**Artificial Intelligence**

**Session 8**

1. Classification of Logic-based technologies:
   1. logic is the main means to endow machines with human-like reasoning
      1. Knowledge representation → Expert systems
      2. Reasoning → Intelligent systems
2. Knowledge engineering
   1. Role of Knowledge Engineer is to
      1. elicit or otherwise ascertain knowledge
      2. represent it in most appropriate way
      3. use it to derive previously unknown facts
         1. follow chain of reasoning from new data to a conclusion (e.g. medical diagnosis)
         2. make explicit things that were previously implicit in system too complex for human to understand all at once
   2. One way to represent knowledge is using logic
      1. Based on simple (atomic) logic statements
3. Forms of atomic sentences
   1. Most atomic sentences have one of the following forms:
      1. Statement
      2. e.g. Socrates-is-a-man
   2. Property(Object)
      1. e.g. Man(Socrates), Dead(Socrates),
      2. Perhaps clearer if written IsMan(Socrates), IsDead(Socrates)
   3. Relation(Object1, Object2, …)
      1. e.g. Occupation(Socrates, Philosopher), Mother(Elizabeth, Charles), LessThan(2, 5)
      2. Convention is that Object1 would be subject of the sentence if expressed in English (Socrates has occupation philosopher; Elizabeth is the mother of Charles; 2 is less than 5)
   4. In each case, they are sentences, i.e. they say something
      1. What the sentence says might be true or false
4. Knowledge engineering
   1. Often, in one formalism or another, this involves maintaining database of facts known to be true and rules that can apply to them
   2. Problem formulation in real-world situations is often very difficult
      1. different experts have different opinions
      2. world is continuous and unpredictable
      3. clients don’t really know what they want from you
   3. Example: the Blocks World:
      1. An approach to understanding KE issues is to use simplified “blocks” world, then generalise with experience
      2. Example
         1. There is/are
            1. a table
            2. some distinguishable blocks
            3. a robot hand/arm
         2. Problems specified by initial and desired states
         3. Express solutions as sequence of actions by arm
         4. Predicates
            1. On(x,y), On-table(x)
            2. Clear(x)
            3. Empty-table
5. Knowledge representation & inference
   1. KR should allow us to express facts or beliefs using formal language
      1. expressively and unambiguously
   2. Inference procedure should allow us to determine automatically what follows from these facts
      1. correctly (soundly) and completely (and tractably)
   3. Example:
      1. Be able to express formally that:
         1. “The red block is above the blue block”
         2. “The green block is above the red block”
      2. Be able to infer:
         1. “The green block is above the blue block”
         2. “The blocks form a tower”
6. Components of a logical calculus
   1. Formal language
      1. words and syntactic rules that tell us how to build up sentences
         1. so we can build up more complex statements from simple ones
      2. semantic mappings that tell us what the words mean
   2. Inference procedure which allows us to compute which sentences are valid inferences from other sentences
   3. There are many different logical calculi; we will study
      1. Propositional Calculus
      2. First Order Predicate Calculus
7. Example
   1. Given
      1. If it rains in the morning, then I wear my black coat
      2. If I wear my black coat, then I wear my black shoes
      3. I am not wearing black shoes
   2. Find out
      1. Was it raining this morning?
   3. Human reasoning:
      1. Brown shoes, so no black coat, so it was not raining this morning
         1. We want a computer to do that, reliably and in general
8. The Propositional Calculus (PC)
   1. Each symbol in Propositional Calculus is either:
      1. proposition: a basic, smallest unit of meaning in the calculus
         1. e.g. “It-is-raining”
      2. connective: for combining propositions into more complex sentences
   2. Two reserved, special propositions
      1. True and False
         1. with the obvious meanings!
   3. Convention: propositions begin with upper case letters
      1. P, Q, Sunny, etc.
   4. Connectives use special symbols
      1. ∧ (and [Inverted V]) , ∨ (or) , ¬ (not), → (implies), ≡ (is equivalent to)
9. Sentences (syntax) in PC
   1. Sentence is syntactic unit to which truth values can be attached
      1. called Well-Formed Formulae
   2. Every propositional symbol is a sentence. True, False, P
   3. Negation of a sentence is a sentence. ¬P, ¬False.
   4. Conjunction (and) of two sentences is sentence. P ∧ Q
   5. Disjunction (or) of two sentences is sentence. P ∨ Q
   6. Implication of one sentence by another is sentence. P → Q
      1. Can also be expressed as ¬P ∨ Q
   7. Equivalence of two sentences is a sentence. P ≡ Q
      1. Can also be expressed as (P → Q) ∧ (Q → P)
      2. ≡ is therefore sometimes omitted from the propositional calculus
10. Semantics (meaning) in PC
    1. Interpretation of set of sentences is assignment of truth value T or F to each propositional symbol (and so to each sentence)
       1. proposition True is always assigned truth value T
       2. proposition False is always assigned truth value F
       3. assignment of negation, ¬P is F iff (if and only if) assignment of P is T
       4. assignment of conjunction, P ∧ Q is T iff both P and Q are assigned T
       5. assignment of disjunction, P ∨ Q is F iff both P and Q are assigned F
       6. assignment of implication, P → Q is F iff assignment of P is T and assignment of Q is F
       7. assignment of equivalence, P ≡ Q is T iff assignments of P and Q are same
11. Properties of logical connectives
    1. commutativity
       1. P ∨ Q ≡ Q ∨ P
       2. P ∧ Q ≡ Q ∧ P
    2. associativity
       1. ( P ∨ Q ) ∨ R ≡ P ∨ ( Q ∨ R )
       2. ( P ∧ Q ) ∧ R ≡ P ∧ ( Q ∧ R )
    3. distributivity
       1. P ∨ ( Q ∧ R ) ≡ ( P ∨ Q ) ∧ ( P ∨ R )
       2. P ∧ ( Q ∨ R ) ≡ ( P ∧ Q ) ∨ ( P ∧ R )
12. Some useful laws and equivalences
    1. excluded middle: P ∨ ¬ P
    2. double negation: ¬ ¬ P ≡ P
    3. contrapositive: P → Q ≡ ¬ Q → ¬ P
    4. de Morgan’s laws
       1. ¬ ( P ∨ Q ) ≡ ¬ P ∧ ¬ Q
       2. ¬ ( P ∧ Q ) ≡ ¬ P ∨ ¬ Q
    5. Note order of operator precedence
       1. ¬ precedes ∧ precedes ∨
       2. → are ≡ are complicated: use parentheses
       3. Compare with arithmetic operators: –, x, +
13. Truth tables
    1. A truth table has all sentences along its top, usually in increasing order of syntactic complexity
       1. rows are all possible interpretations, one row each
       2. how many rows are needed in general?

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **P** | **Q** | **¬P** | **P ∧ Q** | **P V Q** | **P 🡪 Q** |
| T | T | F | T | T | T |
| T | F | F | F | T | F |
| F | T | T | F | T | T |
| F | F | T | F | F | T |

1. We can prove things using truth tables
   1. ¬P ∨ Q ≡ P → Q

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **P** | **Q** | **¬P** | **P ∧ Q** | **P V Q** | **P 🡪 Q** |
| T | T | F | T | T | T |
| T | F | F | F | F | T |
| F | T | T | T | T | T |
| F | F | T | T | T | T |

1. Proofs in propositional calculus
   1. Problem Description
      1. If it rains in the morning, then I wear my black coat
      2. If I wear my black coat, then I wear my black shoes
      3. I am not wearing black shoes
      4. Did it rain this morning?
   2. Propositions
      1. P: It rained this morning.
      2. Q: I am wearing my black coat.
      3. R: I am wearing black shoes.
   3. Premises
      1. P→Q
      2. Q→R
      3. ¬R
      4. Question: P? (Given that Premises are true, is P true?) [LAST ROW]

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Propositions** | | | **Premises** | | | **Trail conclusions** | |
| **P** | **Q** | **R** | **P 🡪 Q** | **Q 🡪 P** | **¬R** | **P** | **¬P** |
| T | T | T | T | T | F | T | F |
| T | T | F | T | F | T | T | F |
| T | F | T | F | T | F | T | F |
| T | F | F | F | T | T | T | F |
| F | T | T | T | T | F | F | T |
| F | T | F | T | F | T | F | T |
| F | F | T | T | T | F | F | T |
| F | F | F | T | T | T | F | T |

When all premises are true, P is false, so it did not rain this morning

1. Proof procedures:
   1. Proof procedure consists of
      1. set of inference rules
      2. algorithm for applying inference rules
         1. usually, we start from the thing we want to prove
         2. then work “backwards” towards things we already know, such as axioms and theorems
   2. Semantics of logical entailment
      1. A sentence, S, logically follows from, or is entailed by, set E, of sentences if and only if every interpretation and variable assignment that satisfies E also satisfies S.
2. Properties of inference rules
   1. Soundness
      1. Set of inference rules is sound iff every sentence it infers from a set, E, of sentences logically follows from E
   2. Completeness
      1. Set of inference rules is complete iff it can infer every expression that logically follows from a set of sentences
3. Inference rules
   1. Modus Ponens (implication elimination)
      1. if we know that P implies Q, and P is true, then infer Q
      2. ( P ∧ ( P → Q )) → Q
   2. Modus Tollens
      1. given that P implies Q, and Q is false, infer ¬P
      2. ( ¬Q ∧ ( P → Q )) → ¬ P
   3. Hypothetical syllogism
      1. (( P → Q) ∧ ( Q→R)) → P →R
   4. Disjunctive syllogism
      1. ((P∨Q) ∧ ¬P) → Q
   5. Conjunction (And) elimination
      1. P is true and Q is true if P ∧ Q is true
   6. Conjunction (And) introduction
      1. P ∧ Q is true if P is true and Q is true
   7. Addition
   8. Resolution
      1. This plays an important role in Prolog
4. The Problem
   1. Given knowledge base, KB (a set of sentences) and interpretation, I
   2. Prove sentence, S (under same interpretation, I)
   3. Formally
      1. Show that KB yields S
         1. KB entails S
         2. S follows from KB
   4. Reasoning: form long chains of inference to prove a sentence
      1. forwards from what we know to what we want to prove
      2. backwards from what we want to prove to what we know
         1. backwards is often more efficient: no search branches leading off-topic
5. Theorem Proving
   1. Theorem proving process involves choosing & applying such rules until desired sentence is shown to be entailed
   2. Called proof because rules known, a priori, to be sound (i.e. correct)
   3. Choice of rule is hard, because you can’t know whether chosen rule will help lead toward solution, in a long proof
      1. e.g., Modus Ponens is incomplete
      2. so each time we use it, we also have to consider other possibilities
      3. so each time we use it, we create a set of alternative choices
6. Modus Ponens is incomplete
   1. Consider these rules:
      1. If it is raining (R), I will carry an umbrella (U)
      2. If it is not raining (¬R), I will carry an umbrella (U)
   2. Easy to conclude (as a human) that I always carry an umbrella
      1. { R → U, ¬R → U } yields { U }
      2. but this isn’t provable using modus ponens alone
         1. we’d need the law of excluded middle: R ∨ ¬R
7. Resolution refutation
   1. To prove sentence S, the following rule can be used:
      1. add negation of S to the KB
      2. see if this leads to a contradiction
   2. This applies the law of the excluded middle (S ∨ ¬S)
      1. if ¬S is inconsistent with KB, then KB yields S
   3. Known as resolution refutation
      1. A form of proof by contradiction
      2. Basis of “Logic Programming” language, Prolog
   4. Example Proof:
      1. Hypothesis:
         1. ¬p ∧ q, r 🡪 p, ¬r 🡪 ¬s, s 🡪 t
      2. Conclusion:
         1. T
   5. Example Question
      1. Use a truth table to verify that ((A ∧ B) → C) ≡ (¬A ∨ ¬B ∨ C)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **A** ∧ **B** | **(A** ∧ **B) 🡪 C** | **¬A** | **¬B** | **¬AV ¬ BVC** | **LHS = RHS** |
| T | T | T | T | T | F | F | T | T |
| T | T | F | T | F | F | F | F | T |
| T | F | T | F | T | F | F | T | T |
| T | F | F | F | T | F | F | T | T |
| F | T | T | F | T | T | T | T | T |
| F | T | F | F | T | T | T | T | T |
| F | F | T | F | T | T | T | T | T |
| F | F | F | F | T | T | T | T | T |

1. First Order Predicate Calculus
   1. Propositional Calculus is not very expressive
   2. Only has propositions
      1. can’t make statements about all of a thing
      2. or things that don’t exist
      3. or whether things exist
   3. In “rains/coat/shoes” example, we omit day on which we checked the premises
   4. How can we make statements for doing something each day?
      1. If it rains on Monday morning ...
      2. If it rains on Tuesday morning ... etc.
2. First Order Predicate Calculus (FOPC) has Propositional Calculus properties and more
   1. constant symbols: stand for objects, things which sentences are about; written like propositions, but occur in different positions
   2. variables: usually lower case single letters, ranging over objects
   3. predicate symbols: describe relationships between (and properties of) objects; written like propositions with arguments
   4. function symbols: map between objects; written like predicates
   5. existential quantifier ∃ [INVERTED E]: for some; before variable & sentence
   6. universal quantifier ∀ [INVERETED A]: for all; before variable & sentence
3. Rain example in FOPC
   1. Problem description
      1. If it rains in the morning [on particular day], then I wear my black coat [on that day].
      2. If I wear my black coat [on particular day], then I wear black shoes [on that day].
      3. I am not wearing black shoes [today].
      4. Did it rain in the morning [today]?
   2. Premises:
      1. ∀d Rains(d) → BlackCoat(d)
      2. ∀d BlackCoat(d) → BlackShoes(d)
      3. ¬BlackShoes(Tuesday)
   3. Question: Rains(Tuesday)?
   4. Note quantifiers have lowest precedence
      1. so ∀x P → Q means ∀x (P → Q), not (∀x P) → Q
      2. if in doubt, use parentheses
4. Function and quantifier examples
   1. Function maps its arguments to fixed single value
      1. do not have truth values: map between objects
      2. denoted in same way as predicates
         1. you can tell which is which from where they appear: predicates are outermost
      3. have arity: number of arguments they take
   2. A person’s mother is that person’s parent
      1. ∀x Person( x ) → Parent( Mother-of( x ), x )
      2. A person can only have one mother, so using a function like this is OK
   3. All computers have a mouse connected by USB
      1. ∀x Computer( x ) → ∃y Mouse( y ) ∧ USB-Connection( x, y )
   4. There is at least one person in this class who thinks
   5. ∃x Person( x ) ∧ Registered( x, AIClass ) ∧ Thinks( x )
5. Syntax of FOPC
   1. Useful extra operator is =
      1. TEST for equality, not assignment statement
      2. like == in Java, Javascript, C, C++
      3. Example: ∀x y z . x = y ∧ y = z → x = z
   2. Terms: correspond with things in the world (like nouns in grammar)
      1. Constants: Thursday, Socrates, 25
      2. Variables: x
      3. Function expressions
         1. Function symbol of arity n followed by n terms, enclosed in (), separated by ,
         2. Function(var, AnotherFunction(Thing))
   3. Sentences: statements that can be true or false
      1. Atomic Sentence
         1. Predicate symbol of arity n followed by n terms, enclosed in (), separated by ,
      2. Result of applying connective to one or more sentences
      3. Result of applying quantifier (∀, ∃) with its variable to a sentence
6. Semantics of FOPC: Interpretation
   1. Let domain D be nonempty set of objects, which may related in various ways
      1. n-ary relation is set of n-tuples of elements of D (i.e. those n-tuples for which relation holds)
         1. unary relations represent properties of objects
   2. n-ary function is relation between n-tuples and objects in D, which maps each n-tuple to exactly one object
      1. interpretation over D is assignment of entities in D to each constant, variable, predicate and function symbols of an expression
      2. Each constant is assigned an element of D
      3. Each variable is assigned to nonempty subset of D (allowable substitutions)
      4. Each function of arity m is defined (D^m 🡪 D) ‣ Each predicate of arity n is defined (D^n 🡪 {T,F}).
7. Rain example in FOPC revisited
   1. Problem description
      1. If it rains in the morning [on a particular day], then I wear my black coat [on that day]
      2. If I wear my black coat [on a particular day], then I wear my black shoes [on that day]
      3. I am not wearing my black shoes [today].
      4. Did it rain in the morning [today]?
   2. Premises:
      1. ∀d Rains-in-morning(d) → Black-coat(d)
      2. ∀d Black-coat(d) → Black-shoes(d)
      3. ¬Black-shoes(Tuesday)
   3. Question: Rains-in-morning(Tuesday)?
   4. Proof is complicated: which inference rule to use next?
   5. A simpler approach is better:
      1. Resolution Theorem Proving
8. Past Example Question
   1. Using following predicates and their natural language meanings:
      1. cat(x): x is a cat
      2. dog(x): x is a dog
      3. owns(x, y): x owns y
      4. grey(x): x is grey
   2. express following sentences in first order logic:
      1. (i) John has a cat.
      2. (ii) Dogs are never grey.
      3. (iii) All of John's cats are grey.
      4. (iv) No dog owner owns any cats.
9. Clausal Form
   1. Resolution is single, simple, sound, complete rule
   2. Used in automated theorem proving, logic programming, circuit analysis…
   3. First get sentences in conjunctive (or clausal) normal form (CNF),
   4. AND of ORs
      1. RIGHT:
         1. ¬A ∨ B
         2. ¬B ∧ ¬C
         3. (A ∨ C) ∧ (B ∨ C)
         4. A ∧ (B ∨ D) ∧ (B ∨ E)
         5. ¬P(A)
      2. WRONG:
         1. A → B
         2. ¬(B ∨ C)
         3. (A ∧ B) ∨ C
         4. A ∧ (B ∨ (D ∧ E))
         5. ¬∀x.P(x)
10. Conjunctive Normal Form in FOPC
    1. Rewrite →: A → B ⇒ ¬A ∨ B
    2. Minimise negations using logical definitions
       1. ¬∃x.A(x) ⇒ ∀x. ¬A(x)
       2. ¬∀x.A(x) ⇒ ∃x.¬A(x)
       3. ¬(A ∨ B) ⇒ ¬A ∧ ¬B (De Morgan’s laws)
       4. ¬(A ∧ B) ⇒ ¬A ∨ ¬B (De Morgan’s laws)
       5. Rewrite any double negations: ¬¬A ⇒ A
    3. Standardise variables apart
       1. rename variables so each quantifier associated with different name
          1. (Ax)P(x) ⇒ (Ay)P(y) if x already used
       2. Skolemise (make up a name) for all existential quantifiers
          1. ∃x.P(x) ⇒ P(A) where A is arbitrary object
             1. Use different arbitrary object for each quantifier
          2. ∀x.∃y.P(x, y) ⇒ ∀x.P(x,F(x))
             1. Replace existential variable in universal variable with function
       3. Drop universal quantifiers
          1. At this point, all variables universally quantified
       4. Convert sentence to conjunctive normal form
          1. conjunction of disjunctions of atomic sentences
          2. AND of ORs
       5. Split top-level conjunction up, to make a set of disjunctions
       6. Standardise variables apart again, w.r.t. clauses
          1. so there is no overlap between variables in different clauses
       7. Use resulting set of clauses as Knowledge Base in a resolution proof
11. First Order Term Unification
    1. Unification procedure for assigning values to variables
       1. Related to ∀ instantiation in standard FOPC
       2. using relevant constants and functions to instantiate the variables
    2. Resolve two clauses if they contain complementary literals which share no variables
       1. Animal (g(x) V Loves (f(x), x) and ￢ Loves(a, b) V ￢Hunts(a, b)
       2. unified to Animal (g(x) V ￢ Hunts(f(x), x)
       3. with unifier (or substitution set) of {a/f(x), b/x}
       4. notation: a/x means “a replaces x”
    3. To unify two terms (or literals):
       1. if either is a variable, set it identical to the other, add resulting pair to unifier; otherwise...
       2. compare their functors (outermost predicate or function symbol); if they do not match, then fail; otherwise...
       3. for each pair of respective arguments, unify arguments using this procedure, and combine unifiers for each argument pair.
    4. Resulting list of substitutions is called Most General Unifier (MGU)
    5. In 1, expression substituted for variable cannot contain variable it substitutes (e.g. P(x) and P(f(y,x)) do not unify, since substitution f(y,x)/x not allowed)
    6. Notation
       1. Sometimes write Term followed by Unifier to mean “Result of applying this unifier to this term”
          1. P( x, y ) { A/x, B/y } evaluates to P( A, B )
    7. Successive application of unifiers
       1. Can write Term Unifier1 Unifier2 to mean “Result of applying these unifiers, one at a time, to Term”
          1. P( x, y ) { A/x }{ B/y } evaluates to P( A, B )
    8. Composition of unifiers
       1. Can combine unifiers, so long as there are no contradictory assignments
          1. { A/x } { B/y } combine to give { A/x, B/y }
          2. { A/x } { B/x } do not combine, because x would take 2 different values at once
    9. Unification examples
       1. P( x, y ) unified with P( A, B ) gives P( A, B ) unifier { A/x, B/y }
       2. P( x, y ) unified with Q( A, B ) gives no unifier ( P != Q )
       3. P( F(x) ) unified with P( F( A ) ) gives P( F( A )) unifier { A/x }
       4. P( F(x), x, u, u ) unified with P( F(y), z, z, A ) gives P( F( A ), A, A, A ) unifier { x/y, x/z, x/u, A/x }
          1. we don’t need to write down all the different permutations
             1. this is enough to say that they’re all the same
12. Resolution example
    1. Some rules
       1. All people who are graduating are happy.
       2. All happy people smile.
       3. Jane is graduating.
    2. and a question
       1. Is Jane smiling?
    3. First convert to predicate logic
       1. Premise ∀x.(Graduating(x)→Happy(x))∧ ∀x.(Happy(x)→Smiling(x))∧ Graduating(Jane)
       2. Goal
          1. Smiling( Jane )
13. ∀x.(Graduating(x)→Happy(x))∧∀x.(Happy(x)→Smiling(x))∧Graduating(Jane)
    1. Rewrite → •
       1. ∀x.(¬Graduating(x) ∨ Happy(x)) ∧ ∀x.(¬Happy(x) ∨ Smiling(x)) ∧ Graduating( Jane )
    2. Reduce scope of negations: all minimal, nothing to do
    3. Rewrite double negations: no double negations, nothing to do
    4. Standardise variables apart
       1. ∀x.(¬Graduating(x) ∨ Happy(x)) ∧ ∀y.(¬Happy(y) ∨ Smiling(y)) ∧ Graduating( Jane ) 5. Skolemise existentials: no existentials, so nothing to do
    5. Drop universal quantifiers
       1. ▮ (¬Graduating(x) ∨ Happy(x)) ∧ ▮(¬Happy(y) ∨ Smiling(y)) ∧ Graduating( Jane )
    6. Convert to CNF: already in CNF
    7. Separate into disjunctive clauses
       1. ¬Graduating(x) ∨ Happy(x)
       2. ¬Happy(y) ∨ Smiling(y)
       3. Graduating( Jane )
    8. Standardise variables apart between clauses: already named apart
    9. Now, only rules we need are
       1. unification
       2. factoring: unifying literals of the same polarity within a clause   
          Resolution
    10. Example: Proof by contradiction
14. Example with existential quantification
    1. Some rules
       1. All graduating people are happy. All happy people smile. Someone is graduating.
    2. and a question
       1. Is anyone smiling?
    3. First convert to predicate logic
       1. Premise: ∀x.(Graduating(x) → Happy(x)) ∧ ∀x.(Happy(x) → Smiling(x)) ∧ ∃x.Graduating(x)
       2. Goal: ∃x.Smiling( x )
    4. Use shorter notation
       1. Premise: ∀x.(G(x) → H(x)) ∧ ∀x.(H(x) → S(x)) ∧ ∃x.G(x)
       2. Goal: ∃x.S( x )
15. Example with existential quantification:
    1. ∀x.(G(x) → H(x)) ∧ ∀x.(H(x) → S(x)) ∧ ∃x.G(x) ∧ ¬∃x.S(x)
    2. ∀x.(¬G(x) ∨ H(x)) ∧ ∀x.(¬H(x) ∨ S(x)) ∧ ∃x.G(x) ∧ ¬∃x.S(x)
    3. ∀x.(¬G(x) ∨ H(x)) ∧ ∀x.(¬H(x) ∨ S(x)) ∧ ∃x.G(x) ∧ ∀x.¬S(x)
    4. ∀x.(¬G(x) ∨ H(x)) ∧ ∀y.(¬H(y) ∨ S(y)) ∧ ∃z.G(z) ∧ ∀u.¬S(u)
    5. ∀x.(¬G(x) ∨ H(x)) ∧ ∀y.(¬H(y) ∨ S(y)) ∧ G( p ) ∧ ∀u.¬S(u)
    6. (¬G(x) ∨ H(x)) ∧ (¬H(y) ∨ S(y)) ∧ G( p ) ∧ ¬S(u)
    7. { ¬G(x) ∨ H(x), ¬H(y) ∨ S(y), G( p ), ¬S(u) }
    8. Same as with constant “Jane”, but variable u gets passed around instead
16. Example with universal quantification
    1. Some rules & a question
       1. All graduating people are happy. All happy people smile. Everyone is graduating.
       2. Is everyone smiling?
    2. Convert to predicate logic
       1. Premise: ∀x.(G(x) → H(x)) ∧ ∀x.(H(x) → S(x)) ∧ ∀x.G(x)
       2. Goal: ∀x.S(x)
    3. Conversion to CNF
       1. ∀x.(Graduating(x) → Happy(x)) ∧ ∀x.(Happy(x) → S(x)) ∧ ∀x.Graduating(x) ∧ ¬∀x.S(x)
       2. ∀x.(¬Graduating(x) ∨ Happy(x)) ∧ ∀x.(¬Happy(x) ∨ S(x)) ∧ ∀x.Graduating(x) ∧ ¬∀x.S(x)
       3. ∀x.(¬Graduating(x) ∨ Happy(x)) ∧ ∀x.(¬Happy(x) ∨ S(x)) ∧ ∀x.Graduating(x) ∧ ∃x.¬S(x)
       4. ∀x.(¬Graduating(x) ∨ Happy(x)) ∧ ∀y.(¬Happy(y) ∨ S(y)) ∧ ∀z.Graduating(z) ∧ ∃u.¬S(u)
       5. ∀x.(¬Graduating(x) ∨ Happy(x)) ∧ ∀y.(¬Happy(y) ∨ S(y)) ∧ ∀z. Graduating(z) ∧ ¬S( p )
       6. (¬Graduating(x) ∨ Happy(x)) ∧ (¬Happy(y) ∨ S(y)) ∧ Graduating(z) ∧ ¬S( p )
       7. { ¬Graduating(x) ∨ Happy(x), ¬Happy(y) ∨ S(y), Graduating(z), ¬S( p ) }
17. Slightly more realistic example
    1. In this example, not all quantifiers match up neatly
       1. Some rules and a question
          1. All people who are graduating are happy. All happy people smile. Someone is graduating.
          2. Is everyone smiling?
    2. First convert to predicate logic
       1. ∀x.(G(x) → H(x)) ∧ ∀x.(H(x) → S(x)) ∧ ∃x.G(x)
       2. ∀x.S(x)
    3. Resolution-ready form
       1. { ¬G(x) ∨ H(x), ¬H(y) ∨ S(y), G( Z ), ¬S( U ) }
    4. Can’t infer everyone is smiling from knowledge that one person is graduating
    5. The other way round
       1. In this example, not all quantifiers match up neatly
       2. Some rules and a question
          1. All people who are graduating are happy. All happy people smile. Everyone is graduating.
          2. Is anyone smiling?
       3. First convert to predicate logic
          1. ∀x.(G(x)→H(x)) ∧ ∀x.(H(x)→S(x)) ∧ ∀x.G(x) ∃x.S( x )
       4. Resolution-ready form
          1. { ¬G(x) ∨ H(x), ¬H(y) ∨ S(y), G(z), ¬S(u) }
       5. Can infer one person is smiling from knowledge that everyone is graduating
18. Summary
    1. Resolution (with unification and factoring) is simple, powerful rule to perform inference in logical databases
    2. It is sound and complete for propositional calculus
    3. It is sound and “refutation complete” for first order predicate calculus
       1. set of sentences is unsatisfiable if and only if there exists a derivation of a contradiction (the empty clause)
       2. if set of sentences is satisfiable, computation might not terminate
19. Past Example Questions
    1. Express the following first order formula in clausal form: isNumber(0) ∧ ∀x isNumber(x) → ∃y Greater(y, x)
    2. Using the predicate S(x, y) meaning “x shaves y”, express the following sentence in first order logic:   
         
       There is a person who shaves only every person who does not shave themselves.   
         
       Then convert logical expression into clausal form and use resolution and factoring to show that the sentence is inconsistent. (Note: factoring reduces two identical literals within a clause to one; if two literals in a clause are unifiable, their most general unifier is applied to the whole clause and then the duplicate literal is deleted.)