objects without naming them.
- Examples:

  fatherof(x) age(x) times(x,y) succ(x)

- Using functions
  - $\forall\ x\ Equal(x,x)$
  - $Equal(factorial(0),1)$
  - $\forall\ x\ Equal(factorial(s(x)),\ times(s(x),factorial(x)))$

If we use logical statements as a way of representing knowledge, then we have available a good way of reasoning with that knowledge.

Representing facts with Predicate Logic

1) Marcus was a man man(Markus)
2) Marcus was a Pompeian pompeian(Markus)
3) All Pompeians were Romans $\forall x : pompeian(x) \rightarrow roman(x)$
4) Caeser was a ruler. ruler(caeser)
5) All romans were either loyal to caeser or hated him.

   $\forall x : roman(x) \rightarrow loyalto(x, caeser) \vee hate(x, caeser)$

6) Everyone loyal to someone. $\forall x, \exists y : loyalto(x, y)$
7) People only try to assassinate rulers they are not loyal to.

$$\forall x, \forall y: Person(x) \wedge Ruler(y) \wedge try\_assassinate(x, y) \ \rightarrow\ \neg Loyal\_to(x, y)$$

8) Marcus try to assassinate Ceaser $try\_assacinate(Marcus, Ceaser)$

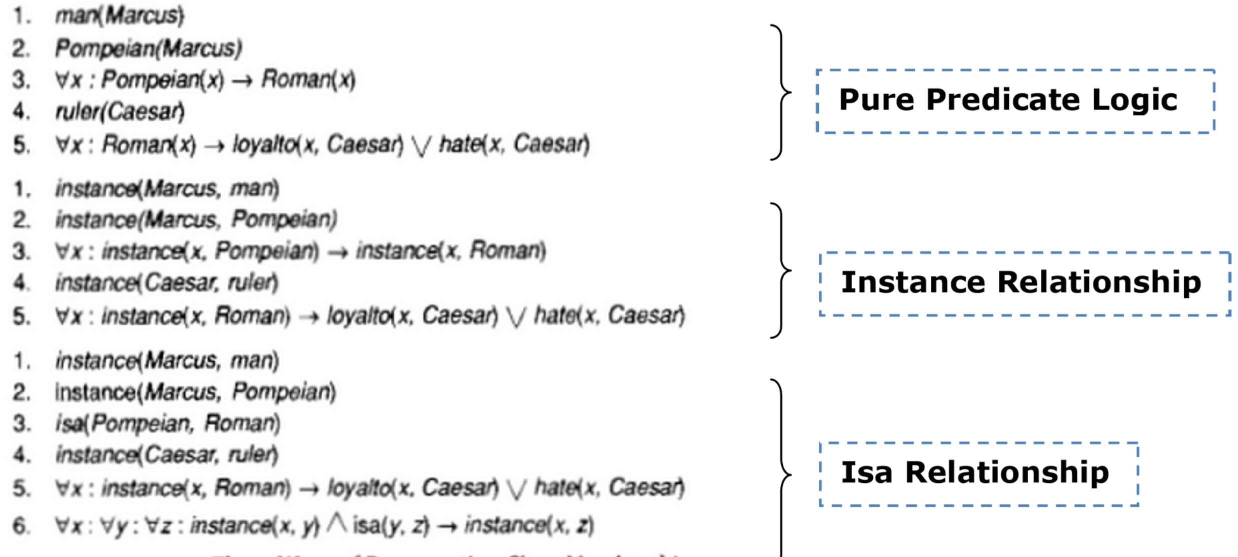
Q. Prove that Marcus is not loyal to Ceaser by backward substitution

$$4. \quad \neg Loyal\_to(Marcus, Ceaser)$$
$$\uparrow$$
$$5. \quad Person(Marcus) \land Ruler(Ceaser) \land Try\_assacinate(Marcus, Ceaser)$$
$$6. \quad \uparrow$$
$$7. \quad Person(Marcus) \land Ruler(Ceaser)$$
$$8. \quad \uparrow$$
$$9. \quad Person(Marcus)$$

Representing Instance and Isa Relationships

Two attributes isa and instance play an important role in many aspects of knowledge representation. The reason for this is that they support property inheritance.

isa - used to show class inclusion, e.g. isa (mega_star,rich). instance - used to show class membership, e.g. instance(prince,mega_star).



1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $\forall x : Pompeian(x) \rightarrow Roman(x)$
4. $ruler(Caesar)$
5. $\forall x : Roman(x) \rightarrow loyalto(x, Caesar) \lor hate(x, Caesar)$

Pure Predicate Logic

1. $instance(Marcus, man)$
2. $instance(Marcus, Pompeian)$
3. $\forall x : instance(x, Pompeian) \rightarrow instance(x, Roman)$
4. $instance(Caesar, ruler)$
5. $\forall x : instance(x, Roman) \rightarrow loyalto(x, Caesar) \lor hate(x, Caesar)$

Instance Relationship

1. $instance(Marcus, man)$
2. $instance(Marcus, Pompeian)$
3. $isa(Pompeian, Roman)$
4. $instance(Caesar, ruler)$
5. $\forall x : instance(x, Roman) \rightarrow loyalto(x, Caesar) \lor hate(x, Caesar)$
6. $\forall x : \forall y : \forall z : instance(x, y) \land isa(y, z) \rightarrow instance(x, z)$

Isa Relationship

Three Ways of Representing Class Membership

In the figure above,

- ➢ The first five sentences of the represent the *pure predicate logic*. In these representations, class membership is represented with unary predicates (such as Roman), each of which corresponds to a class. Asserting that P(x) is true is equivalent to asserting that x is an instance of P.
- ➢ The second part of the figure contains representations that use the *instance* predicate explicitly. The predicate instance is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs. But these representations do not use an explicit isa predicate.
- ➢ The third part contains representations that use both the instance and *isa* predicates explicitly. The use of the isa predicate simplifies the representation of sentence 3, but it requires that one additional axiom be provided. This additional axiom describes how an instance relation and an isa relation can be combined to derive a new instance relation.

Computable Functions and Predicates

Prove that Marcus is dead now

This is fine if the number of facts is not very large or if the facts themselves are sufficiently unstructured that there is little alternative. But suppose we want to express simple facts, such as the following greater-than and less-than relationships:

gt(1,0)	lt(0,1)
gt(2,1)	lt(1,2)
gt(3,2)	lt(2,3)

Clearly we do not want to have to write out the representation of each of these facts individually. For one thing, there are infinitely many of them. But even if we only consider the finite number of them that can be represented, say, using a single machine word per number, it would be extremely inefficient to store explicitly a large set of statements when we could, instead, so easily compute each one as we need it. Thus it becomes useful to augment our representation by these *computable predicates*.

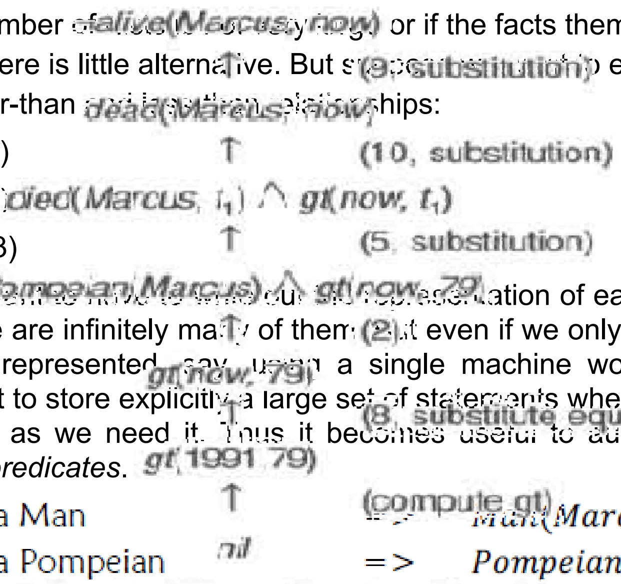
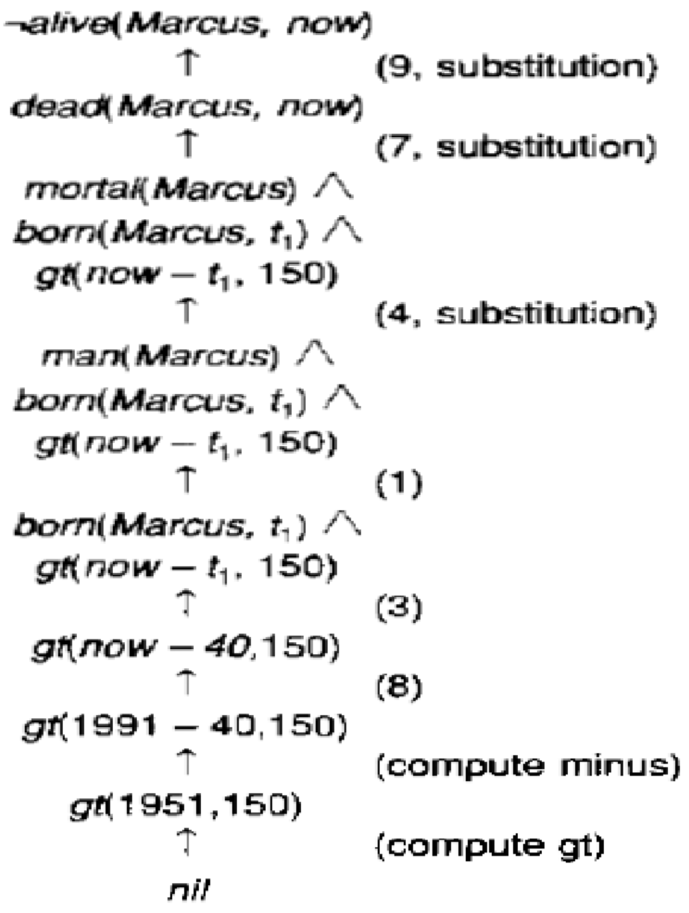


Fig. 5.50 One Way of Proving That Marcus Is Dead

1. Marcus was a Man => $Man(Marcus)$
2. Marcus was a Pompeian => $Pompeian(Marcus)$
3. Marcus b
4. All m => $Mortal(x)$
5. All Po $79))$
6. No m $\forall$ $) \rightarrow died(x, t2)$
7. It is n
8. Alive $e(x, t))$
9. If son $x, t2)$



Another Way of Proving That Marcus is Dead

Resolution:

A procedure to prove a statement, Resolution attempts to show that Negation of Statement gives Contradiction with known statements. It simplifies proof procedure by first converting the statements into canonical form. Simple iterative process; at each step, 2 clauses called the parent clauses are compared, yielding a new clause that has been inferred from them.

Resolution inference rule

$$\frac{(\alpha \vee \neg\beta) \wedge (\gamma \vee \beta) \text{ premise}}{(\alpha \vee \gamma) \text{ conclusion}}$$

Resolution refutation:

- Convert all sentences to CNF (conjunctive normal form)
- Negate the desired conclusion (converted to CNF)

  Apply resolution rule until either
  - Derive false (a contradiction)
  - Can't apply any more

Resolution refutation is sound and complete

- If we derive a contradiction, then the conclusion follows from the axioms
- If we can't apply any more, then the conclusion cannot be proved from the axioms.

Sometimes from the collection of the statements we have, we want to know the answer of this question - "Is it possible to prove some other statements from what we actually know?" In order to prove this we need to make some inferences and those other statements can be shown true using Refutation proof method i.e. proof by contradiction using Resolution. So for the asked goal we will negate the goal and will add it to the given statements to prove the contradiction.

So resolution refutation for propositional logic is a complete proof procedure. So if the thing that you're trying to prove is, in fact, entailed by the things that you've assumed, then you can prove it using resolution refutation.

Clauses:

- Resolution can be applied to certain class of wff called clauses.
- A clause is defined as a wff consisting of disjunction of literals.

Conjunctive Normal Form or Clause Normal Form:

Clause form is an approach to Boolean logic that expresses formulas as conjunctions of clauses with an AND or OR. Each clause connected by a conjunction or AND must be wither a literal or contain a disjunction or OR operator. In clause form, a statement is a series of ORs connected by ANDs.

A statement is in conjunctive normal form if it is a conjunction (sequence of ANDs) consisting of one or more conjuncts, each of which is a disjunction (OR) of one or more literals (i.e., statement letters and negations of statement letters).

All of the following formulas in the variables A, B, C, D, and E are in conjunctive normal form:

- $\neg A \wedge (B \vee C)$
- $(A \vee B) \wedge (\neg B \vee C \vee \neg D) \wedge (D \vee \neg E)$
- $A \vee B$
- $A \wedge B$

Conversion to Clause Form:

$$\forall x : [Roman(x) \wedge know(x, Marcus)] \rightarrow$$
$$[hate(x, Caesar) \vee (\forall y : \exists z : hate(y, z) \rightarrow thinkcrazy(x, y))]$$

➔ Clause Form:

$$\neg Roman(x) \wedge \neg know(x, Marcus) \vee$$
$$hate(x, Caesar) \vee \neg hate(y, z) \vee thinkcrazy(x, z)$$

Algorithm:

1. Eliminate implies relation ($\rightarrow$) Using (Ex: $\rightarrow$ =>)

$$\forall x : \neg[Roman(x) \wedge know(x, Marcus)] \vee$$
$$[hate(x, Caesar) \vee (\forall y : \neg(\exists z : hate(y, z)) \vee thinkcrazy(x,y))]$$

2. Reduce the scope of each to a single term

$$\neg(\neg P) = P$$
$$\neg(a \vee b) = \neg a \wedge \neg b$$
$$\neg(a \wedge b) = \neg a \vee \neg b$$

$$\forall x : [\neg Roman(x) \vee \neg know(x, Marcus)] \vee$$
$$[hate(x, Caesar) \vee (\forall y : \forall z : \neg hate(y, z) \vee thinkcrazy(x, y))]$$

3. Standardize variables so that each quantifier binds a unique variable.

$\forall x: P(x) \vee \forall x: Q(x)$ can be converted to

$\forall x: P(x) \vee \forall y: Q(y)$

4. Move all quantifiers to the left of the formulas without changing their relative order.

$$\exists x: \forall x, \forall y : P(x) \vee Q(x)$$

$$\forall x : \forall y : \forall z : [\neg Roman(x) \vee \neg know(x\ Marcus)] \vee$$
$$[hate(x, Caesar) \vee (\neg hate(y, z) \vee thinkcrazy(x,y))]$$

5. Eliminate existential quantifiers. We can eliminate the quantifier by substituting for the variable a reference to a function that produces the desired value. $\exists$y: President(y) => President(S1)

$\exists$y: President(y) => President(S1)

$\forall$x,$\exists$y: Fatherof(y,x) =>$\forall$x: Fatherof(S2(s),x)

President(func()) → func is called a skolem function.

In general the function must have the same number of arguments as the number of universal quantifiers in the current scope.

Skolemize to remove existential quantifiers. This step replaces existentially quantified variables by Skolem functions. For example, convert (x)P(x) to P(c) where c is a brand new constant symbol that is not used in any other sentence (c is called a Skolem constant). More generally, if the existential quantifier is within the scope of a universal quantified variable, then introduce a Skolem function that depends on the universally quantified variable. For example, " x y P(x,y) is converted to " x P(x, f(x)). f is called a Skolem function, and must be a brand new function name that does not occur in any other part of the logic sentence.

6. Drop the prefix. At this point, all remaining variables are universally quantified.

$$P(x) \vee Q(x)$$

$$[\neg Roman(x) \vee \neg know(x, Marcus)] \vee [hate(x, Caesar) \vee (\neg hate(y, z) \vee thinkcrazy(x, y))]$$

7. Convert the matrix into a conjunction of disjunctions.

$$(a \vee b) \vee c = a \vee (b \vee c) \qquad \textit{Associative Law}$$
$$(a \vee b) \wedge c = (a \wedge c) \vee (b \wedge c) \qquad \textit{Distributive Laws}$$
$$(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$$
$$a \vee b = b \vee a \qquad \textit{Commutative Law}$$

$$\neg Roman(x) \vee \neg know(x, Marcus) \vee hate(x, Caesar) \vee \neg hate(y, z) \vee thinkcrazy(x, y)$$

8. Create a separate clause corresponding to each conjunct in order for a well formed formula to be true, all the clauses that are generated from it must be true.
9. Standardize apart the variables in set of clauses generated in step 8. Rename the variables. So that no two clauses make reference to same variable.

Convert the statements to clause form

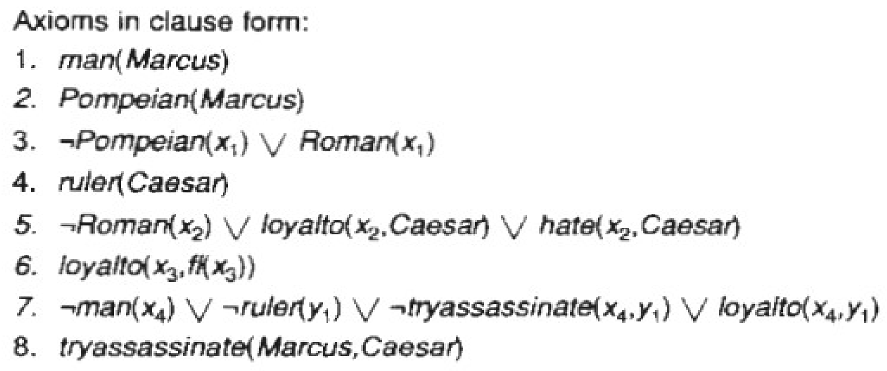
1. man(marcus)
2. pompeian(marcus)
3. ∀ pompeian(x) → roman(x)
4. ruler(caeser)
5. ∀x: roman(x) → loyalto(x,caeser) V hate(x,caeser)

∀x, ∃y: loyalto(x,y)

∀x, ∀y: person(x) ∧ ruler(y) ∧ tryassacinate(x,y) → ¬ loyalto(x,y)

tryassacinate(marcus, caeser)

The resultant clause form is



Axioms in clause form:

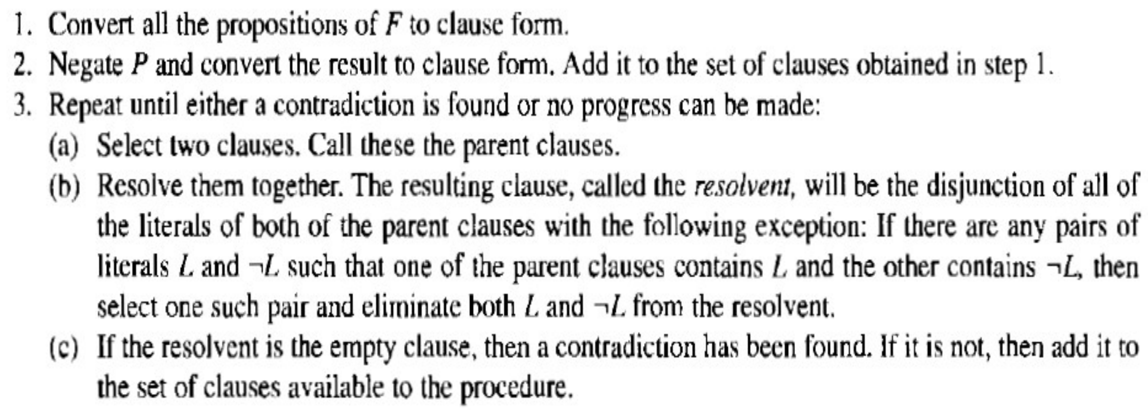
1. *man*(*Marcus*)
2. *Pompeian*(*Marcus*)
3. $\neg Pompeian(x_1) \vee Roman(x_1)$
4. *ruler*(*Caesar*)
5. $\neg Roman(x_2) \vee loyalto(x_2, Caesar) \vee hate(x_2, Caesar)$
6. $loyalto(x_3, f1(x_3))$
7. $\neg man(x_4) \vee \neg ruler(y_1) \vee \neg tryassassinate(x_4, y_1) \vee loyalto(x_4, y_1)$
8. *tryassassinate*(*Marcus*, *Caesar*)

Basis of Resolution:

Resolution process is applied to pair of parent clauses to produce a derived clause. Resolution procedure operates by taking 2 clauses that each contain the same literal. The literal must occur in the positive form in one clause and negative form in the other. The resolvent is obtained by combining all of the literals of two parent clauses except ones that cancel. If the clause that is produced in an empty clause, then a contradiction has been found.
Eg: winter and   winter will produce the empty clause.

If a contradiction exists, then eventually it will be found. Of course, if no contradiction exists, it is possible that the procedure will never terminate, although as we will see, there are often ways of detecting that no contradiction exists.

Resolution in Propositional Logic:



1. Convert all the propositions of *F* to clause form.
2. Negate *P* and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found or no progress can be made:
   (a) Select two clauses. Call these the parent clauses.
   (b) Resolve them together. The resulting clause, called the *resolvent*, will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals L and $\neg L$ such that one of the parent clauses contains L and the other contains $\neg L$, then select one such pair and eliminate both L and $\neg L$ from the resolvent.
   (c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

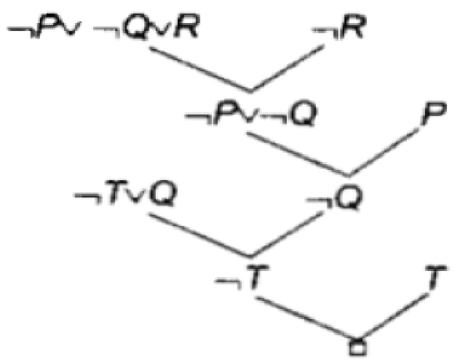
Example: Consider the following axioms

$$P \qquad (P^{\wedge}Q) \rightarrow R \qquad (S^{V}T) \rightarrow Q \quad T$$

Convert them into clause form and prove that R is true

1. P
2. $(P \wedge Q) \rightarrow R \quad => \quad \neg (P \wedge Q) \vee R \quad \rightarrow \quad \neg P \vee \neg Q \vee R$
3. $(S \vee T) \rightarrow R$

   $\neg (S \vee T) \vee Q \quad -> \quad (\neg S \wedge \neg T) \vee Q \quad -> \quad (\neg S \vee Q) \wedge (\neg T \vee Q)$
4. T



$\neg$ ***R is contradiction.*** Hence, R is true.

Unification Algorithm

- In propositional logic it is easy to determine that two literals cannot both be true at the same time.
- Simply look for L and ~L . In predicate logic, this matching process is more complicated, since bindings of variables must be considered.
- In order to determine contradictions we need a matching procedure that compares two literals and discovers whether there exist a set of substitutions that makes them identical.
- There is a recursive procedure that does this matching. It is called Unification algorithm.
- The process of finding a substitution for predicate parameters is called unification.
- We need to know:
  - that 2 literals can be matched.
  - the substitution is that makes the literals identical.
- There is a simple algorithm called the unification algorithm that does this.

The Unification Algorithm

1. Initial predicate symbols must match.
2. For each pair of predicate arguments:
   - Different constants cannot match.
   - A variable may be replaced by a constant.
   - A variable may be replaced by another variable.
   - A variable may be replaced by a function as long as the function does not contain an instance of the variable.

- When attempting to match 2 literals, all substitutions must be made to the entire literal.

- There may be many substitutions that unify 2 literals; the most general unifier is always desired.

Unification Example:

$P(x)$ and $P(y)$: substitution $= (x/y) \rightarrow$ substitution x for y

$P(x,x)$ and $P(y,z)$: $P(z/y)(y/x) \rightarrow y$ for x, then z for y

$P(f(x))$ and $P(x)$: can't do it!

$P(x) \vee Q(Jane)$ and $P(Bill) \vee Q(y)$: $(Bill/x, Jane/y)$

$Father(Bill, Chelsea)\ \neg Father(Bill, x) \vee Mother(Hillary, x)$
$Man(Marcus) \neg Man(x) \vee Mortal(x)$

$Loves(father(a), a) \neg Loves(x, y) \vee Loves(y, x)$

The object of the Unification procedure is to discover at least one substitution that causes two literals to match. Usually, if there is one such substitution there are many

$hate(x, y)$
$hate(Marcus, z)$
could be unified with any of the following substitutions:
$(Marcus/x, z/y)$
$(Marcus/x, y/z)$
$(Marcus/x, Caeser/y, Caeser/z)$
$(Marcus/x, Polonius/y, Polunius/z)$

In Unification algorithm each literal is represented as a list, where first element is the name of a predicate and the remaining elements are arguments. The argument may be a single element (atom) or may be another list.

The unification algorithm recursively matches pairs of elements, one pair at a time. The matching rules are:

- Different constants, functions or predicates cannot match, whereas identical ones can.
- A variable can match another variable, any constant or a function or predicate expression, subject to the condition that the function or [predicate expression must not contain any instance of the variable being matched (otherwise it will lead to infinite recursion).

- The substitution must be consistent. Substituting y for x now and then z for x later is inconsistent. (a substitution y for x written as y/x)

Algorithm: Unify(Ll, L2)

```
1. If L1 or L2 are both variables or constants, then:
   (a) If L1 and L2 are identical, then return NIL.
   (b) Else if L1 is a variable, then if L1 occurs in L2 then return {FAIL}, else return (L2/L1).
   (c) Else if L2 is a variable then if L2 occurs in L1 then return {FAIL}, else return (L1/L2).
   (d) Else return {FAIL}.
2. If the initial predicate symbols in L1 and L2 are not identical, then return {FAIL).
3. If LI and L2 have a different number of arguments, then return {FAIL}.
4. Set SUBST to NIL. (At the end of this procedure, SUBST will contain all the substitutions used to unify
   L1 and L2.)
5. For i ← 1 to number of arguments in L1:
   (a) Call Unify with the ith argument of L1 and the ith argument of L2, putting result in S.
   (b) If S contains FAIL then return {FAIL}.
   (c) If S is not equal to NIL then:
       (i) Apply S to the remainder of both L1 and L2.
       (ii) SUBST : = APPEND(S, SUBST).
6. Return SUBST.
```

Example:

Suppose we want to unify *p(X,Y,Y)* with *p(a,Z,b)*.

Initially *E* is *{p(X,Y,Y)=p(a,Z,b)}*.

The first time through the while loop, *E* becomes *{X=a,Y=Z,Y=b}*.

Suppose *X=a* is selected next.

Then *S* becomes*{X/a}* and *E* becomes *{Y=Z,Y=b}*.

Suppose *Y=Z* is selected.

Then *Y* is replaced by *Z* in *S* and *E*.

S becomes*{X/a,Y/Z}* and *E* becomes *{Z=b}*.

Finally *Z=b* is selected, *Z* is replaced by *b*, *S* becomes *{X/a,Y/b,Z/b}*, and *E* becomes empty.

The substitution *{X/a,Y/b,Z/b}* is returned as an MGU.

Unification:

$$\forall x: knows(John, x) \rightarrow hates(John, x)$$

$$knows(John, Jane)$$

$$\forall y: knows(y, Leonid)$$

$$\forall y: knows(y, mother(y))$$

$$\forall x: knows(x, Elizabeth)$$

$$UNIFY(knows(John, x), knows(John, Jane)) = \{Jane/x\}$$
$$UNIFY(knows(John, x), knows(y, Leonid)) = \{Leonid/x, John/y\}$$
$$UNIFY(knows(John, x), knows(y, mother(y))) = \{John/y, mother(John)/x\}$$
$$UNIFY(knows(John, x), knows(x, Elizabeth)) = FAIL$$

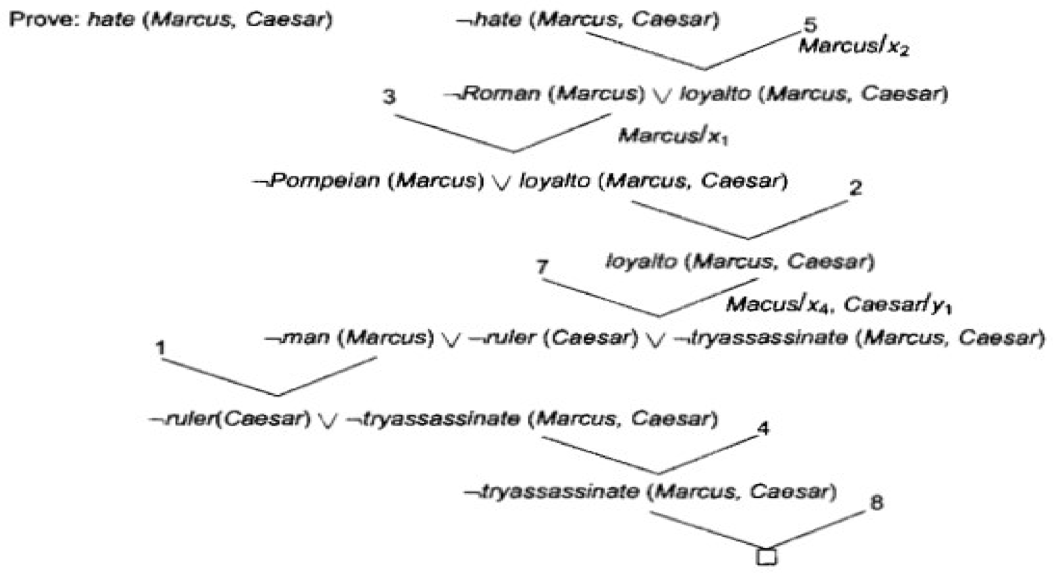
Resolution in Predicate Logic

- Two literals are contradictory if one can be unified with the negation of the other.
  - For example man(x) and man (Himalayas) are contradictory since man(x) and man(Himalayas) can be unified.
- In predicate logic unification algorithm is used to locate pairs of literals that cancel out.
- It is important that if two instances of the same variable occur, then they must be given identical substitutions

Algorithm: Resolution

1. Convert all the statements of *F* to clause form.
2. Negate *P* and convert the result to clause form. Add it to the set of clauses obttfHied in 1.
3. Repeat until either a contradiction is found, no progress can be made, or a prede- termined amount of effort has been expended.
   (a) Select two clauses. Call these the parent clauses.
   (b) Resolve them together. The resolvent will be the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals *T1* and ¬*T2* such that one of the parent clauses contains *T2* and the other contains *T1* and if *T1* and *T2* are unifiable, then neither *T1* nor *T2* should appear in the resolvent. We call *T1* and *T2* *Complementary literals*. Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should be omitted from the resolvent.
   (c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

Prove that Marcus hates ceaser using resolution.


Prove: hate (Marcus, Caesar)
¬hate (Marcus, Caesar)
5
Marcus/x₂
3
¬Roman (Marcus) ∨ loyalto (Marcus, Caesar)
Marcus/x₁
¬Pompeian (Marcus) ∨ loyalto (Marcus, Caesar)
2
7
loyalto (Marcus, Caesar)
Macus/x₄, Caesar/y₁
1
¬man (Marcus) ∨ ¬ruler (Caesar) ∨ ¬tryassassinate (Marcus, Caesar)
¬ruler(Caesar) ∨ ¬tryassassinate (Marcus, Caesar)
4
¬tryassassinate (Marcus, Caesar)
8

Example:

John likes all kinds of food.

Apples are food.

Chicken is food.

Anything anyone eats and it is not killed is food.

Bill eats peanuts and is still alive.

Swe eats everything bill eats

(a) Convert all the above statements into predicate logic
(b) Show that John likes peanuts using back chaining
(c) Convert the statements into clause form
(d) Using Resolution show that "John likes peanuts"

Answer:

(a) Predicate Logic:

1. $\forall x: food(x) \rightarrow like(John)$
2. $Food(Apples)$
3. $Food(Chicken)$
4. $\forall\, x, \forall\, y: Eat(x, y) \wedge \neg Killed(x) \rightarrow Food(y)$
5. $Eats(Bill, Peanuts) \wedge Alive(Bill)$
6. $\forall\, x: Eats(Bill, x) \;\rightarrow\; Eats(Swe, x)$

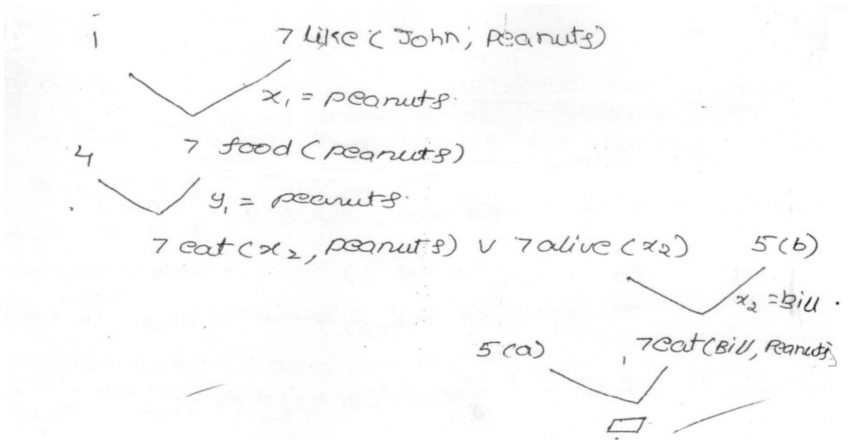
(b) Backward Chaining Proof:

$$Like\ (John, Peanuts)$$
$$\uparrow$$
$$Food(Peanuts)$$
$$\uparrow$$
$$Eat(Bill, Peanuts) \wedge Alive(Bill)$$
$$\uparrow$$
$$Nil$$

(c) Clause Form:

1. $\neg\ Food(x) \vee Like(John, x)$
2. $Food(Apples)$
3. $Food(Chicken)$
4. $\neg\ (Eat(x, y) \wedge \neg\ Killed(x)) \vee Food(y)\ \ => (\neg\ Eat(x, y) \vee Killed(x))\ \vee\ Food(y)$
5. $Eats(Bill, Peanuts)$
6. $Alive(Bill)$
7. $\neg\ (Eats(Bill, x))\ \vee Eats(Swe, x)$

(d) Resolution Proof:



Answering Questions

We can also use the proof procedure to answer questions such as "who tried to assassinate Caesar" by proving:

– Tryassassinate(y,Caesar).

- Once the proof is complete we need to find out what was substitution was made for y.

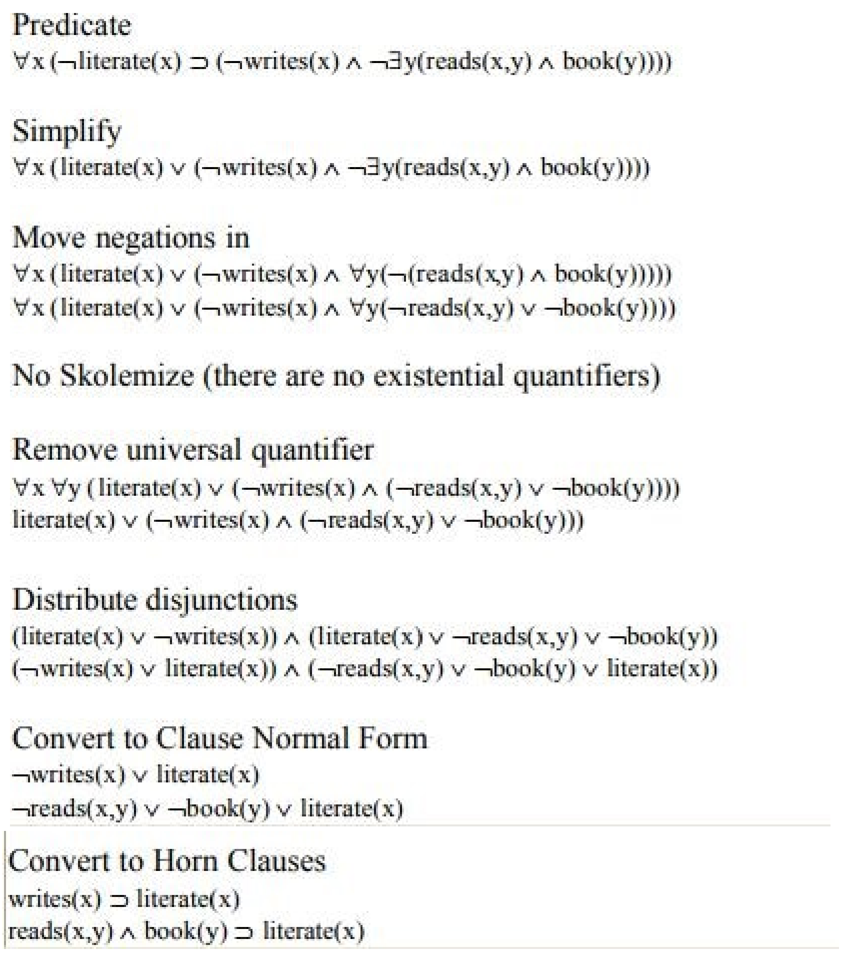
We show how resolution can be used to answer fill-in-the-blank questions, such as "When did Marcus die?" or "Who tried to assassinate a ruler?" Answering these questions involves finding a known statement that matches the terms given in the question and then responding with another piece of the same statement that fills the slot demanded by the question.

From Clause Form to Horn Clauses

The operation is to convert Clause form to Horn Clauses. This operation is not always possible. Horn clauses are clauses in normal form that have one or zero positive literals. The conversion from a clause in normal form with one or zero positive literals to a Horn clause is done by using the implication property.

$$\neg P \vee Q \;\; \textit{Rewrites to } P \rightarrow Q$$

Example:



Predicate

$\forall x\, (\neg \text{literate}(x) \supset (\neg \text{writes}(x) \wedge \neg \exists y(\text{reads}(x,y) \wedge \text{book}(y))))$

Simplify

$\forall x\, (\text{literate}(x) \vee (\neg \text{writes}(x) \wedge \neg \exists y(\text{reads}(x,y) \wedge \text{book}(y))))$

Move negations in

$\forall x\, (\text{literate}(x) \vee (\neg \text{writes}(x) \wedge \forall y(\neg(\text{reads}(x,y) \wedge \text{book}(y)))))$

$\forall x\, (\text{literate}(x) \vee (\neg \text{writes}(x) \wedge \forall y(\neg \text{reads}(x,y) \vee \neg \text{book}(y))))$

No Skolemize (there are no existential quantifiers)

Remove universal quantifier

$\forall x\, \forall y\, (\text{literate}(x) \vee (\neg \text{writes}(x) \wedge (\neg \text{reads}(x,y) \vee \neg \text{book}(y))))$

$\text{literate}(x) \vee (\neg \text{writes}(x) \wedge (\neg \text{reads}(x,y) \vee \neg \text{book}(y)))$

Distribute disjunctions

$(\text{literate}(x) \vee \neg \text{writes}(x)) \wedge (\text{literate}(x) \vee \neg \text{reads}(x,y) \vee \neg \text{book}(y))$

$(\neg \text{writes}(x) \vee \text{literate}(x)) \wedge (\neg \text{reads}(x,y) \vee \neg \text{book}(y) \vee \text{literate}(x))$

Convert to Clause Normal Form

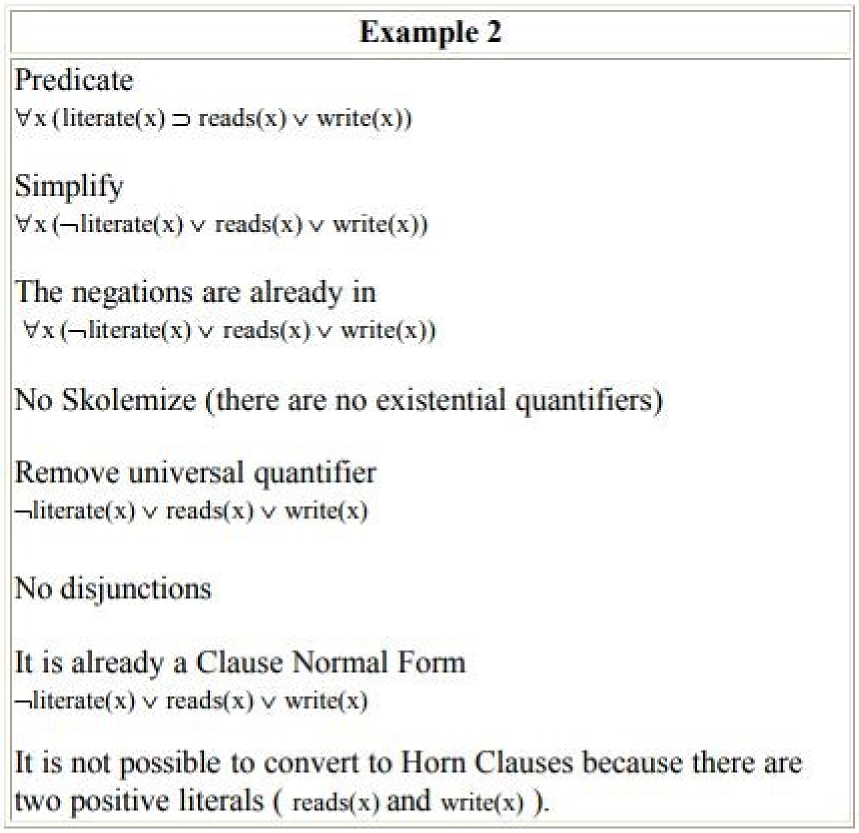
$\neg \text{writes}(x) \vee \text{literate}(x)$

$\neg \text{reads}(x,y) \vee \neg \text{book}(y) \vee \text{literate}(x)$

Convert to Horn Clauses

$\text{writes}(x) \supset \text{literate}(x)$

$\text{reads}(x,y) \wedge \text{book}(y) \supset \text{literate}(x)$



Example 2

Predicate
∀x (literate(x) ⊃ reads(x) ∨ write(x))

Simplify
∀x (¬literate(x) ∨ reads(x) ∨ write(x))

The negations are already in
∀x (¬literate(x) ∨ reads(x) ∨ write(x))

No Skolemize (there are no existential quantifiers)

Remove universal quantifier
¬literate(x) ∨ reads(x) ∨ write(x)

No disjunctions

It is already a Clause Normal Form
¬literate(x) ∨ reads(x) ∨ write(x)

It is not possible to convert to Horn Clauses because there are two positive literals (reads(x) and write(x)).

4. Knowledge Representation Issues

Introduction:

Knowledge plays an important role in AI systems. The kinds of knowledge might need to be represented in AI systems:

- Objects: Facts about objects in our world domain. e.g. Guitars have strings, trumpets are brass instruments.
- Events: Actions that occur in our world. e.g. Steve Vai played the guitar in Frank Zappa's Band.
- Performance: A behavior like playing the guitar involves knowledge about how to do things.
- Meta-knowledge: Knowledge about what we know. e.g. Bobrow's Robot who plan's a trip. It knows that it can read street signs along the way to find out where it is.

Representations & Mappings:

In order to solve complex problems in AI we need:

- A large amount of knowledge
- Some mechanisms for manipulating that knowledge to create solutions to new problem.

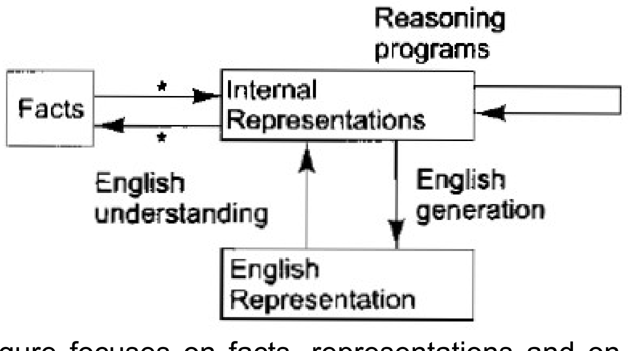
A variety of ways of representing knowledge have been exploited in AI problems. In this regard we deal with two different kinds of entities:

- Facts: truths about the real world and these are the things we want to represent.
- Representation of the facts in some chosen formalism. These are the things which we will actually be able to manipulate.

One way to think of structuring these entities is as two levels:

- Knowledge Level, at which facts are described.
- Symbol Level, at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.

Mappings between Facts and Representations:



The model in the above figure focuses on facts, representations and on the 2-way mappings that must exist between them. These links are called *Representation Mappings*.

- Forward Representation mappings maps from Facts to Representations.
- Backward Representation mappings maps from Representations to Facts.

English or natural language is an obvious way of representing and handling facts. Regardless of representation for facts, we use in program, we need to be concerned with English

Representation of those facts in order to facilitate getting information into or out of the system.

Mapping functions from English Sentences to Representations: Mathematical logic as representational formalism.

Example:

"Spot is a dog"

The fact represented by that English sentence can also be represented in logic as:

dog(Spot)

Suppose that we also have a logical representation of the fact that

"All dogs have tails" → $\forall x: dog(x) \rightarrow hastail(x)$

Then, using the deductive mechanisms of logic, we may generate the new representation object: astail(Spot)

Using an appropriate backward mapping function the English sentence "Spot has a tail" can be generated.

Fact-Representation mapping may not be one-to-one but rather are many-to-many which are a characteristic of English Representation. Good Representation can make a reasoning program simple.

Example:

"All dogs have tails"

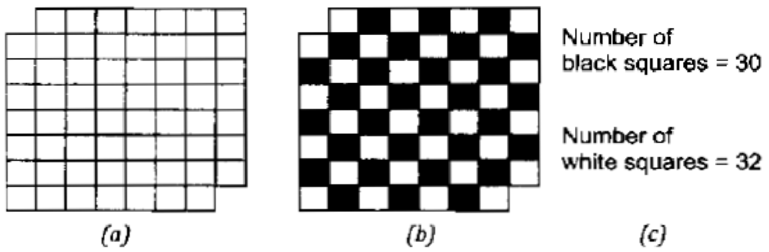
"Every dog has a tail"

From the two statements we can conclude that "Each dog has a tail." From the statement 1, we conclude that "Each dog has more than one tail."

When we try to convert English sentence into some other represent such as logical propositions, we first decode what facts the sentences represent and then convert those facts into the new representations. When an AI program manipulates the internal representation of facts these new representations should also be interpretable as new representations of facts.

Mutilated Checkerboard Problem:

Problem: In a normal chess board the opposite corner squares have been eliminated. The given task is to cover all the squares on the remaining board by dominoes so that each domino covers two squares. No overlapping of dominoes is allowed, can it be done? Consider three data structures

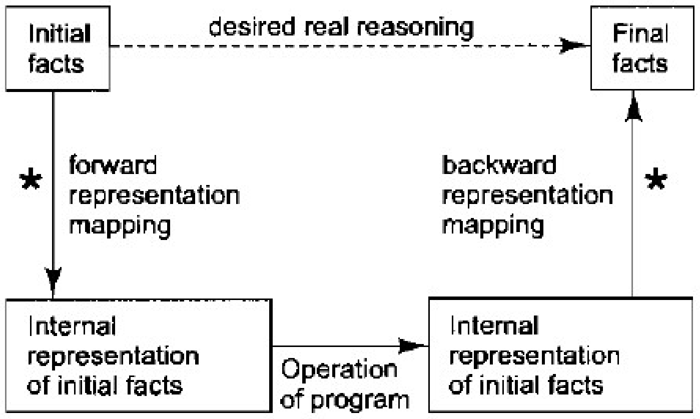


Three Representations of a Mutilated Checker board

The first representation does not directly suggest the answer to the problem. The second may suggest. The third representation does, when combined with the single additional facts that each domino must cover exactly one white square and one black square.

The puzzle is impossible to complete. A domino placed on the chessboard will always cover one white square and one black square. Therefore a collection of dominoes placed on the board will cover an equal numbers of squares of each color. If the two white corners are removed from the board then 30 white squares and 32 black squares remain to be covered by dominoes, so this is impossible. If the two black corners are removed instead, then 32 white squares and 30 black squares remain, so it is again impossible.

The solution is number of squares must be equal for positive solution.


Initial facts
desired real reasoning
Final facts
*
forward representation mapping
backward representation mapping
*
Internal representation of initial facts
Operation of program
Internal representation of initial facts

Representation of Facts

In the above figure, the dotted line across the top represents the abstract reasoning process that a program is intended to model. The solid line across the bottom represents the concrete reasoning process that a particular program performs. This program successfully models the abstract process to the extent that, when the backward representation mapping is applied to the program's output, the appropriate final facts are actually generated.

If no good mapping can be defined for a problem, then no matter how good the program to solve the problem is, it will not be able to produce answers that correspond to real answers to the problem.

Using Knowledge

Let us consider to what applications and how knowledge may be used.

- ❑ Learning: acquiring knowledge. This is more than simply adding new facts to a knowledge base. New data may have to be classified prior to storage for easy retrieval, etc.. Interaction and inference with existing facts to avoid redundancy and replication in the knowledge and also so that facts can be updated.
- ❑ Retrieval: The representation scheme used can have a critical effect on the efficiency of the method. Humans are very good at it. Many AI methods have tried to model human.
- ❑ Reasoning: Infer facts from existing data.

If a system on only knows:

- Miles Davis is a Jazz Musician.
- All Jazz Musicians can play their instruments well.

If things like *Is Miles Davis a Jazz Musician?* or *Can Jazz Musicians play their instruments well?* are asked then the answer is readily obtained from the data structures and procedures.

However a question like "*Can Miles Davis play his instrument well?*" requires reasoning. The above are all related. For example, it is fairly obvious that learning and reasoning involve retrieval etc.

Approaches to Knowledge Representation

A good Knowledge representation enables fast and accurate access to Knowledge and understanding of content. *The goal of Knowledge Representation (KR) is to facilitate conclusions from knowledge.*

The following properties should be possessed by a knowledge representation system.

- Representational Adequacy: the ability to represent all kinds of knowledge that are needed in that domain;

- Inferential Adequacy: the ability to manipulate the knowledge represented to produce new knowledge corresponding to that inferred from the original;

- Inferential Efficiency: the ability to incorporate into the knowledge structure additional information that can be used to focus the attention of the inference mechanisms in the most promising directions.

- Acquisitional Efficiency: the ability to acquire new information easily. The simplest case involves direct insertion, by a person of new knowledge into the database. Ideally, the program itself would be able to control knowledge acquisition.

No single system that optimizes all of the capabilities for all kinds of knowledge has yet been found. As a result, multiple techniques for knowledge representation exist.

Knowledge Representation Schemes

There are four types of Knowledge Representation:

- Relational Knowledge:
  - provides a framework to compare two objects based on equivalent attributes
  - any instance in which two different objects are compared is a relational type of knowledge
- Inheritable Knowledge:
  - is obtained from associated objects
  - it prescribes a structure in which new objects are created which may inherit all or a subset of attributes from existing objects.
- Inferential Knowledge
  - is inferred from objects through relations among objects
  - Example: a word alone is simple syntax, but with the help of other words in phrase the reader may infer more from a word; this inference within linguistic is called semantics.
- Declarative Knowledge
  - a statement in which knowledge is specified, but the use to which that knowledge is to be put is not given.
  - Example: laws, people's name; there are facts which can stand alone, not dependent on other knowledge
- Procedural Knowledge
  - a representation in which the control information, to use the knowledge is embedded in the knowledge itself.
  - Example: computer programs, directions and recipes; these indicate specific use or implementation

Simple relational knowledge

The simplest way of storing facts is to use a relational method where each fact about a set of objects is set out systematically in columns. This representation gives little opportunity for inference, but it can be used as the knowledge basis for inference engines.

- Simple way to store facts.
- Each fact about a set of objects is set out systematically in columns.
- Little opportunity for inference.
- Knowledge basis for inference engines.

Table - Simple Relational Knowledge

Player	Height	Weight	Bats - Throws
Aaron	6-0	180	Right - Right
Mays	5-10	170	Right - Right
Ruth	6-2	215	Left - Left
Williams	6-3	205	Left - Right

Given the facts it is not possible to answer simple question such as "Who is the heaviest player?" but if a procedure for finding heaviest player is provided, then these facts will enable that procedure to compute an answer. We can ask things like who "bats - left" and "throws - right".

Inheritable Knowledge

Here the knowledge elements inherit attributes from their parents. The knowledge is embodied in the design hierarchies found in the functional, physical and process domains. Within the hierarchy, elements inherit attributes from their parents, but in many cases not all attributes of the parent elements be prescribed to the child elements.

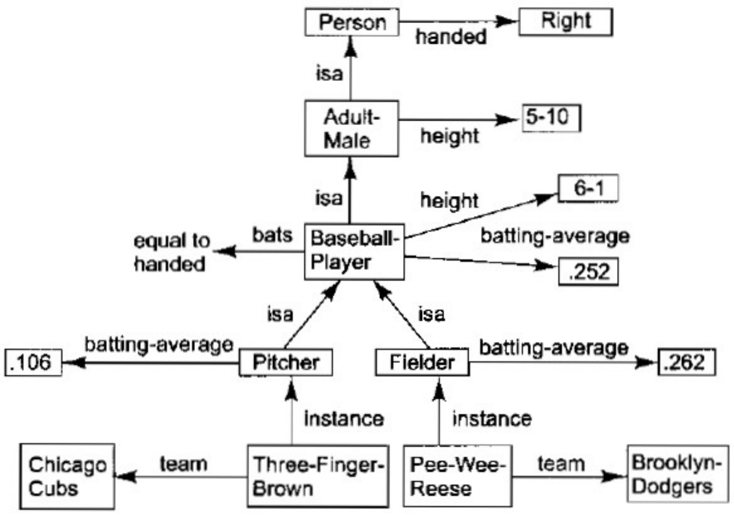
The inheritance is a powerful form of inference, but not adequate. The basic KR needs to be augmented with inference mechanism.

The KR in hierarchical structure, shown below, is called “semantic network” or a collection of “frames” or “slot-and-filler structure”. The structure shows property inheritance and way for insertion of additional knowledge.

Property inheritance: The objects or elements of specific classes inherit attributes and values from more general classes. The classes are organized in a generalized hierarchy.

Baseball Knowledge

- isa: show class inclusion
- i



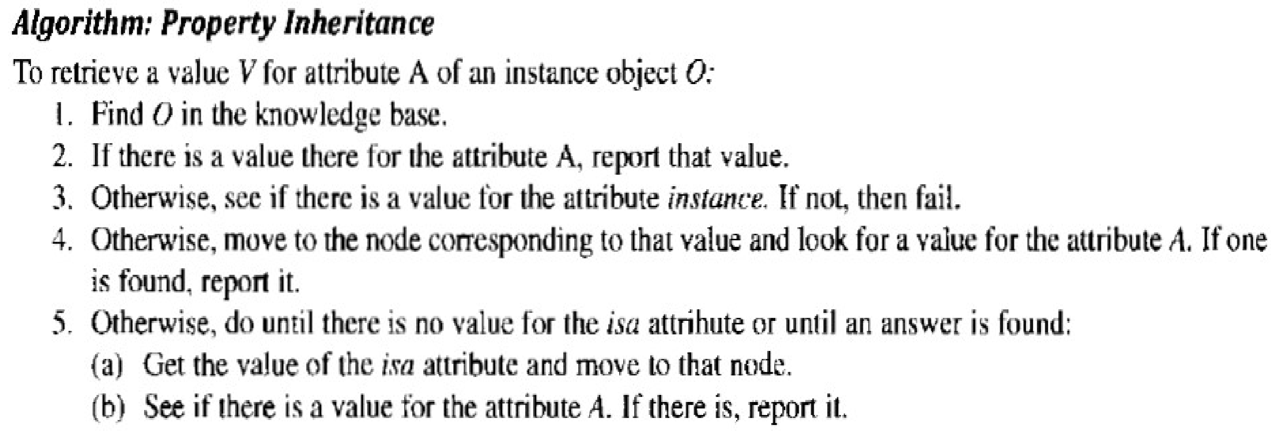
Inheritable Knowledge

- The directed arrows represent attributes (isa, instance, team) originates at object being described and terminates at object or its value.
- The box nodes represent objects and values of the attributes.

Viewing a node as a frame

Example: Baseball-player

Isa:	Adult-Male
Bats:	EQUAL handed
Height:	6-1
Batting-average:	0.252



Algorithm: Property Inheritance

To retrieve a value *V* for attribute A of an instance object *O:*

1. Find *O* in the knowledge base.
2. If there is a value there for the attribute A, report that value.
3. Otherwise, see if there is a value for the attribute *instance.* If not, then fail.
4. Otherwise, move to the node corresponding to that value and look for a value for the attribute *A*. If one is found, report it.
5. Otherwise, do until there is no value for the *isa* attrihute or until an answer is found:
   (a) Get the value of the *isa* attribute and move to that node.
   (b) See if there is a value for the attribute *A*. If there is, report it.

This algorithm is simple. It describes the basic mechanism of inheritance. It does not say what to do if there is more than one value of the instance or "isa" attribute.

This can be applied to the example of knowledge base, to derive answers to the following queries:

- team (Pee-Wee-Reese) = Brooklyn-Dodger
- batting-average (Three-Finger-Brown) = 0.106
- height (Pee-Wee-Reese) = 6.1
- bats (Three-Finger-Brown) = right

Inferential Knowledge:

This knowledge generates new information from the given information. This new information does not require further data gathering from source, but does require analysis of the given information to generate new knowledge. In this, we represent knowledge as formal logic.

Example:

- given a set of relations and values, one may infer other values or relations
- a predicate logic (a mathematical deduction) is used to infer from a set of attributes.
- inference through predicate logic uses a set of logical operations to relate individual data.
- the symbols used for the logic operations are:

" → " (implication), " ¬ " (not), " V " (or), " Λ " (and),
" ∀ " (for all), " ∃ " (there exists).

Examples of predicate logic statements :

1. *"Wonder"* is a name of a dog :	**dog (wonder)**
2. All dogs belong to the class of animals :	∀ **x : dog (x) → animal(x)**
3. All animals either live on land or in water :	∀ **x : animal(x) → live (x, land) V live (x, water)**

From these three statements we can infer that :

" *Wonder* lives either on land or on water."

Note : If more information is made available about these objects and their relations, then more knowledge can be inferred.

Procedural Knowledge

Procedural knowledge can be represented in programs in many ways. The most common way is simply as for doing something. The machine uses the knowledge when it executes the code to perform a task. Procedural Knowledge is the knowledge encoded in some procedures.

Unfortunately, this way of representing procedural knowledge gets low scores with respect to the properties of inferential adequacy (because it is very difficult to write a program that can reason about another program's behavior) and acquisitional efficiency (because the process of updating and debugging large pieces of code becomes unwieldy).

The most commonly used technique for representing procedural knowledge in AI programs is the use of production rules.

```
If:      ninth inning, and
         score is close, and
         less than 2 outs, and
         first base is vacant, and
         batter is better hitter than next batter,
Then:    walk the batter.
```

Procedural Knowledge as Rules

Production rules, particularly ones that are augmented with information on how they are to be used, are more procedural than are the other representation methods. But making a clean distinction between declarative and procedural knowledge is difficult. The important difference is in how the knowledge is used by the procedures that manipulate it.

Heuristic or Domain Specific knowledge can be represented using Procedural Knowledge.

Issues in Knowledge Representation

Below are listed issues that should be raised when using knowledge representation techniques:

- ◇ Important Attributes :
  Any attribute of objects so basic that they occur in almost every problem domain ?
- ◇ Relationship among attributes:
  Any important relationship that exists among object attributes ?
- ◇ Choosing Granularity :
  At what level of detail should the knowledge be represented ?
- ◇ Set of objects :
  How sets of objects be represented ?
- ◇ Finding Right structure :
  Given a large amount of knowledge stored, how can relevant parts be accessed ?

Important Attributes : *(Ref. Example - Fig. Inheritable KR)*

There are attributes that are of general significance.

There are two attributes **"instance"** and **"isa"**, that are of general importance. These attributes are important because they support *property inheritance*.

The attributes are called a variety of things in AI systems, but the names do not matter. What does matter is that they represent class membership and class inclusion and that class inclusion is transitive. The predicates are used in Logic Based Systems.

Relationship among Attributes

- ☐ The attributes to describe objects are themselves entities that we represent.
- ☐ The relationship between the attributes of an object, independent of specific knowledge they encode, may hold properties like:

  Inverses, existence in an isa hierachy, techniques for reasoning about values and single valued attributes.

Inverses :

This is about *consistency check*, while a value is added to one attribute. The entities are related to each other in many different ways. The figure shows attributes *(isa, instance, and team)*, each with a directed arrow, originating at the object being described and terminating either at the object or its value.

There are two ways of realizing this:

- first, represent two relationships in a *single representation*; e.g., a logical representation, ***team(Pee-Wee-Reese, Brooklyn–Dodgers)***, that can be interpreted as a statement about Pee-Wee-Reese or Brooklyn–Dodger.
- second, use attributes that focus on a *single entity but use them in pairs*, one the inverse of the other; for e.g., one, ***team = Brooklyn–Dodgers*** , and the other, ***team = Pee-Wee-Reese, ...***

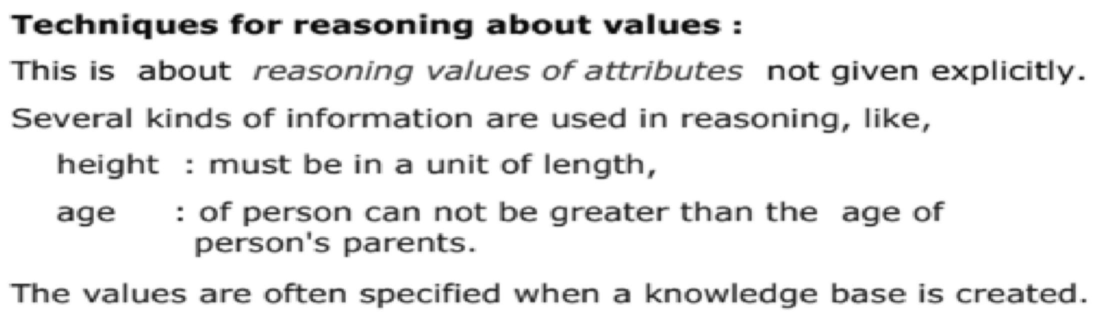
The second way can be realized using semantic net and frame based systems. This Inverses is used in Knowledge Acquisition Tools.

Existence in an "isa" hierarchy :

This is about *generalization-specialization*, like, classes of objects and specialized subsets of those classes. There are attributes and specialization of attributes.

Example: the attribute *"height"* is a specialization of general attribute *"physical-size"* which is, in turn, a specialization of *"physical-attribute"*. These generalization-specialization relationships for attributes are important because they support inheritance.

This also provides information about constraints on the values that the attribute can have and mechanisms for computing those values.



Techniques for reasoning about values :

This is about *reasoning values of attributes* not given explicitly.

Several kinds of information are used in reasoning, like,

height : must be in a unit of length,

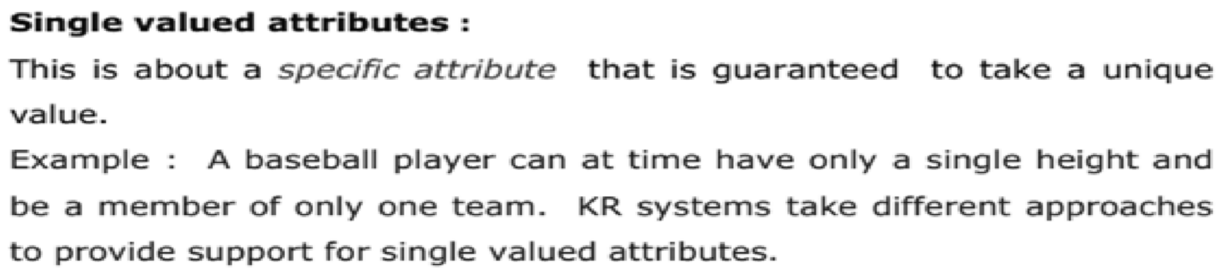
age : of person can not be greater than the age of person's parents.

The values are often specified when a knowledge base is created.

Several kinds of information can play a role in this reasoning, including:

Information about the type of the value.

- Constraints on the value often stated in terms of related entities.
- Rules for computing the value when it is needed. (Example: of such a rule in for bats attribute). These rules are called backward rules. Such rules have also been called ifneeded rules.
- Rules that describe actions that should be taken if a value ever becomes known. These rules are called forward rules, or sometimes if-added rules.

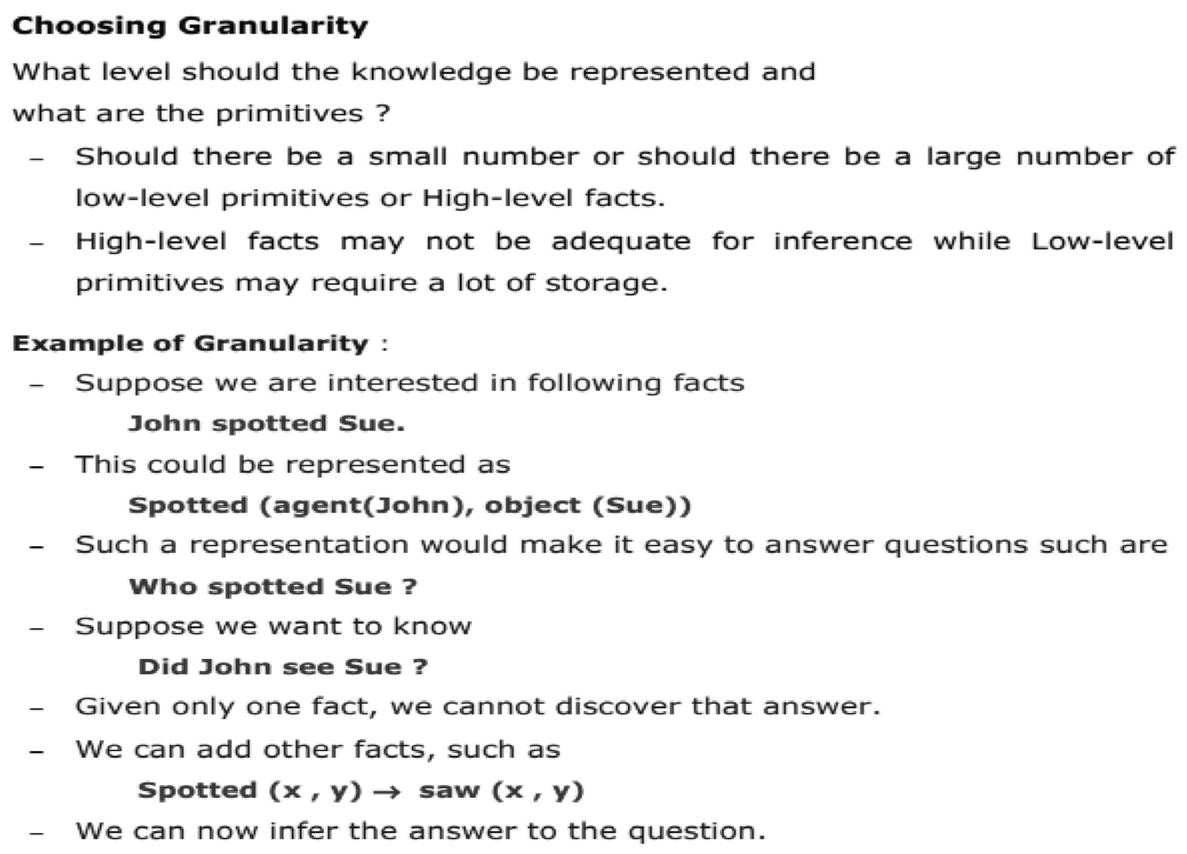


Single valued attributes :

This is about a *specific attribute* that is guaranteed to take a unique value.

Example : A baseball player can at time have only a single height and be a member of only one team. KR systems take different approaches to provide support for single valued attributes.

- Introduce an explicit notation for temporal interval. If two different values are ever asserted for the same temporal interval, signal a contradiction automatically.
- Assume that the only temporal interval that is of interest is now. So if a new value is asserted, replace the old value.
- Provide no explicit support. Logic-based systems are in this category. But in these systems, knowledge base builders can add axioms that state that if an attribute has one value then it is known not to have all other values.



Choosing Granularity

What level should the knowledge be represented and
what are the primitives ?

- Should there be a small number or should there be a large number of low-level primitives or High-level facts.
- High-level facts may not be adequate for inference while Low-level primitives may require a lot of storage.

Example of Granularity :

- Suppose we are interested in following facts
  John spotted Sue.
- This could be represented as
  Spotted (agent(John), object (Sue))
- Such a representation would make it easy to answer questions such are
  Who spotted Sue ?
- Suppose we want to know
  Did John see Sue ?
- Given only one fact, we cannot discover that answer.
- We can add other facts, such as
  Spotted (x , y) → saw (x , y)
- We can now infer the answer to the question.

Choosing the Granularity of Representation Primitives are fundamental concepts such as holding, seeing, playing and as English is a very rich language with over half a million words it is clear we will find difficulty in deciding upon which words to choose as our primitives in a series of situations. Separate levels of understanding require different levels of primitives and these need many rules to link together similar primitives.

Set of Objects

Certain properties of objects that are true as member of a set but not as individual;

Example : Consider the assertion made in the sentences

"there are more *sheep* than *people* in Australia", and

"*English* speakers can be found all over the world."

To describe these facts, the only way is to attach assertion to the sets representing people, sheep, and English.

The reason to represent sets of objects is :

If a property is true for all or most elements of a set,
then it is more efficient to associate it once with the set
rather than to associate it explicitly with every elements of the set .

This is done in different ways :

- in logical representation through the use of *universal quantifier*, and
- in hierarchical structure where node represent sets, the *inheritance propagate* set level assertion down to individual.

Example: assert **large (elephant);**

Remember to make clear distinction between,

- whether we are asserting some property of the set itself, means, **the set of elephants is large**, or
- asserting some property that holds for individual elements of the set , means, **any thing that is an elephant is large**.

There are three ways in which sets may be represented :

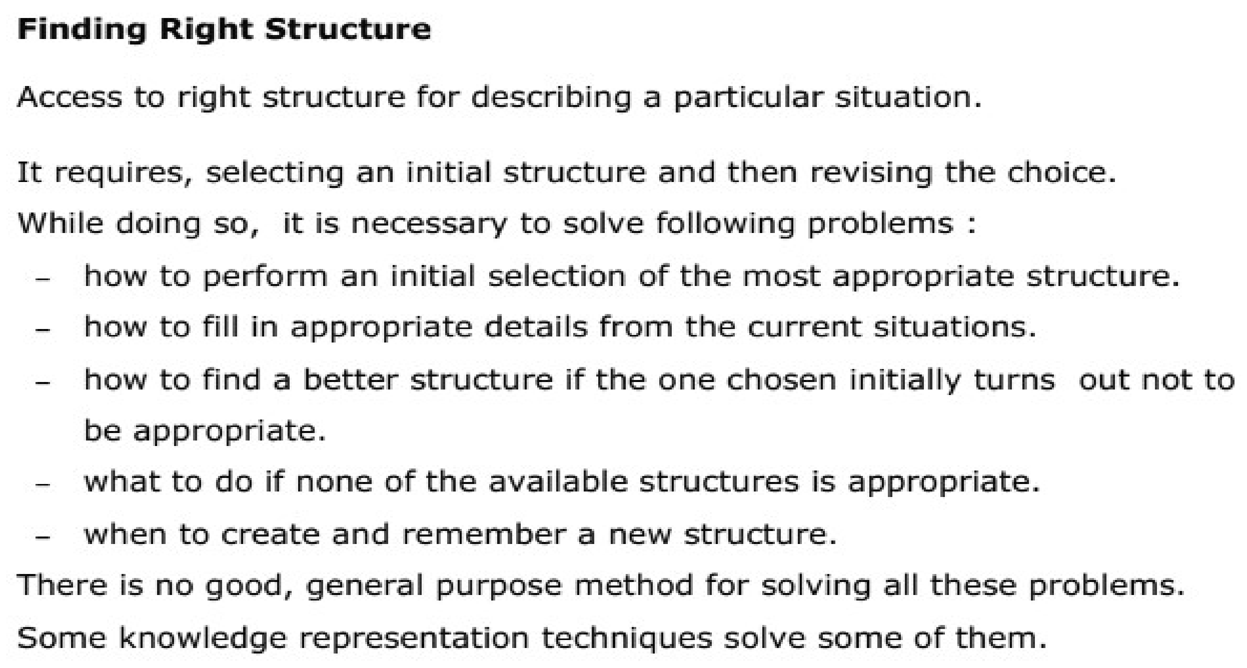
(a) Name, as in the example – Ref Fig. Inheritable KR, the node - Baseball-Player and the predicates as Ball and Batter in logical representation.

(b) Extensional definition is to list the numbers, and

(c) In tensional definition is to provide a rule, that returns true or false depending on whether the object is in the set or not.

$\{x: sun - planet(x) \wedge human - inhabited(x)\}$ – $Intensional\ Definition$

$Extensional\ Definition - Set\ of\ our\ sun\ planets\ on\ which\ people\ live\ is\ Earth$



Finding Right Structure

Access to right structure for describing a particular situation.

It requires, selecting an initial structure and then revising the choice.
While doing so, it is necessary to solve following problems :

- how to perform an initial selection of the most appropriate structure.
- how to fill in appropriate details from the current situations.
- how to find a better structure if the one chosen initially turns out not to be appropriate.
- what to do if none of the available structures is appropriate.
- when to create and remember a new structure.

There is no good, general purpose method for solving all these problems.
Some knowledge representation techniques solve some of them.

6. Representing Knowledge using Rules

Procedural versus Declaration Knowledge

Declarative Knowledge	**Procedural Knowledge**
Factual information stored in memory and known to be static in nature.	the knowledge of how to perform, or how to operate
knowledge of facts or concepts	a skill or action that you are capable of performing
knowledge about that something true or false	Knowledge about how to do something to reach a particular objective or goal
knowledge is specified but how to use to which that knowledge is to be put is not given	control information i.e., necessary to use the knowledge is considered to be embedded in the knowledge itself
E.g.: concepts, facts, propositions, assertions, semantic nets …	E.g.: procedures, rules, strategies, agendas, models
It is explicit knowledge (describing)	It is tacit knowledge (doing)

The declarative representation is one in which the knowledge is specified but how to use to which that knowledge is to be put is not given.

- Declarative knowledge answers the question 'What do you know?'
- It is your understanding of things, ideas, or concepts.
- In other words, declarative knowledge can be thought of as the who, what, when, and where of information.
- Declarative knowledge is normally discussed using nouns, like the names of people, places, or things or dates that events occurred.

The procedural representation is one in which the control information i.e., necessary to use the knowledge is considered to be embedded in the knowledge itself.

- Procedural knowledge answers the question 'What can you do?'
- While declarative knowledge is demonstrated using nouns,
- Procedural knowledge relies on action words, or verbs.
- It is a person's ability to carry out actions to complete a task.

The real difference between declarative and procedural views of knowledge lies in which the control information presides.

Example:

1. $man(marcus)$
2. $man(ceaser)$
3. $\forall x: man(x) \rightarrow person(x)$
4. $person(cleopatra)$

The statements *1, 2 and 3 are procedural knowledge* and *4 is a declarative knowledge*.

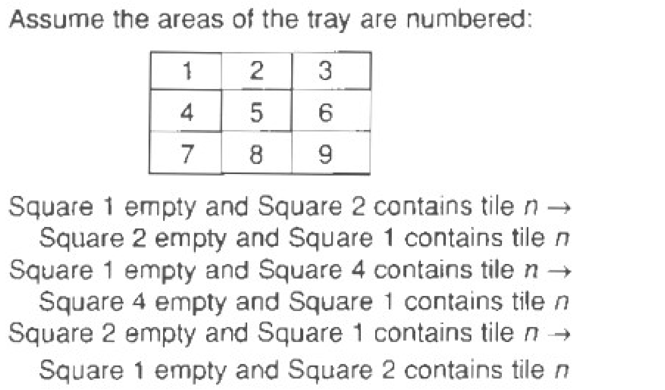
Forward & Backward Reasoning

The object of a search procedure is to discover a path through a problem space from an initial configuration to a goal state. There are actually two directions in which such a search could proceed:

- Forward Reasoning,
  - from the start states
  - LHS rule must match with initial state
  - Eg: A → B, B→C => A→C
- Backward Reasoning,
  - from the goal states
  - RHS rules must match with goal state
  - Eg: 8-Puzzle Problem

In both the cases, the control strategy is it must cause motion and systematic. The production system model of the search process provides an easy way of viewing forward and backward reasoning as symmetric processes.

Consider the problem of solving a particular instance of the 8-puzzle problem. The rules to be used for solving the puzzle can be written as:



Assume the areas of the tray are numbered:

1	2	3
4	5	6
7	8	9

Square 1 empty and Square 2 contains tile *n* →
Square 2 empty and Square 1 contains tile *n*
Square 1 empty and Square 4 contains tile *n* →
Square 4 empty and Square 1 contains tile *n*
Square 2 empty and Square 1 contains tile *n* →
Square 1 empty and Square 2 contains tile *n*

A Sample of the Rules for Solving the 8-Puzzle

Reasoning Forward from Initial State:

- Begin building a tree of move sequences that might be solved with initial configuration at root of the tree.
- Generate the next level of the tree by finding all the rules whose left sides match the root node and using their right sides to create the new configurations.
- Generate the next level by taking each node generated at the previous level and applying to it all of the rules whose left sides match it.
- Continue until a configuration that matches the goal state is generated.

Reasoning Backward from Goal State:

- Begin building a tree of move sequences that might be solved with goal configuration at root of the tree.

- Generate the next level of the tree by finding all the rules whose right sides match the root node. These are all the rules that, if only we could apply them, would generate the state we want. Use the left sides of the rules to generate the nodes at this second level of the tree.
- Generate the next level of the tree by taking each node at the previous level and finding all the rules whose right sides match it. Then use the corresponding left sides to generate the new nodes.
- Continue until a node that matches the initial state is generated.
- This method of reasoning backward from the desired final state is often called goaldirected reasoning.

To reason forward, the left sides (preconditions) are matched against the current state and the right sides (results) are used to generate new nodes until the goal is reached. To reason backward, the right sides are matched against the current node and the left sides are used to generate new nodes representing new goal states to be achieved.

The following 4 factors influence whether it is better to reason Forward or Backward:

1. Are there more possible start states or goal states? We would like to move from the smaller set of states to the larger (and thus easier to find) set of states.
2. In which direction branching factor (i.e, average number of nodes that can be reached directly from a single node) is greater? We would like to proceed in the direction with lower branching factor.
3. Will the program be used to justify its reasoning process to a user? If so, it is important to proceed in the direction that corresponds more closely with the way the user will think.
4. What kind of event is going to trigger a problem-solving episode? If it is arrival of a new fact, forward reasoning makes sense. If it is a query to which a response is desired, backward reasoning is more natural.

Backward-Chaining Rule Systems

- Backward-chaining rule systems are good for goal-directed problem solving.
- For example, a query system would probably use backward chaining to reason about and answer user questions.
- Unification tries to find a set of bindings for variables to equate a (sub) goal with the head of some rule.
- Medical expert system, diagnostic problems

Forward-Chaining Rule Systems

- Instead of being directed by goals, we sometimes want to be directed by incoming data.
- For example, suppose you sense searing heat near your hand. You are likely to jerk your hand away.
- Rules that match dump their right-hand side assertions into the state and the process repeats.
- Matching is typically more complex for forward-chaining systems than backward ones. ➢ Synthesis systems – Design/Configuration

Example of Typical Forward Chaining

Rules

1) If hot and smoky then ADD fire
2) If alarm_beeps then ADD smoky
3) If fire then ADD switchon_sprinkles

Facts

1) alarm_beeps (given)
2) hot (given)

………

(3) smoky (from F1 by R2)
(4) fire (from F2, F4 by R1)
(5) switch_on_sprinklers (from F2 by R3)

Example of Typical Backward Chaining

Goal: Should I switch on sprinklers?

Combining Forward and Backward Reasoning

Sometimes certain aspects of a problem are best handled via forward chaining and other aspects by backward chaining. Consider a forward-chaining medical diagnosis program. It might accept twenty or so facts about a patient's condition then forward chain on those concepts to try to deduce the nature and/or cause of the disease.

Now suppose that at some point, the left side of a rule was nearly satisfied – nine out of ten of its preconditions were met. It might be efficient to apply backward reasoning to satisfy the tenth precondition in a directed manner, rather than wait for forward chaining to supply the fact by accident.

Whether it is possible to use the same rules for both forward and backward reasoning also depends on the form of the rules themselves. If both left sides and right sides contain pure assertions, then forward chaining can match assertions on the left side of a rule and add to the state description the assertions on the right side. But if arbitrary procedures are allowed as the right sides of rules then the rules will not be reversible.

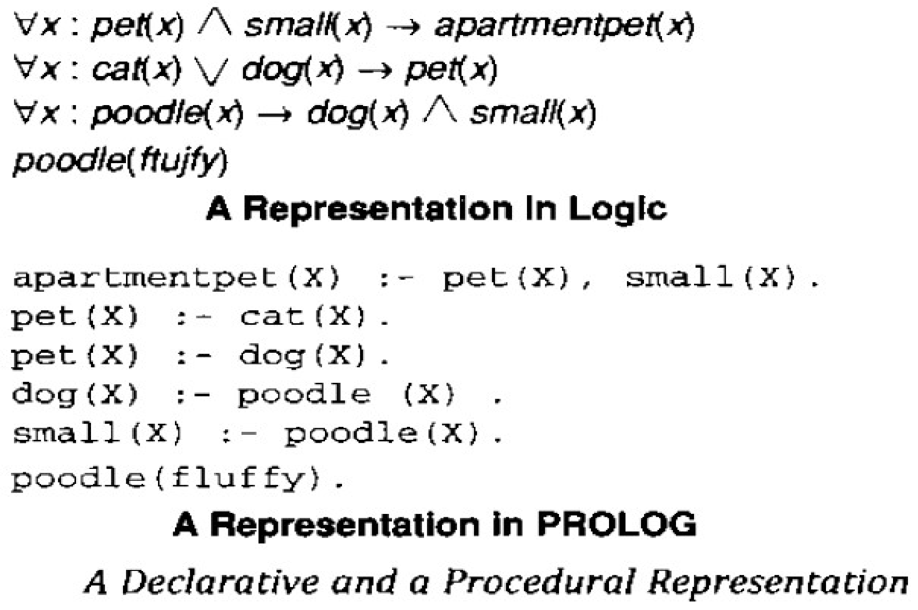
Logic Programming

- Logic Programming is a programming language paradigm in which logical assertions are viewed as programs.
- There are several logic programming systems in use today, the most popular of which is PROLOG.
- A PROLOG program is described as a series of logical assertions, each of which is a Horn clause.
- A Horn clause is a clause that has at most one positive literal. Thus p, p q, p → q are all Horn clauses.

Programs written in pure PROLOG are composed only of Horn Clauses.
Syntactic Difference between the logic and the PROLOG representations, including:

- *In logic, variables are explicitly quantified. In PROLOG, quantification is provided implicitly by the way the variables are interpreted.*
  - *The distinction between variables and constants is made in PROLOG by having all variables begin with uppercase letters and all constants begin with lowercase letters.*
- *In logic, there are explicit symbols for and ( ) and or ( ). In PROLOG, there is an explicit symbol for and (,), but there is none for or.*
- *In logic, implications of the form "p implies q" as written as p→q. In PROLOG, the same implication is written "backward" as q: -p.*

Example:



$\forall x : pet(x) \wedge small(x) \rightarrow apartmentpet(x)$
$\forall x : cat(x) \vee dog(x) \rightarrow pet(x)$
$\forall x : poodle(x) \rightarrow dog(x) \wedge small(x)$
$poodle(ftujfy)$

A Representation In Logic

```
apartmentpet(X)  :-  pet(X),  small(X).
pet(X)  :-  cat(X).
pet(X)  :-  dog(X).
dog(X)  :-  poodle (X) .
small(X)  :-  poodle(X).
poodle(fluffy).
```

A Representation in PROLOG

A Declarative and a Procedural Representation

The first two of these differences arise naturally from the fact that PROLOG programs are actually sets of Horn Clauses that have been transformed as follows:

1. If the Horn Clause contains no negative literals (i.e., it contains a single literal which is positive), then leave it as it is.
2. Otherwise, return the Horn clause as an implication, combining all of the negative literals into the antecedent of the implication and leaving the single positive literal (if there is one) as the consequent.

This procedure causes a clause, which originally consisted of a disjunction of literals (all but one of which were negative), to be transformed to single implication whose antecedent is a conjunction of (what are now positive) literals.

$\forall x : \forall y : cat(x) \wedge fish(y) \rightarrow likes-to-eat(x,y)$

$\forall x : calico(x) \rightarrow cat(x)$

$\forall x : tuna(x) \rightarrow fish(x)$

$tuna(Charlie)$

$tuna(Herb)$

$calico(Puss)$

(a) Convert these wff's into Horn clauses.

(b) Convert the Horn clauses into a PROLOG program.

(c) Write a PROLOG query corresponding to the question, "What does Puss like to eat?" and show how it will be answered by your program.

(a) Horn clauses:

1. $\neg cat(x) \vee \neg fish(y) \vee likes\text{-}to\text{-}eat(x.y)$
2. $\neg calico(x) \vee cat(x)$
3. $\neg tuna(x) \vee fish(x)$
4. $tuna(Charlie)$
5. $tuna(Herb)$
6. $calico(Puss)$

(b) PROLOG program:

```
likestoeat(X,Y)  :- cat(X), fish(Y).
cat(X)  :- calico(X).
fish(X)  :- tuna(X).
tuna(charlie).
tuna(herb).
calico(puss).
```

(c) Query:

```
?- likestoeat(puss,X).
```

Answer: `charlie`

Matching

We described the process of using search to solve problems as the application of appropriate rules to individual problem states to generate new states to which the rules can then be applied and so forth until a solution is found.

How we extract from the entire collection of rules those that can be applied at a given point? To do so requires some kind of matching between the current state and the preconditions of the rules. How should this be done? The answer to this question can be critical to the success of a rule based system.

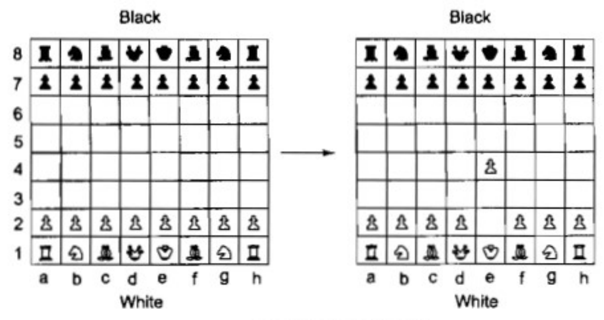
A more complex matching is required when the preconditions of rule specify required properties that are not stated explicitly in the description of the current state. In this case, a separate set of rules must be used to describe how some properties can be inferred from others. An even more complex matching process is required if rules should be applied and if their pre condition approximately match the current situation. This is often the case in situations involving physical descriptions of the world.

Indexing

One way to select applicable rules is to do a simple search though all the rules comparing each one's precondition to the current state and extracting all the one's that match. There are two problems with this simple solution:

i. The large number of rules will be necessary and scanning through all of them at every step would be inefficient.
ii. It's not always obvious whether a rule's preconditions are satisfied by a particular state.

Solution: Instead of searching through rules use the current state as an index into the rules and select the matching one's immediately.



One Legal Chess Move

White pawn at
Square(file e, rank 2)
AND
Square(file e, rank 3) → move pawn from
is empty Square(file e, rank 2)
AND to Square(file e, rank 4)
Square(file e, rank 4)
is empty

Another Way to Describe Chess Moves

Matching process is easy but at the price of complete lack of generality in the statement of the rules. Despite some limitations of this approach, Indexing in some form is very important in the efficient operation of rule based systems.

Matching with Variables

The problem of selecting applicable rules is made more difficult when preconditions are not stated as exact descriptions of particular situations but rather describe properties that the situations must have. It often turns out that discovering whether there is a match between a particular situation and the preconditions of a given rule must itself involve a significant search process.

Backward-chaining systems usually use depth-first backtracking to select individual rules, but forward-chaining systems generally employ sophisticated conflict resolution strategies to choose among the applicable rules.

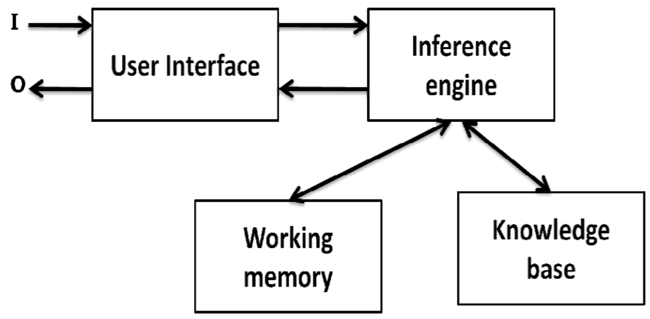
While it is possible to apply unification repeatedly over the cross product of preconditions and state description elements, it is more efficient to consider the many-many match problem, in which many rules are matched against many elements in the state description simultaneously. One efficient many-many match algorithm is RETE.

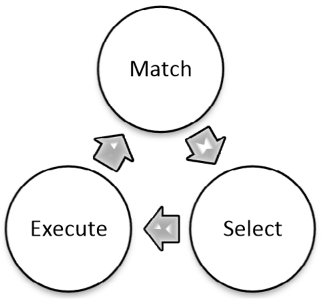
RETE Matching Algorithm

The matching consists of 3 parts

1. Rules & Productions
2. Working Memory
3. Inference Engine

The inference Engine is a cycle of production system which is match, select, execute.





INFERENCE ENGINE

The above cycle is repeated until no rules are put in the conflict set or until stopping condition is reached. In order to verify several conditions, it is a time consuming process. To eliminate the need to perform thousands of matches of cycles on effective matching algorithm is called RETE.

The Algorithm consists of two Steps.

1. Working memory changes need to be examined.
2. Grouping rules which share the same condition & linking them to their common terms.

RETE Algorithm is many-match algorithm (In which many rules are matched against many elements). RETE uses forward chaining systems which generally employee sophisticated conflict resolution strategies to choose among applicable rules. RETE gains efficiency from 3 major sources.

1. RETE maintains a network of rule condition and it uses changes in the state description to determine which new rules might apply. Full matching is only pursued for candidates that could be affected by incoming/outgoing data.
2. Structural Similarity in rules: RETE stores the rules so that they share structures in memory, set of conditions that appear in several rules are matched once for cycle.
3. Persistence of variable binding consistency. While all the individual preconditions of the rule might be met, there may be variable binding conflicts that prevent the rule from firing.

$$son(Mary, John)\ and\ son\ (Bill, Bob)$$
$$son(x, y)\ ^\wedge\ son(y, z) \rightarrow grandparents(x, z)$$

can be minimized. RETE remembers its previous calculations and is able to merge new binding information efficiently.

Approximate Matching:

Rules should be applied if their preconditions approximately match to the current situation

Eg: Speech understanding program

Rules: A description of a physical waveform to phones

Physical Signal: difference in the way individuals speak, result of background noise.

Conflict Resolution:

When several rules matched at once such a situation is called conflict resolution. There are 3 approaches to the problem of conflict resolution in production system.

1. Preference based on rule match:
    a. Physical order of rules in which they are presented to the system
    b. Priority is given to rules in the order in which they appear

2. Preference based on the objects match:
    a. Considers importance of objects that are matched
    b. Considers the position of the match able objects in terms of Long Term Memory (LTM) & Short Term Memory(STM)

       LTM: Stores a set of rules

       STM (Working Memory): Serves as storage area for the facts deduced by rules in long term memory
3. Preference based on the Action:
    a. One way to do is find all the rules temporarily and examine the results of each. Using a Heuristic Function that can evaluate each of the resulting states compare the merits of the result and then select the preferred one.

Search Control Knowledge:

- It is knowledge about which paths are most likely to lead quickly to a goal state
- Search Control Knowledge requires Meta Knowledge.
- It can take many forms. Knowledge about

  which states are more preferable to others.

  which rule to apply in a given situation

  the Order in which to pursue sub goals

  useful Sequences of rules to apply.

MODULE-2