First-Order Logic (FOL)
aka. predicate calculus

22c:145
Computer Science Department
The University of Iowa

First-Order Logic (FOL) *Syntax*

- User defines these primitives:
  - **Constant symbols** (i.e., the "individuals" in the world)
    E.g., Mary, 3
  - **Function symbols** (mapping individuals to individuals)
    E.g., father-of(Mary) = John, color-of(Sky) = Blue
  - **Predicate symbols** (mapping from individuals to truth values)
    E.g., greater(5,3), green(Grass), color(Grass, Green)

First-Order Logic (FOL) *Syntax…*

- FOL supplies these primitives:
  - **Variable symbols**. E.g., `x,y`
  - **Connectives**. Same as in PL: not (~), and (^), or (v), implies (=>), if and only if (<=>)
  - **Quantifiers**: Universal (A) and Existential (E)

Quantifiers

- Universal quantification corresponds to conjunction ("and") in that **`(Ax)P(x)`** means that **`P`** holds for all values of **`x`** in the domain associated with that variable.
  - E.g., **`(Ax) dolphin(x) => mammal(x)`**
- Existential quantification corresponds to disjunction ("or") in that **`(Ex)P(x)`** means that **`P`** holds for some value of **`x`** in the domain associated with that variable.
  - E.g., **`(Ex) mammal(x) ^ lays-eggs(x)`**
- Universal quantifiers are usually used with "implies" to form "if-then rules."
  - E.g., **`(Ax) cs15-381-student(x) => smart(x)`** means "All cs15-381 students are smart."
  - You rarely use universal quantification to make blanket statements about every individual in the world: **`(Ax)cs15-381-student(x) ^ smart(x)`** meaning that everyone in the world is a cs15-381 student and is smart.

Quantifiers …

- Existential quantifiers are usually used with "and" to specify a list of properties or facts about an individual.
  - E.g., **`(Ex) cs15-381-student(x) ^ smart(x)`** means "there is a cs15-381 student who is smart."
  - A common mistake is to represent this English sentence as the FOL sentence: **`(Ex) cs15-381-student(x) => smart(x)`**
- Switching the order of universal quantifiers does not change the meaning: **`(Ax)(Ay)P(x,y)`** is logically equivalent to **`(Ay)(Ax)P(x,y).`** Similarly, you can switch the order of existential quantifiers.
- Switching the order of universals and existentials *does* change meaning:
  - Everyone likes someone: **`(Ax)(Ey)likes(x,y)`**
  - Someone is liked by everyone: **`(Ey)(Ax)likes(x,y)`**

First-Order Logic (FOL) *Syntax…*

- *Sentences* are built up of terms and atoms:
  - A **term** (denoting a real-world object) is a constant symbol, a variable symbol, or a function e.g. left-leg-of ( ). For example, *x* and *f(x1, ..., xn)* are terms, where each *xi* is a term.
  - An **atom** (which has value true or false) is either an n-place predicate of n terms, or, if **`P`** and **`Q`** are atoms, then **`~P, P V Q, P ^ Q, P => Q, P <=> Q`** are atoms
  - A **sentence** is an atom, or, if **`P`** is a sentence and **`x`** is a variable, then **`(Ax)P`** and **`(Ex)P`** are sentences
  - A **well-formed formula (wff)** is a sentence containing no "free" variables. I.e., all variables are "bound" by universal or existential quantifiers.
    - E.g., **`(Ax)P(x,y)`** has **`x`** bound as a universally quantified variable, but **`y`** is free.

Translating English to FOL

- Every gardener likes the sun.
  `(Ax) gardener(x) => likes(x,Sun)`
- You can fool some of the people all of the time.
  `(Ex)(At) (person(x) ^ time(t)) => can-fool(x,t)`
- You can fool all of the people some of the time.
  `(Ax)(Et) (person(x) ^ time(t) => can-fool(x,t)`
- All purple mushrooms are poisonous.
  `(Ax) (mushroom(x) ^ purple(x)) => poisonous(x)`

Translating English to FOL…

- No purple mushroom is poisonous.
  `~(Ex) purple(x) ^ mushroom(x) ^ poisonous(x)`
  or, equivalently,
  `(Ax) (mushroom(x) ^ purple(x)) => ~poisonous(x)`
- There are exactly two purple mushrooms.
  `(Ex)(Ey) mushroom(x) ^ purple(x) ^ mushroom(y) ^ purple(y) ^ ~(x=y) ^ (Az) (mushroom(z) ^ purple(z)) => ((x=z) v (y=z))`
- Deb is not tall.
  `~tall(Deb)`
- X is above Y if X is directly on top of Y or else there is a pile of one or more other objects directly on top of one another starting with X and ending with Y.
  `(Ax)(Ay) above(x,y) <=> (on(x,y) v (Ez) (on(x,z) ^ above(z,y)))`

Automated inference for FOL

- Automated inference using FOL is harder than PL
  - Variables can potentially take on an *infinite* number of possible values from their domains
  - Hence there are potentially an *infinite* number of ways to apply the Universal Elimination rule of inference
- *Godel's Completeness Theorem* says that FOL entailment is only *semidecidable*
  - If a sentence is **true** given a set of axioms, there is a procedure that will determine this
  - If the sentence is **false**, then there is no guarantee that a procedure will ever determine this — i.e., it **may never halt**

Generalized Modus Ponens

- Modus Ponens
  - P, P=>Q |= Q
- Generalized Modus Ponens (GMP) extends this to rules in FOL
- Combines And-Introduction, Universal-Elimination, and Modus Ponens, e.g.
  - *from* $P(c)$ *and* $Q(c)$ *and* $\forall x\ P(x) \wedge Q(x) \rightarrow R(x)$ *derive* $R(c)$
- Need to deal with
  - more than one condition on left side of rule
  - variables

Generalized Modus Ponens

- General case: **Given**
  - **atomic sentences** $P_1, P_2, \ldots, P_N$
  - **implication sentence** $(Q_1 \wedge Q_2 \wedge \ldots \wedge Q_N) \rightarrow R$
    - $Q_1, \ldots, Q_N$ and R are atomic sentences
  - **substitution** $\text{subst}(\theta, P_i) = \text{subst}(\theta, Q_i)$ for i=1,...,N
  - **Derive new sentence: subst(θ, R)**
- Substitutions
  - subst(θ, α) denotes the result of applying a set of substitutions defined by θ to the sentence α
  - A substitution list $\theta = \{v_1/t_1, v_2/t_2, \ldots, v_n/t_n\}$ means to replace all occurrences of variable symbol v_i by term t_i
  - Substitutions made in left-to-right order in the list
  - subst({x/Cheese, y/Mickey}, eats(y,x)) = eats(Mickey, Cheese)

Unification

- Unification is a "pattern matching" procedure that takes two atomic sentences, called **literals**, as input, and returns "failure" if they do not match and a substitution list, Theta, if they do match.
  - That is, `unify(p,q) = Theta` means `subst(Theta, p) = subst(Theta, q)` for two atomic sentences `p` and `q`.
  - `Theta` is called the **most general unifier (mgu)**
- All variables in the given two literals are implicitly universally quantified
- To make literals match, replace (universally-quantified) variables by terms

Unification algorithm

```
procedure unify(p, q, θ)
    Scan p and q left-to-right and find the first corresponding
      terms where p and q “disagree” (i.e., p and q not equal)
    If there is no disagreement, return θ  (success!)
    Let r and s be the terms in p and q, respectively,
      where disagreement first occurs
    If variable(r) then {
      Let θ = union(θ, {r/s})
      Return unify(subst(θ, p), subst(θ, q), θ)
    } else if variable(s) then {
      Let θ = union(θ, {s/r})
      Return unify(subst(θ, p), subst(θ, q), θ)
    } else return “Failure”
  end
```

13

Unification…

- Examples

Literal 1	Literal 2	Literal 3
`parents(x, father(x), mother(Bill))`	`parents(Bill, father(Bill), y)`	`{x/Bill, y/mother(Bill)}`
`parents(x, father(x), mother(Bill))`	`parents(Bill, father(y), z)`	`{x/Bill, y/Bill, z/mother(Bill)}`
`parents(x, father(x), mother(Jane))`	`parents(Bill, father(y), mother(y))`	`Failure`

Unification…

- Unify is a linear time algorithm that returns the **most general unifier (mgu)**, i.e., a shortest length substitution list that makes the two literals match.
  - (In general, there is not a unique minimum length substitution list, but unify returns one of them.)
- A variable can never be replaced by a term containing that variable. For example, `x/f(x)` is illegal. This **"occurs check"** should be done in the above pseudo-code before making the recursive calls.

More Unification Examples

- Make sentences look alike.
- Unify p(a,X) and p(a,b)
- Unify p(a,X) and p(Y,b)
- Unify p(a,X) and p(Y, f(Y))
- Unify p(a,X) and p(X,b)
- Unify p(a,X) and p(Y,b)
- Unify p(a,b) and p(X, X)

More Unification Examples

- Unify p(a,X) and p(a,b)
- answer: X/b p(a,b)
- Unify p(a,X) and p(Y,b)
- answer: Y/a, X/b p(a,b)
- Unify p(a,X) and p(Y, f(Y))
- answer: Y/a, X/f(a) p(a,f(a))

More Unification Examples

- Unify p(a,X) and p(X,b)
- failure
- Unify p(a,X) and p(Y,b)
- answer: Y/a, X/b p(a,b)
- Unify p(a,b) and p(X,X)
- failure
- Unify p(X, f(Y), b) and P(X, f(b), b)
- answer: Y/b this is an mgu
- X/b, Y/b this in not an mgu

Most general unifier (mgu)

- If s is any unifier of expressions E and g is the most general unifier of E, then for s applied to E there exists another unifier s' such that subst(s, E) = subst(s', subs(g, E)).
- Basic idea: Commit a variable to an expression only if you have to; keep it as general as possible.

Assume KB are Horn clauses

- A Horn clause is a sentence of the form:
  $P_1(x) \wedge P_2(x) \wedge ... \wedge P_n(x) \rightarrow Q(x)$

where
  - ≥ 0 P_is and 0 or 1 Q
  - the P_is and Q are positive (i.e., non-negated) literals
- Equivalently: $P_1(x) \vee P_2(x) \ldots \vee P_n(x)$ where the P_i are all literals and *at most one* is positive
- Prolog is based on Horn clauses
- Horn clauses represent a *subset* of the set of sentences representable in FOL

Horn clauses II

- Several cases
  - *A fact:* Q
  - *Typical rule:* $P_1 \wedge P_2 \wedge \ldots P_n \rightarrow Q$
  - *Constraint:* $P_1 \wedge P_2 \wedge \ldots P_n \rightarrow$ false
- In Prolog
  - Q.
  - Q :- P_1 , P_2 , … , P_n .
  - ?- P_1 , P_2 , … , P_n . (only in a query)
- These are not Horn clauses:
  - $p(a) \vee q(a)$
  - $(P \wedge Q) \rightarrow (R \vee S)$

Horn clauses III

- Where are the quantifiers?
  - Variables appearing in clauses are universally quantified
- Example: grandparent relation
  - parent(P1, X) $\wedge$ parent(X, P2) $\rightarrow$ grandParent(P1, P2)
  - $\forall$ P1,P2, X (parent(P1,X) $\wedge$ parent(X, P2) $\rightarrow$ grandParent(P1, P2))
  - Prolog:
    grandParent(P1,P2) :- parent(P1,X), parent(X,P2).

Forward & Backward Reasoning

- We usually talk about two reasoning strategies: Forward and backward 'chaining'
- Both are equally powerful
- You can also have a mixed strategy

Forward chaining

- Proofs start with the given axioms/premises in KB, deriving new sentences using GMP until the goal/query sentence is derived
- This defines a **forward-chaining** inference procedure because it moves "forward" from the KB to the goal [eventually]
- Inference using GMP is **sound** and **complete** for KBs containing **only Horn clauses**

Forward chaining algorithm

```
procedure FORWARD-CHAIN(KB, p)

  if there is a sentence in KB that is a renaming of p then return
  Add p to KB
  for each (p1 ∧ ... ∧ pn ⇒ q) in KB such that for some i, UNIFY(pi, p) = θ succeeds do
      FIND-AND-INFER(KB, [p1, ..., pi-1, pi+1, ..., pn], q, θ)
  end
```

```
procedure FIND-AND-INFER(KB, premises, conclusion, θ)

  if premises = [ ] then
      FORWARD-CHAIN(KB, SUBST(θ, conclusion))
  else for each p' in KB such that UNIFY(p', SUBST(θ, FIRST(premises))) = θ2 do
      FIND-AND-INFER(KB, REST(premises), conclusion, COMPOSE(θ, θ2))
  end
```

Forward chaining example

- KB:
  - allergies(X) → sneeze(X)
  - cat(Y) ∧ allergicToCats(X) → allergies(X)
  - cat(felix)
  - allergicToCats(mary)
- Goal:
  - sneeze(mary)

Example of forward chaining

Example: KB = All cats like fish, cats eat everything they like, and Ziggy is a cat. In FOL, KB =

```
1.  (Ax) cat(x) => likes(x, Fish)
2.  (Ax)(Ay) (cat(x) ^ likes(x,y)) => eats(x,y)
3.  cat(Ziggy)
```

- Goal query: Does Ziggy eat fish?

Proof: Data-driven

1. Use GMP with (1) and (3) to derive: 4. `likes(Ziggy, Fish)`
2. Use GMP with (3), (4) and (2) to derive `eats(Ziggy, Fish)`
3. So, yes, Ziggy eats fish.

Backward chaining

- **Backward-chaining** deduction using GMP is also **complete** for KBs containing **only Horn clauses**
- Proofs start with the goal query, find rules with that conclusion, and then prove each of the antecedents in the implication
- Keep going until you reach premises
- Avoid loops: check if new subgoal is already on the goal stack
- Avoid repeated work: check if new subgoal
  - Has already been proved true
  - Has already failed

Backward chaining algorithm

```
function BACK-CHAIN(KB, q) returns a set of substitutions

  BACK-CHAIN-LIST(KB, [q], {})
```

```
function BACK-CHAIN-LIST(KB, qlist, θ) returns a set of substitutions
  inputs: KB, a knowledge base
          qlist, a list of conjuncts forming a query (θ already applied)
          θ, the current substitution
  static: answers, a set of substitutions, initially empty

  if qlist is empty then return {θ}
  q ← FIRST(qlist)
     for each qi' in KB such that θi ← UNIFY(q, qi') succeeds do
        Add COMPOSE(θ, θi) to answers
     end
     for each sentence (p1 ∧ ... ∧ pn ⇒ qi') in KB such that θi ← UNIFY(q, qi') succeeds do
        answers ← BACK-CHAIN-LIST(KB, SUBST(θi, [p1 ... pn]), COMPOSE(θ, θi)) ∪ answers
     end
  return the union of BACK-CHAIN-LIST(KB, REST(qlist), θ) for each θ ∈ answers
```

Backward chaining example

- KB:
  - allergies(X) → sneeze(X)
  - cat(Y) ∧ allergicToCats(X) → allergies(X)
  - cat(felix)
  - allergicToCats(mary)
- Goal:
  - sneeze(mary)

Forward vs. backward chaining

- FC is *data-driven*
  - Automatic, unconscious processing
  - E.g., object recognition, routine decisions
  - May do lots of work that is irrelevant to the goal
  - Efficient when you want to compute all conclusions
- BC is goal-driven, better for problem-solving
  - Where are my keys? How do I get to my next class?
  - Complexity of BC can be much less than linear in the size of the KB
  - Efficient when you want one or a few decisions

Mixed strategy

- Many practical reasoning systems do both forward and backward chaining
- The way you encode a rule determines how it is used, as in

  % this is a forward chaining rule
  spouse(X,Y) => spouse(Y,X).
  % this is a backward chaining rule
  wife(X,Y) <= spouse(X,Y), female(X).
- Given a model of the rules you have and the kind of reason you need to do, it's possible to decide which to encode as FC and which as BC rules.

Completeness of GMP

- GMP (using forward or backward chaining) is complete for KBs that contain only Horn clauses
- ***not* complete** for simple KBs with **non-Horn clauses**
- The following entail that S(A) is true:
  1. $(\forall x)\ P(x) \rightarrow Q(x)$
  2. $(\forall x)\ \neg P(x) \rightarrow R(x)$
  3. $(\forall x)\ Q(x) \rightarrow S(x)$
  4. $(\forall x)\ R(x) \rightarrow S(x)$
- If we want to conclude S(A), with GMP we cannot, since the second one is not a Horn clause
- It is equivalent to $P(x) \vee R(x)$

Resolution

- Resolution is a **sound** and **complete** inference procedure for unrestricted FOL
- Reminder: Resolution rule for propositional logic:
  - $P_1 \vee P_2 \vee \ldots \vee P_n$
  - $\neg P_1 \vee Q_2 \vee \ldots \vee Q_m$
  - Resolvent: $P_2 \vee \ldots \vee P_n \vee Q_2 \vee \ldots \vee Q_m$
- We'll need to extend this to handle quantifiers and variables

Resolution covers many cases

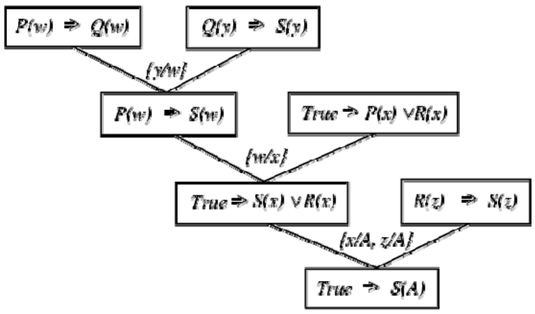
- Modes Ponens
  - from P and $P \rightarrow Q$ derive Q
  - from P and $\neg P \vee Q$ derive Q
- Chaining
  - from $P \rightarrow Q$ and $Q \rightarrow R$ derive $P \rightarrow R$
  - from $(\neg P \vee Q)$ and $(\neg Q \vee R)$ derive $\neg P \vee R$
- Contradiction detection
  - from P and $\neg P$ derive false
  - from P and $\neg P$ derive the empty clause (=false)

Resolution in first-order logic

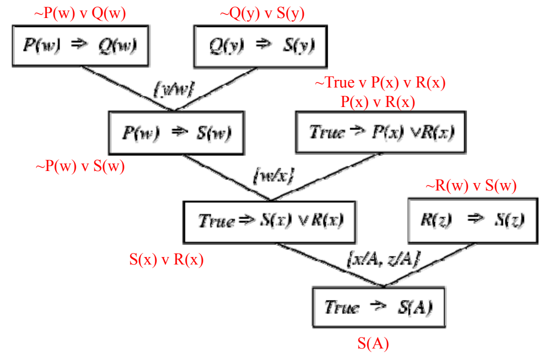
- Given sentences in *conjunctive normal form:*
  - $P_1 \vee \ldots \vee P_n$ and $Q_1 \vee \ldots \vee Q_m$
  - P_i and Q_i are literals, i.e., positive or negated predicate symbol with its terms
- if P_j and $\neg Q_k$ **unify** with substitution list θ, then derive the resolvent sentence:

  $subst(\theta, P_1 \vee \ldots \vee P_{j-1} \vee P_{j+1} \ldots P_n \vee Q_1 \vee \ldots Q_{k-1} \vee Q_{k+1} \vee \ldots \vee Q_m)$
- Example
  - from clause $\mathbf{P(x, f(a)) \vee P(x, f(y)) \vee Q(y)}$
  - and clause $\mathbf{\neg P(z, f(a)) \vee \neg Q(z)}$
  - derive resolvent $\mathbf{P(z, f(y)) \vee Q(y) \vee \neg Q(z)}$
  - Using $\boldsymbol{\theta = \{x/z\}}$

A resolution proof tree



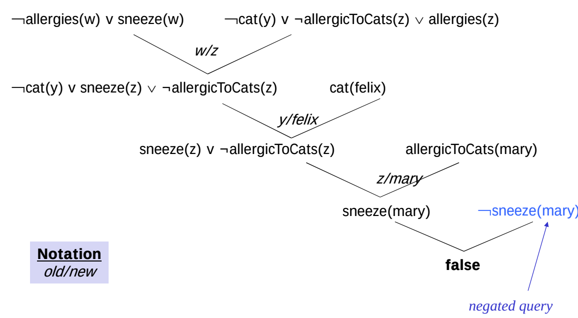
A resolution proof tree



Resolution example

- KB:
  - allergies(X) → sneeze(X)
  - cat(Y) ∧ allergicToCats(X) → allergies(X)
  - cat(felix)
  - allergicToCats(mary)
- Goal:
  - sneeze(mary)

Refutation resolution proof tree



questions to be answered

- How to convert FOL sentences to conjunctive normal form (a.k.a. CNF, clause form): **normalization and skolemization**
- How to unify two argument lists, i.e., how to find their most general unifier (**mgu**) q: **unification**
- How to determine which two clauses in KB should be resolved next (among all resolvable pairs of clauses) : **resolution (search) strategy**

Resolution Algorithm

```
procedure resolution-refutation(KB, Q)
  ;; KB is a set of consistent, true FOL sentences
  ;; Q is a goal sentence that we want to derive
  ;; return success if KB |- Q, and failure otherwise
  KB = union(KB, ~Q)
  while false not in KB do
     pick 2 sentences, S1 and S2, in KB that contain
     literals that resolve(if none, return "failure")

     resolvent = resolution-rule(S1, S2)

     KB = union(KB, resolvent)

  return "success"
```

Resolution example (using PL sentences)

- From "Heads I win, tails you lose" prove that "I win"
- First, define the axioms in KB:
  1. "Heads I win, tails you lose."
     `(Heads => IWin)` or, equivalently, `(~Heads v IWin)`
     `(Tails => YouLose)` or, equivalently, `(~Tails v YouLose)`
  2. Add some general knowledge axioms about coins, winning, and losing:
     `(Heads v Tails)`
     `(YouLose => IWin)` or, equivalently, `(~YouLose v IWin)`
     `(IWin => YouLose)` or, equivalently, `(~IWin v YouLose)`
- Goal: `IWin`

Resolution example (using PL sentences)…

Sentence 1	Sentence 2	Resolvent
`~IWin`	`~Heads v IWin`	`~Heads`
`~Heads`	`Heads v Tails`	`Tails`
`Tails`	`~Tails v YouLose`	`YouLose`
`YouLose`	`~YouLose v Iwin`	`IWin`
`IWin`	`~IWin`	`False`

Problems yet to be addressed

- Resolution rule of inference is only applicable with sentences that are in the form `P1 v P2 v ... v Pn`, where each `Pi` is a negated or nonnegated predicate and contains functions, constants, and universally quantified variables, so can we convert every FOL sentence into this form?
- Resolution *strategy*
  - How to pick which pair of sentences to resolve?
  - How to pick which pair of literals, one from each sentence, to unify?

Converting sentences to CNF

1. Eliminate all $\leftrightarrow$ connectives
   $(P \leftrightarrow Q) \Rightarrow ((P \rightarrow Q) \wedge (Q \rightarrow P))$
2. Eliminate all $\rightarrow$ connectives
   $(P \rightarrow Q) \Rightarrow (\neg P \vee Q)$
3. Reduce the scope of each negation symbol to a single predicate
   $\neg\neg P \Rightarrow P$
   $\neg(P \vee Q) \Rightarrow \neg P \wedge \neg Q$
   $\neg(P \wedge Q) \Rightarrow \neg P \vee \neg Q$
   $\neg(\forall x)P \Rightarrow (\exists x)\neg P$
   $\neg(\exists x)P \Rightarrow (\forall x)\neg P$
4. Standardize variables: rename all variables so that each quantifier has its own unique variable name

Converting sentences to clausal form
Skolem constants and functions

5. Eliminate existential quantification by introducing Skolem constants/functions
   $(\exists x)P(x) \Rightarrow P(C)$
   C is a Skolem constant (a brand-new constant symbol that is not used in any other sentence)
   $(\forall x)(\exists y)P(x,y) \Rightarrow (\forall x)P(x, f(x))$
   since $\exists$ is within scope of a universally quantified variable, use a **Skolem function f** to construct a new value that **depends on** the universally quantified variable
   f must be a brand-new function name not occurring in any other sentence in the KB
   E.g., $(\forall x)(\exists y)loves(x,y) \Rightarrow (\forall x)loves(x,f(x))$
   In this case, f(x) specifies the person that x loves
   a better name might be **oneWhoIsLovedBy**(x)

Converting sentences to clausal form

6. Remove universal quantifiers by (1) moving them all to the left end; (2) making the scope of each the entire sentence; and (3) dropping the "prefix" part
   Ex: $(\forall x)P(x) \Rightarrow P(x)$
7. Put into conjunctive normal form (conjunction of disjunctions) using distributive and associative laws
   $(P \wedge Q) \vee R \Rightarrow (P \vee R) \wedge (Q \vee R)$
   $(P \vee Q) \vee R \Rightarrow (P \vee Q \vee R)$
8. Split conjuncts into separate clauses
9. Standardize variables so each clause contains only variable names that do not occur in any other clause

An example

$(\forall x)(P(x) \rightarrow ((\forall y)(P(y) \rightarrow P(f(x,y))) \wedge \neg(\forall y)(Q(x,y) \rightarrow P(y))))$

2. Eliminate $\rightarrow$
   $(\forall x)(\neg P(x) \vee ((\forall y)(\neg P(y) \vee P(f(x,y))) \wedge \neg(\forall y)(\neg Q(x,y) \vee P(y))))$
3. Reduce scope of negation
   $(\forall x)(\neg P(x) \vee ((\forall y)(\neg P(y) \vee P(f(x,y))) \wedge (\exists y)(Q(x,y) \wedge \neg P(y))))$
4. Standardize variables
   $(\forall x)(\neg P(x) \vee ((\forall y)(\neg P(y) \vee P(f(x,y))) \wedge (\exists z)(Q(x,z) \wedge \neg P(z))))$
5. Eliminate existential quantification
   $(\forall x)(\neg P(x) \vee ((\forall y)(\neg P(y) \vee P(f(x,y))) \wedge (Q(x,g(x)) \wedge \neg P(g(x)))))$
6. Drop universal quantification symbols
   $(\neg P(x) \vee ((\neg P(y) \vee P(f(x,y))) \wedge (Q(x,g(x)) \wedge \neg P(g(x)))))$

Example

7. Convert to conjunction of disjunctions
   $(\neg P(x) \vee \neg P(y) \vee P(f(x,y))) \wedge (\neg P(x) \vee Q(x,g(x))) \wedge (\neg P(x) \vee \neg P(g(x)))$
8. Create separate clauses
   $\neg P(x) \vee \neg P(y) \vee P(f(x,y))$
   $\neg P(x) \vee Q(x,g(x))$
   $\neg P(x) \vee \neg P(g(x))$
9. Standardize variables
   $\neg P(x) \vee \neg P(y) \vee P(f(x,y))$
   $\neg P(z) \vee Q(z,g(z))$
   $\neg P(w) \vee \neg P(g(w))$

Converting FOL sentences to clause form…

Examples:

- Convert the sentence
- `(Ax)(P(x) => ((Ay)(P(y) => P(f(x,y))) ^ ~(Ay)(Q(x,y) => P(y))))`

1. Eliminate `<=>`
   Nothing to do here.
2. Eliminate `=>`
   `(Ax)(~P(x) v ((Ay)(~P(y) v P(f(x,y))) ^ ~(Ay)(~Q(x,y) v P(y))))`
3. Reduce scope of negation
   `(Ax)(~P(x) v ((Ay)(~P(y) v P(f(x,y))) ^ (Ey)(Q(x,y) ^ ~P(y))))`

Converting FOL sentences to clause form…

4. Standardize variables
   `(Ax)(~P(x) v ((Ay)(~P(y) v P(f(x,y))) ^ (Ez)(Q(x,z) ^ ~P(z))))`
5. Eliminate existential quantification
   `(Ax)(~P(x) v ((Ay)(~P(y) v P(f(x,y))) ^ (Q(x,g(x)) ^ ~P(g(x)))))`
6. Drop universal quantification symbols
   `(~P(x) v ((~P(y) v P(f(x,y))) ^ (Q(x,g(x)) ^ ~P(g(x)))))`
7. Convert to conjunction of disjunctions
   `(~P(x) v ~P(y) v P(f(x,y))) ^ (~P(x) v Q(x,g(x))) ^ (~P(x) v ~P(g(x)))`

Converting FOL sentences to clause form…

8. Create separate clauses
   - `~P(x) v ~P(y) v P(f(x,y))`
   - `~P(x) v Q(x,g(x))`
   - `~P(x) v ~P(g(x))`
9. Standardize variables
   - `~P(x) v ~P(y) v P(f(x,y))`
   - `~P(z) v Q(z,g(z))`
   - `~P(w) v ~P(g(w))`

Colonel West is a criminal

1. It is a crime for an American to sell weapons to a hostile country.
2. The country Nono has some missiles.
3. All of its missiles were sold to it by Colonel West.
4. Nono is an enemy of USA.
5. Colonel West is an American.

Modeling with Horn Clauses: at most one positive literal

1. It is a crime for an American to sell weapons to a hostile country.

1'. American(x)&Weapons(y)&Hostile(z) & Sell(x,y,z) => Criminal (x).

2. The country Nono has some missiles.

There exists x Owns(Nono,x)&Missile(x).

2'. Missile(M1). … Skolem Constant introduction

2''. Owns(Nono,M1).

Prove: West is a criminal

3. All of its missiles were sold to it by Colonel West.

3'. Missile(x)&Owns(Nono,x) => Sells(West,x,Nono).

4'. Missile(x) => Weapon(x). .. "common sense"

5'. Enemy(x,America) => Hostile(x).

6'. American(West).

7'. Enemy(Nono,American).

Forward Chaining

- Start with facts and apply rules until no new facts appear. Apply means use substitutions.
- Iteration 1: using facts.
- Missile(M1), American(West), Owns(Nono,M1), Enemy(Nono,America)
- Derive: Hostile(Nono), Weapon(M1), Sells(West,M1,Nono).
- Next Iteration: Criminal(West).
- Forward chaining ok if few facts and rules, but it is undirected.

Resolution gives forward chaining

- From
  Enemy(x,America)
  =>Hostile(x),
  Enemy(Nono,America)
- Resolvent:
  Hostile(Nono)

- From
  not Enemy(x,America)
  or Hostile(x),
  Enemy(Nono,America)
- Resolve by {x/Nono}
- Resolvent:
  Hostile(Nono)

Backward Chaining

- Start with goal, Criminal(West) and set up subgoals. This ends when all subgoals are validated.
- Iteration 1: subgoals American(x), Weapons(y) and Hostile(z).
- Etc. Eventually all subgoals unify with facts.

Resolution yeilds Backward Chaining

- A(x) &W(y)&H(z)& S(x,y,z) =>C(x)

- – A(x) or –W(y) or
  –H(z) or –S(x,y,z) or C(x).
- Add goal –C(West).
- Yields –A(West) or
  – W(y) or –H(z) or
  – S(West,y,z). Etc.

Example: Hoofers Club

- **Problem Statement:** Tony, Shi-Kuo and Ellen belong to the Hoofers Club. Every member of the Hoofers Club is either a skier or a mountain climber or both. No mountain climber likes rain, and all skiers like snow. Ellen dislikes whatever Tony likes and likes whatever Tony dislikes. Tony likes rain and snow.
- **Query:** Is there a member of the Hoofers Club who is a mountain climber but not a skier?

Example: Hoofers Club…

- **Translation into FOL Sentences**
- Let `S(x)` mean `x` is a skier, `M(x)` mean x is a mountain climber, and `L(x,y)` mean `x` likes `y`, where the domain of the first variable is Hoofers Club members, and the domain of the second variable is snow and rain. We can now translate the above English sentences into the following FOL wffs:

```
1.   (Ax) S(x) v M(x)
2.   ~(Ex) M(x) ^ L(x, Rain)
3.   (Ax) S(x) => L(x, Snow)
4.   (Ay) L(Ellen, y) <=> ~L(Tony, y)
5.   L(Tony, Rain)
6.   L(Tony, Snow)

7.   Query: (Ex) M(x) ^ ~S(x)
8.   Negation of the Query: ~(Ex) M(x) ^ ~S(x)
```

Example: Hoofers Club…

- **Conversion to Clause Form**

```
1.  S(x1) v M(x1)
2.  ~M(x2) v ~L(x2, Rain)
3.  ~S(x3) v L(x3, Snow)
4.  ~L(Tony, x4) v ~L(Ellen, x4)
5.  L(Tony, x5) v L(Ellen, x5)
6.  L(Tony, Rain)
7.  L(Tony, Snow)
8.  Negation of the Query: ~M(x7) v S(x7)
```

Example: Hoofers Club…

- **Resolution Refutation Proof**

Clause 1	Clause 2	Resolvent	MGU (i.e., Theta)
8	1	9. S(x1)	{x7/x1}
9	3	10. L(x1, Snow)	{x3/x1}
10	4	11. ~L(Tony, Snow)	{x4/Snow, x1/Ellen}
11	7	12. False	{}

Example: Hoofers Club…

- **Answer Extraction**

Clause 1	Clause 2	Resolvent	MGU (i.e., Theta)
~M(x7) v S(x7) v (M(x7) ^ ~S(x7))	1	9. S(x1) v (M(x1) ^ ~S(x1))	{x7/x1}
9	3	10. L(x1, Snow) v (M(x1) ^ ~S(x1))	{x3/x1}
10	4	11. ~L(Tony, Snow) v (M(Ellen) ^ ~S(Ellen))	{x4/Snow, x1/Ellen}
11	7	12. M(Ellen) ^ ~S(Ellen)	{}

- **Answer to the query:** Ellen!

Resolution Theorem Proving as search

- Resolution can be thought of as the **bottom-up construction of a search tree**, where the leaves are the clauses produced by KB and the negation of the goal
- When a pair of clauses generates a new resolvent clause, add a new node to the tree with arcs directed from the resolvent to the two parent clauses
- **Resolution succeeds** when a node containing the **False** clause is produced, becoming the **root node** of the tree
- A strategy is **complete** if its use guarantees that the empty clause (i.e., false) can be derived whenever it is entailed

Strategies

- There are a number of general (domain-independent) strategies that are useful in controlling a resolution theorem prover
- We'll briefly look at the following:
  - Breadth-first
  - Length heuristics
  - Set of support
  - Input resolution
  - Subsumption
  - Ordered resolution

Example

1. Battery-OK ∧ Bulbs-OK → Headlights-Work
2. Battery-OK ∧ Starter-OK → Empty-Gas-Tank ∨ Engine-Starts
3. Engine-Starts → Flat-Tire ∨ Car-OK
4. Headlights-Work
5. Battery-OK
6. Starter-OK
7. ¬Empty-Gas-Tank
8. ¬Car-OK
9. Goal: Flat-Tire ?

Example

1. ¬Battery-OK ∨ ¬Bulbs-OK ∨ Headlights-Work
2. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
3. ¬Engine-Starts ∨ Flat-Tire ∨ Car-OK
4. Headlights-Work
5. Battery-OK
6. Starter-OK
7. ¬Empty-Gas-Tank
8. ¬Car-OK
9. ¬Flat-Tire ← **negated goal**

Breadth-first search

- Level 0 clauses are the original axioms and the negation of the goal
- Level k clauses are the resolvents computed from two clauses, one of which must be from level k-1 and the other from any earlier level
- Compute all possible level 1 clauses, then all possible level 2 clauses, etc.
- Complete, but very inefficient

BFS example

	1.	¬Battery-OK ∨ ¬Bulbs-OK ∨ Headlights-Work
	2.	¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
	3.	¬Engine-Starts ∨ Flat-Tire ∨ Car-OK
	4.	Headlights-Work
	5.	Battery-OK
	6.	Starter-OK
	7.	¬Empty-Gas-Tank
	8.	¬Car-OK
	9.	¬Flat-Tire
1,5	10.	¬Bulbs-OK ∨ Headlights-Work
2,3	11.	¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Flat-Tire ∨ Car-OK
2,5	12.	¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
2,6	13.	¬Battery-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
2,7	14.	¬Battery-OK ¬ Starter-OK ∨ Engine-Starts
3,8	15.	¬Engine-Starts ∨ Flat-Tire
	16.	... [and we're still only at Level 1!]

Length heuristics

- **Shortest-clause heuristic**: Generate a clause with the fewest literals first
- **Unit resolution**: Prefer resolution steps in which at least one parent clause is a "unit clause," i.e., a clause containing a single literal
  - Not complete in general, but complete for Horn clause KBs

Unit resolution example

1. ¬Battery-OK ∨ ¬Bulbs-OK ∨ Headlights-Work
2. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
3. ¬Engine-Starts ∨ Flat-Tire ∨ Car-OK
4. Headlights-Work
5. Battery-OK
6. Starter-OK
7. ¬Empty-Gas-Tank
8. ¬Car-OK
9. ¬Flat-Tire

1,5 10. ¬Bulbs-OK ∨ Headlights-Work
2,5 11. ¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
2,6 12. ¬Battery-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
2,7 13. ¬Battery-OK ¬ Starter-OK ∨ Engine-Starts
3,8 14. ¬Engine-Starts ∨ Flat-Tire
3,9 15. ¬Engine-Starts ¬ Car-OK
16. ... [this doesn't seem to be headed anywhere either!]

Set of support

- At least one parent clause must be the negation of the goal *or* a "descendant" of such a goal clause (i.e., derived from a goal clause)
- *When there's a choice, take the most recent descendant*
- Complete, if all the clauses from the negation of the goal are included.
- Gives a goal-directed character to the search (e.g., like backward chaining)

Set of support example

1. ¬Battery-OK ∨ ¬Bulbs-OK ∨ Headlights-Work
2. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
3. ¬Engine-Starts ∨ Flat-Tire ∨ Car-OK
4. Headlights-Work
5. Battery-OK
6. Starter-OK
7. ¬Empty-Gas-Tank
8. ¬Car-OK
9. ¬Flat-Tire

9,3 10. ¬Engine-Starts ∨ Car-OK
10,2 11. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Car-OK
10,8 12. ¬Engine-Starts
11,5 13. ¬Starter-OK ∨ Empty-Gas-Tank ∨ Car-OK
11,6 14. ¬Battery-OK ∨ Empty-Gas-Tank ∨ Car-OK
11,7 15. ¬Battery-OK ∨ ¬Starter-OK ∨ Car-OK
16. ... [a bit more focused, but we still seem to be wandering]

Unit resolution + set of support example

1. ¬Battery-OK ∨ ¬Bulbs-OK ∨ Headlights-Work
2. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
3. ¬Engine-Starts ∨ Flat-Tire ∨ Car-OK
4. Headlights-Work
5. Battery-OK
6. Starter-OK
7. ¬Empty-Gas-Tank
8. ¬Car-OK
9. ¬Flat-Tire

9,3 10. ¬Engine-Starts ∨ Car-OK
10,8 11. ¬Engine-Starts
11,2 12. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank
12,5 13. ¬Starter-OK ∨ Empty-Gas-Tank
13,6 14. Empty-Gas-Tank
14,7 15. FALSE

[Hooray! Now that's more like it!]

Simplification heuristics

- **Subsumption:**
  Eliminate sentences that are subsumed by (more specific than) an existing sentence to keep KB small
  - If P(x) is already in the KB, adding P(A) makes no sense – P(x) is a superset of P(A)
  - Likewise adding P(A) ∨ Q(B) would add nothing to the KB
- **Tautology:**
  Remove any clause containing two complementary literals (tautology)
- **Pure symbol:**
  If a symbol always appears with the same "sign," remove all the clauses that contain it

Example (Pure Symbol)

1. ~~¬Battery-OK ∨ ¬Bulbs-OK ∨ Headlights-Work~~
2. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
3. ¬Engine-Starts ∨ Flat-Tire ∨ Car-OK
4. ~~Headlights-Work~~
5. Battery-OK
6. Starter-OK
7. ¬Empty-Gas-Tank
8. ¬Car-OK
9. ¬Flat-Tire

Input resolution

- At least one parent must be one of the input sentences (i.e., either a sentence in the original KB or the negation of the goal)
- Not complete in general, but complete for Horn clause KBs
- Linear resolution
  - Extension of input resolution
  - One of the parent sentences must be an input sentence *or* an ancestor of the other sentence
  - Complete

Ordered resolution

- Search for resolvable sentences in order (left to right)
- This is how Prolog operates
- Resolve the first element in the sentence first
- This forces the user to define what is important in generating the "code"
- The way the sentences are written controls the resolution

Prolog: logic programming language based on Horn clauses

- Resolution refutation: goal-directed and depth-first
  - always start from the goal clause
  - always use new resolvent as one of parent clauses for resolution
  - backtracking when the current thread fails
  - complete for Horn clause KB
- Supports answer extraction (can request single or all answers)
- Orders clauses & literals within a clause to resolve non-determinism
  - Q(a) may match both Q(x) <= P(x) and Q(y) <= R(y)
  - A (sub)goal clause may contain >1 literals, i.e., <= P1(a), P2(a)
- Use "closed world" assumption (negation as failure)
  - If it fails to derive P(a), then assume ~P(a)

Summary

- Logical agents apply inference to a KB to derive new information and make decisions
- Basic concepts of logic:
  - Syntax: formal structure of sentences
  - Semantics: truth of sentences wrt models
  - Entailment: necessary truth of one sentence given another
  - Inference: deriving sentences from other sentences
  - Soundness: derivations produce only entailed sentences
  - Completeness: derivations can produce all entailed sentences
- FC and BC are linear time, complete for Horn clauses
- Resolution is a sound and complete inference method for propositional and first-order logic