To do computation, we must first convert from first-order logic to propositional logic.
In general, we do replacement, written as: {x/y} — replace x with y.
We must handle instantiation uniquely, however:

1. UNIVERSAL INSTANTIATION
   (∀ variable) (sentence) subst(variable/ground-term, sentence)
   ex: for constants {John, Richard}:
   ∀ x King(x) ∧ Greedy(x) $\implies$ Evil(x)
   → King(John) ∧ Greedy(John) $\implies$ Evil(John)
   → King(Richard) ∧ Greedy(Richard) $\implies$ Evil(Richard)
   → King(Father(John)) ∧ Greedy(Father(John)) $\implies$ Evil(Father(John))
   While this creates ground sentences, we can see that it could run infinitely
2. EXISTENTIAL INSTANTIATION
   (∃ variable) (sentence) subst(variable/new-constant, sentence)
   ex: for constants{C}
   ∃ x Crown(x) ∧ On-Head(x, John)
   → Crown(C) ∧ On-Head(C, John)

Without any functions, this term count is finite, but with functions, it could be infinite.
We can see that in these cases:

1. ∀ level 1 = ∀ level 2
2. SAT(∃ level 1) $\iff$ SAT(∃ level 2)

This has a major result:
Theorem (Herbrand 1980)
if a sentence is entailed by a first order logic knowledge base,
then it is entailed by a finite subset of that propositional knowledge base.
Application:
for n = 0 to ∞ do
create a propositional knowledge base by instantiating with depth-n terms
see if the sentence is entailed by this knowledge base
We have a problem, however:
this only terminates if the sentence is entailed!
$\implies$ inference is considered **semi-decidable** (Church/Turing 1936)

We can use resolution to prove implications as with propositional logic.
We must first define the idea of **unification**.
A **unifier** is a set of variable assignments that make two statements equivalent.
Examples:

1. Knows(John, x) & Knows(John, John), {x/John} → Knows(John, John)
2. Knows(John, x) & Knows(y, OJ), {x/OJ, y/John} → Knows(John, OJ)
3. Knows(John, x) & Knows(y, Mother(y)), {x/Mother(John), y/John} → Knows(John, Mother(John)
4. Knows(John, x) & Knows(x, OJ), NO SOLUTION

We can improve unification by indexing facts/predicate indexing
≡ we cache highly used facts — very useful for many predicate, few clause logics.
If we store by predicate and first argument, then we can lookup by either.
This formed a subsumption lattice where children are formed by substitution on a higher.
In this lattice, the highest common descendent of two nodes comes from their MGU.
Since this is $O(2^n)$, it requires a small n; thankfully, most AI problems meet this.

First order resolution goes as in the following example:

$$((\forall x Rich(x) \implies Unhappy(x)), Rich(Ken))$$
$$\{\neg\text{Rich(x)} \lor \text{Unhappy(x),Rich(Ken)}\}$$
$$\frac{\neg\text{Rich(x)} \lor \text{Unhappy}(x), \text{Rich(Ken)}}{\text{Unhappy(Ken)}} \text{ with } \{\text{x/Ken}\}$$

We use this to walk through the following proof:
Statements:

- Jack owns a dog.
- Every dog owner is an animal lover
- No animal lover kills an animal
- Either Jack or curiosity killed the cat

- Cats are animals

Query: Did curiosity kill the cat?

First Order Knowledge Base:

- Owns(Jack, Dog)
- $\forall$ x ($exists$ y Owns(x, y) $\wedge$ Dog(y)) $\implies$ ALover(x)
- $\forall$ x ALover(x) $\implies$ ($\forall$ y Animal(y) $\implies$ $\neg$kills(x, y))
- Kills(Jack, Tuna) $\vee$ Kills(Curiosity, Tuna)
- Cat(Tuna)
- $\forall$ x Cat(x) $\implies$ Animal(x)

CNF Knowledge Base:

1. Dog(D)
2. Owns(Jack, D)
3. $\neg$Owns(x, y) $\vee$ $\neg$Dog(y) $\vee$ ALover(x)
4. $\neg$ALover(x) $\vee$ $\neg$Animal(y) $\vee$ $\neg$Kills(x, y)
5. Kills(Jack, Tuna) $\vee$ Kills(Curiosity, Tuna)
6. Cat(Tuna)
7. $\neg$Cat(x) $\vee$ Animal(x)
8. Query — $\neg$Kills(Curiosity, Tuna)

9. $\neg$Owns(x, D) $\vee$ ALover(x) by <1, 5>: y/D
10. ALover(Jack) by <2, 9>: x/Jack
11. Animal(Tuna) by <7, 8>: x/Tuna
12. $\neg$ALover(x) $\vee$ $\neg$Kills(x, Tuna) by <4, 11>: x/Tuna
13. $\neg$Kills(Jack, Tuna) by <10, 12>: x/Jack
14. Kills(Jack, Tuna) by <8, 5>
15. CONTRADICTION by <12, 13>

To perform resolution, we need a CNF; we convert to a CNF much the same as with propositional: Consider: $\forall$ x [$\forall$ y Animal(y) $\implies$ Loves(x, y)] $\implies$ [$\exists$ y Loves(y, x)]

Step 1: eliminate $\implies$ and $\iff$

$\forall$ x [$\neg\forall$ y $\neg$Animal(y) $\vee$ Loves(x, y)] $\vee$ [$\exists$ y Loves(y, x)]

Step 2: move negation inward

$\forall$ x [$\exists$ y Animal(y) $\wedge$ $\neg$Loves(x, y)] $\vee$ [$\exists$ y Loves(y, x)]

Step 3: standardize variables

$\forall$ x [$\exists$ y Animal(y) $\wedge$ $\neg$Loves(x, y)] $\vee$ [$\exists$ z Loves(z, x)]

Step 4: "skolem"-ize

$\forall$ x [Animal(F(x)) $\wedge$ $\neg$Loves(x, F(x))] $\vee$ [Loves(G(x), x)]

Step 5: drop universal quantifiers

[Animal(F(x)) $\wedge$ $\neg$Loves(x, F(x))] $\vee$ [Loves(G(x), x)]

Step 6: distribute

[Animal(F(x)) $\vee$ Loves(G(x), x)] $\vee$ [$\neg$Loves(x, F(x)) $\vee$ Loves(G(x), x)]

We have a special case when the knowledge base contains only **definite clauses**.

a definite clause contains exactly one positive term (ex $\neg$A $\vee$ $\neg$B $\vee$ C)

we can thus convert these to an if-then rule (ex $A \wedge B \implies C$)

We will show two special algorithms for dealing with definite clause knowledge bases.
We will do this with the following example:

1. American(x) $\wedge$ Weapon(x) $\wedge$ Sells(x, y, z) $\wedge$ Hostile(z) $\implies$ Criminal(x)
2. $\exists$ x Owns(Nono, x) $\wedge$ Missile(x)
   Owns(Nono, M1) $\wedge$ Missile(M1)
3. Missile(x) $\wedge$ Owns(Nono, x) $\implies$ sells(West, x, Nono)
4. Missile(x) $\implies$ Weapon(x)
5. Enemy(x, America) $\implies$ Hostile(x)
6. American(West)
7. Enemy(Nono, America)

Notice that to be operated on, all clauses must be universally quantified

Forward Chaining

![](https://paper − attachments.dropbox.com/s$_3$7BC3BDDFB4E93FABD6C94C74D020C3CD8B25F78236483B019E726793

If there are no function symbols and all clauses are definite, the KB is a **data log**
These are especially conducive to representing databases.
If we preprocess as rules come in, we can handle many facts and even model a brain in real time.

Backward Chaining

![](https://paper − attachments.dropbox.com/s$_3$7BC3BDDFB4E93FABD6C94C74D020C3CD8B25F78236483B019E726793

This is often used with improvements for logic programming (as in prolog)
The system often caches subgoals (like Missile(y)) for efficiency's sake