CSC384: Lecture 3

- Last time
  - DCL: syntax, semantics, proofs
  - bottom-up proof procedure
- Today
  - top-down proof procedure (SLD-resolution)
  - perhaps start on uses of DCL
- Readings:
  - Today: 2.7, 2.8 (details in tutorial), perhaps Ch.3 (except 3.7); we'll discuss only part
  - Next week: wrap Ch.3; start on Ch.4: 4.1-4.4/4.6

Top-Down Proof Procedure

- BUPP is data-driven
  - not influenced by query q, just facts and rules in KB!
  - wasteful: proves things unneeded to prove q
- Top-down proof procedure is query-driven:
  - focussed on deriving a specific query
- We'll describe a TDPP called SLD-resolution
  - Basically, the strategy implemented within Prolog
  - stands for selected linear, definite-clause resolution

SLD-Resolution (No vars)

- Basic intuitions:
  - suppose we have query \(?q_1 \& q_2\)
  - suppose we have rule \(q_1 \leftarrow a \& b \& c\).
  - if we prove subgoal query \(?a \& b \& c \& q_2\) then we know that original query must be true
- SLD a form of backchaining or subgoaling:
  - to prove \(q\), we look for a rule with the head \(q\), and then attempt to prove the body of that rule; if proven, we know \(q\) must be a consequence of KB
  - Progress: when subgoals are facts!
- Defn: An answer clause: \(\text{yes} \leftarrow q_1 \& \ldots \& q_m\)
- Defn: An answer: \(\text{yes} \leftarrow \ldots\)

SLD-Resolution: Algorithm (no vars)

Given query \(?q_1 \& \ldots \& q_m\) and a KB
1. Construct answer clause \(\text{yes} \leftarrow q_1 \& \ldots \& q_m\)
2. Until no KB-clause choosable or AC is an answer
   (a) Select an atom \(a_i\) from the current AC
      \(\text{yes} \leftarrow q_1 \& \ldots \& a_i \& \ldots \& q_m\)
   (b) Choose a clause \(a_i \leftarrow b_1 \& \ldots \& b_l\) from KB whose head matches selected atom
   (c) Replace \(a_i\) in AC with body to obtain new AC
      \(\text{yes} \leftarrow q_1 \& \ldots \& a_i \& \ldots \& b_1 \& \ldots \& b_l \& a_{i+1} \& \ldots \& q_m\)

SLD-Resolution

- If we reach an answer, return YES
  - query is a logical consequence of KB
- If we find no choosable clauses, return NO
  - query not a consequence (but not necessarily false)
- A sequence of answer clauses that culminates in an answer is an SLD-derivation of the query
- Our algorithm attempts to find a derivation:
  - If it chooses incorrectly at Step 2, it may fail
  - see text for distinction between choice and selection
  - we say derivation attempt fails if we get stuck
  - how does Prolog deal with failure?

SLD: Example

Derivation Attempt #1

KB: (1) \(a \leftarrow b \& c\).
   (2) \(b \leftarrow d \& e\).
   (3) \(b \leftarrow c\).
   (4) \(c \leftarrow e\).
   (5) \(d\).
   (6) \(e\).
   (7) \(f \leftarrow a \& g\).
Query: \(?a\)

Select \(a\): choose (1)
Select \(b\): choose (3)
Select \(c\): choose (6)
Select \(g\): FAIL no choosable clause
**SLD: Example**

**KB:**
1. \( a \leftarrow b \land c \).
2. \( b \leftarrow d \land e \).
3. \( b \leftarrow g \land e \).
4. \( c \leftarrow e \).
5. \( d \).
6. \( e \).
7. \( f \leftarrow a \land g \).

**Query:** \(?a\)

**Derivation Attempt #2**

- \( \text{yes} \leftarrow a \)
- \( \text{yes} \leftarrow b \land c \). Select a, choose (1)
- \( \text{yes} \leftarrow d \land e \). Select b, choose (2)
- \( \text{yes} \leftarrow e \land c \). Select c, choose (4)
- \( \text{yes} \leftarrow c \). Select e, choose (6)
- \( \text{yes} \leftarrow -. \)

**QUERY IS TRUE:** obtained answer

---

**SLD Notes**

- **Does atom selected to resolve away matter?**
  - No: all must be "proven" eventually
- **Does KB clause chosen to resolve with matter?**
  - Yes: wrong choice can lead to failure
  - We'll talk later about backtracking/search for a proof
- **Soundness:** should be fairly obvious
  - Exercise: prove that if any body in any answer clause is a consequence of KB, then so is query (soundness follows: if we derive an answer, query holds)
- **Completeness:** if KB \( \models q \) there is a derivation
  - can we find it? Yes, if we make correct choices
  - How? Might have to try all options (watch for cycles)

---

**Aside: Resolution**

\[
\begin{align*}
  a \lor b, \ & \neg b \lor c \\
  \hline
  a \lor c
\end{align*}
\]

**Resolution Proof Rule**

**Query**

\( \text{yes} \leftarrow g \land h \) equivalent to \( \neg g \lor h \lor \text{yes} \)

**Rule**

\( h \leftarrow a \land b \land c \) equivalent to \( \neg h \lor a \lor b \lor c \)

\[\neg g \lor h \lor \text{yes} \]

- \( \neg g \lor h \lor \text{yes} \)
- \( h \leftarrow a \land b \land c \)
  - equivalent to \( \text{yes} \leftarrow g \land a \land b \land c \)

**Resolvent**

\( -g \lor -a \lor b \lor c \lor \text{yes} \)

**equivalent to**

\( \text{yes} \leftarrow g \land a \land b \land c \)

---

**Variables in SLD (no functions)**

- **Recall query** \( q(X) \) is interpreted existentially:
  - is there some \( X \) s.t. \( q(X) \) is a consequence?
  - return a ground instance/term \( t \) (or all if \( t \) is empty):
    - with no functions, terms are just constants

**Example:**

- (1) \( \text{rich(joan)}. \)
- (2) \( \text{mother(linda,joan)}. \)
- (3) \( \text{mother(mary,linda)}. \)
- (4) \( \text{rich}(X) \leftarrow \text{mother}(X, Y) \land \text{rich}(Y). \)

**Query:**

\( \text{? rich(linda)}. \)

---

**SLD: Queries with no vars**

- **Query:** \( \text{?rich(linda)} \)
  - set up answer clause: \( \text{yes} \leftarrow \text{rich(linda)} \)
  - but body matches no heads in KB: How to start??

- **Intuitively**, \( \text{rich(linda)} \) **does** match the head of the rule \( \text{rich}(X) \leftarrow \text{mother}(X, Y) \land \text{rich}(Y). \)
  - just need to substitute constant \( \text{linda} \) for \( X \)
  - result: \( \text{yes} \leftarrow \text{mother(linda,Y)} \land \text{rich}(Y). \)

- **Applying constant substitution** \( X/\text{linda} \) to rule (4) gives us an **instance** of rule (4):
  - \( \text{rich(linda)} \leftarrow \text{mother(linda,Y)} \land \text{rich}(Y). \)
  - Note: this instance is clearly entailed by KB

---

**Example: SLD with vars in KB**

**KB:**

- (1) \( \text{rich(joan)}. \)
- (2) \( \text{mother(linda,joan)}. \)
- (3) \( \text{mother(mary,linda)}. \)
- (4) \( \text{rich}(X) \leftarrow \text{mother}(X, Y) \land \text{rich}(Y). \)

**Query:**

\( \text{? rich(linda)}. \)

**Derivation:**

- \( \text{yes} \leftarrow \text{rich(linda)}. \)
- \( \text{yes} \leftarrow \text{mother(linda,Y)} \land \text{rich}(Y). \)

**How Select rich(linda); resolve with (4) using (X/linda)**

- \( \text{yes} \leftarrow \text{rich(joan)}. \)
  - How Select \( \text{mother(linda,Y)} \); resolve with (2) using (Y/joan)
  - \( \text{yes} \leftarrow \text{rich(joan)}. \)
  - How Select \( \text{rich(joan)} \); resolve with (1) using {}
**Example: SLD with vars in query**

- **KB:**
  1. `rich(joan)`.  
  2. `mother(linda, joan)`.  
  3. `mother(mary, linda)`.  
  4. `rich(X) ← mother(X, Y) & rich(Y)`.

- **Query:**
  1. `? rich(Z)`.

**A Different Derivation:**

1. `yes(Z) ← rich(Z)`.  
2. `yes(joan) ← -`.
   - Select `rich(Z)`; resolve with (1) using (Z/joan)

**Example Derivation #1**

**KB**
- `busy(Z) ← teaches(Z, X) & teaches(Z, Y) & distinct(X, Y)`.
- `busy(Z) ← teaches(Z, 148)`.
- `teaches(craig, 2334)`.
- `teaches(kyros, 384)`.
- `teaches(suzanne, 148)`.
- `distinct(2334, 384)`.
- `distinct(2001, 384)`.

**Query**
- `busy(P)`.

**Answer Clause:**
- `yes(P) ← busy(P)`.

**Derivation:**
- `yes(P) ← busy(P)`.
- `yes(P) ← teaches(P, 148)`.
- Select `busy(P)`; resolve with (2) using (P/148)
- `yes(P) ← teaches(P, 148)`.
- Select `busy(P)`; resolve with (2) using (P/148)
- `yes(suzanne) ← -`.
- Select `distinct(2334, 384)`; resolve with (2) using (Z/2334)
- `yes(suzanne) ← -`.

**Example: SLD-Resolution: Algorithm (w/ vars)**

**Given query**
- `¬ X₁ ∧ ... ∧ ¬ Xₙ`  
  with variables `X₁, ..., Xₙ`

**1. Construct answer clause**
- `yes(X₁, ..., Xₙ) ← c₁ Æ ... Æ cₙ`.

**2. Until no KB-clause choosable or AC is an answer**

(a) Select an atom `a₁` from the current AC

(b) Choose a clause `hᵢ ← b₁ Æ ... Æ bₙ` from KB

and a substitution `σ` that unifies the head `hᵢ` of the KB clause with the selected atom `aᵢ` (i.e., that when applied to `hᵢ` and `aᵢ` makes them the same)

(c) apply `σ` to AC and KB clause to obtain `ACσ`, `KBσ`

(d) Replace `aᵢ` in `ACσ` with body of `KBσ` to obtain new AC

**SLD: Queries with vars**

- **Query:** `rich(Z)`
  - set up answer clause: `yes(Z) ← rich(Z)`
  - once derivation reaches an answer, this allows us to extract an "individual" for which query holds
  - can’t just say yes: must say "for who"
- Intuitively, `rich(Z)` does match the head of the rule `rich(X) ← mother(X, Y) & rich(Y)`
  - just need to substitute var `Z` for var `X`
  - result: `yes(Z) ← mother(Z, Y) & rich(Z)`.
- Applying substitution `X[Z]` to rule (4) gives:
  - `rich(Z) ← mother(Z, Y) & rich(Y)`.

**Example:**

**Example: SLD with vars in query**

- **KB:**
  1. `rich(joan)`.  
  2. `mother(linda, joan)`.  
  3. `mother(mary, linda)`.  
  4. `rich(X) ← mother(X, Y) & rich(Y)`.

- **Query:**
  1. `? rich(Z)`.

**Derivation:**
- `yes(Z) ← rich(Z)`.  
- `yes(Z) ← mother(Z, Y) & rich(Y)`.
  - Select `rich(Z)`; resolve with (1) using (Y/joan)
  - Select `mother(Z, joan)`; resolve with (2) using (Z/linda)
**Example Derivation #2**

**KB**
1. busy(Z) ← teach(Z, X) \& \& teach(Z, Y) \& \& distinct(X, Y).
2. busy(Z) ← teach(Z, X) \& \& teach(Z, Y).
3. teach(craig, 384).
4. teach(carrie, 384).
5. teach(kyros, 2534).
6. teach(kyros, 234).
7. teach(Sodore, 148).
8. distinct(2534, 384).
9. distinct(234, 384).
10. \& \& 48384, d(234, 2534), d(2534, 48). 2530/146.

**Problem lies in KB. We didn’t axiomatize domain correctly**
Add distinct(234, 384), etc.

**Example Derivation #3**

**Assume KB fixed with rule:** 12 distinct(C, D) ← distinct(D, C).

**Derivation**
yes(P) ← busy(P).
yes(P) ← P(X, Y) & P(Y, X) & d(X, y).
busy(P): \& \& (Z, P)
yes(craig) ← \& \& (craig, P) & d(384, Y)
t(P, X) (3) P(craig, X, 384)
yes(craig) ← \& \& d(384, 2534).
t(c, Y) \& \& (4) \& \& (X, 2534).

**Example Derivation #4**

**Substitutions**

- **Defn:** A substitution \( \sigma \) is any assignment of terms to variables:
  - we write it like as \( \sigma = \{ X \mapsto 1, Y \mapsto 2, \ldots \} \)
  - constant substitution is a special case; terms can be any terms (nonground included)
  - without functions, only terms are constants, vars
  - e.g. \( \sigma = \{ X/craig, Y/\text{father}(craig), Z/P, W/\text{father}(X) \} \)
- A substitution is applied to an expression by **uniformly and simultaneously** substituting each term for the corresponding variable
  - e.g. using subst. above on related(mother(X), W)
    gives related(mother(craig), father(X))

**Example**

**Substitutions**

- **Defn:** A substitution **unifies** two expressions \( e_1 \) and \( e_2 \) if \( e_1 \sigma \) is identical to \( e_2 \sigma \)
  - E.g., \( p(X, f(a)) \) and \( p(Y, f(Z)) \) are unified by:
    - \( \{ X/b, Y/b, Z/a \} \) gives \( p(b, f(a)) \) for both expressions
    - \( \{ X, Y/a \} \) gives \( p(y, f(a)) \) for both expressions
    - \( \{ X, Y/ Z, Z/a \} \) gives \( p(Z, f(a)) \) for both expressions
- **Unifier** \( \sigma \) is a **most general unifier** (MGU) of \( e_1 \) and \( e_2 \) if \( e_1\sigma \) is an instance of (unifies with) \( e_1\sigma \) for any other unifier \( \sigma' \)
  - An MGU gives the most general instance of an expression; any other unifier gives a result that would unify with that given by the MGU

**Example**

**MGUs: Examples**

- Let \( e_1 = \text{busy}(X) \), \( e_2 = \text{busy}(Y) \)
- **Unifier** \( \sigma_1 = \{ X/\text{kyros}, Y/\text{kyros} \} \)
  - result: \( e_1\sigma_1 = e_2\sigma_1 = \text{busy}(kyros) \)
- **Unifier** \( \sigma_2 = \{ X/\text{craig}, Y/\text{kyros} \} \)
  - result: \( e_1\sigma_2 = e_2\sigma_1 = \text{busy}(craig) \)
- **Unifier** \( \sigma_3 = \{ Y/X \} \)
  - result: \( e_1\sigma_3 = e_2\sigma_3 = \text{busy}(X) \)
- **Unifier** \( \sigma_4 \) an MGU of expressions; \( \sigma_4 = \{ X/\text{kyros} \} \)
  - \( e_1\sigma_1 = e_2\sigma_1 = \text{busy}(kyros) \)
  - \( e_1\sigma_2 = e_2\sigma_2 = \text{busy}(craig) \)

**Notes on General SLD Resolution**

- Generically, if you only use MGUs in SLD resolution to match a body atom with a KB head
  - ensures we don’t make too specific a choice and force us to failure unnecessarily
- To obtain all answers:
  - once we derive an answer, we pretend the derivation failed and backtrack to find other derivations
  - we only reconsider KB-clause choices, not atom selections, or unifier choice
Notes on General SLD Resolution

- Prolog (see Appendix B, Ch3.2, Ch3.3)
  - based on SLD-resolution
  - searches for derivations using a specific strategy: (a) always selects atoms from answer clause in left-to-right order; (b) always chooses KB clauses in top-to-bottom order (using first unifiable rule/fact)
  - records choices and tries alternatives if failure (essentially does depth-first search: why?)
  - provides a single answer for nonground queries; but you can force it to search for others (semicolon op)

Renaming of Variables: Example

KB:
1. rich(joan).
2. mother(linda, joan).
3. mother(mary, linda).
4. rich(X) ← mother(X,Y) & rich(Y).

Derivation:
- yes ← rich(mary).
- yes ← mother(mary,Y) & rich(Y).
- yes ← mother(mary,X) & mother(X,Y) & rich(X).
- rich(Y) : (4) using(Y/X)

Must fail: Nobody (in our KB) is their own mother!

Renaming of Variables

- When we add body of KB clause to answer clause, we may have accidental name conflicts
  - in example, Y in answer clause is not "same person" as Y in KB clause (yet both replaced by X)
- To prevent problems, we always rename vars in KB clause (uniformly) to prevent clashes
  - changing var names in KB clause cannot change meaning
- System: (a) each clause has diff. vars; (b) index KB vars, increase with each use of the clause
  - use rich(X) ← mother(X,Y) & rich(Y). i-th time you use this clause in a derivation

DCL: How can we use it?

- Query-answering system:
  - given KB representing a specific domain, use DCL (and suitable proof procedure) to answer questions
- A Deductive Database System
  - much like the above
- A Programming Language
  - Prolog (we’ve seen) is a dressed up DCL using SLD
  - Important to realize that as a programming language, we are still making logical assertions and proving logical consequences of these assertions

Prolog List Operations

- A distinguishing feature of Prolog is its built-in facilities for list manipulation
  - not hacks, but genuine logical assertions/derivations
- Consider the function cons, constant el:
  - cons accepts two args, returns pair containing them
    - e.g. cons(a,b), cons(a,cons(b,c))
  - el is a constant denoting the empty list
- A proper list is either el or a pair whose second element is a proper list
  - cons(a,cons(b,cons(c,el))) = [a,b,c]
Prolog List Operations

- Prolog uses a more suggestive notation:
  - [] is a constant symbol (empty list)
  - [[ ]] is a binary function symbol: infix notation (cons)
  - [a,b,c] shorthand for [a | b | c | []]
- But these are just terms in DCL
- Standard list manipulation operations correspond to logical assertions
  - e.g., the usual definition of append(X,Y,Z) simply defines what it means for Z to be the appending of X and Y

Defining Append

(A2) append([E1 | R1], Y, [E1 | Rest]) <-
    append(R1, Y, Rest).

Proving the Append Relation #1

Query: append([a,b], [c,d], [a,b,c,d]).

(A2) append([E1 | R1], Y, [E1 | Rest]) <-
    append(R1, Y, Rest).

Derivation:
yes <- append([a,b], [c,d], [a,b,c,d]).
yes <- append([b, c, d], [a, b, c, d]).
  Resolve with (A2) using { [E1 | b, R1 | [b, c, d]], Y/[c, d], Rest/[c, d] } 
  Resolve with (A2) using { [E1 | b, R1 | [b, c, d]], Y/[c, d], Rest/[c, d] } 
  Resolve with (A2) using { [E1 | b, R1 | [b, c, d]], Y/[c, d], Rest/[c, d] } 
  Resolve with (A1) using { Z/[c, d] } 
Answer: yes

Proving the Append Relation #2

Query: append([a,b], [c,d],[g,b,c,d]).

(A2) append([E1 | R1], Y, [E1 | Rest]) <-
    append(R1, Y, Rest).

Derivation:
yes <- append([a,b], [c,d],[g,b,c,d]).
  No append rule can unify with this atom
  (convince yourself: look at E1)
Answer: no

Proving the Append Relation #3

Query: append([L,M], [a,b,c,d]).

(A2) append([E1 | R1], Y, [E1 | Rest]) <-
    append(R1, Y, Rest).

Derivation:
yes(L,M) <- append([L,M],[a,b,c,d]).
yes([a|R], M) <- append([R], M, [b,c,d]).
  Resolve with (A2) using { [U|[a|R], Y/M, R1/[b,c,d] } 
  Resolve with (A2) using { [U|[a|R], Y/M, R1/[b,c,d] } 
  Resolve with (A2) using { [U|[a|R], Y/M, R1/[b,c,d] } 
  Resolve with (A1) using { Z/[b,c,d] } 
Answer: L = [a], M = [b,c,d]

Proving the Append Relation

- Exercise: Give derivations for at least two other answers for the previous query:
  - Query: append(L, M, [a,b,c,d]).
    * L = [], M = [a,b,c,d]
    * L = [a], M = [b,c,d]
    * L = [a,b], M = [c,d]
    * L = [a,b,c], M = [d]
    * L = [a,b,c,d], M = []
DCL and Knowledge Representation

- DCL has obvious uses as a question answering system for complex knowledge
  - A key issue: how does one effectively represent knowledge of a specific domain for this purpose?
  - Unfortunately, there are generally many ways to represent a KB: some more useful (compact, natural, efficient) than others
  - Let’s go through a detailed example to see where choices need to be made, what the difficulties are, etc.

The Herbalist Domain

- Suppose we want to build a KB that answers queries about what sorts of homeopathic remedies we need to treat different symptoms
  - This “expert system” will underly a Web site where users can ask for advice on herbal remedies
  - We need to build a KB that represents info we have about different clients, their symptoms, treatments, etc.

What Functionality is Needed?

- Before designing KB, we need to know what types of queries we’ll ask; do we want:
  a) \(?treatment(john, T)\).
  b) \(?treatment(symptom, T)\).
  c) \(?treatment(combination-of-symptoms, T)\).
  d) \(?safe(combination-of-treatments)\).
  e) \(?medical_records(john, R)\).
  f) \(?paid_bills(john)\).

- and so on

What Individuals Do We Need?

- Diseases: do we need diseases?
  - why? why not? (our treatment philosophy will be to apply treatments to symptoms: simplicity!)
- Combinations of symptoms? treatments?
- We’ll consider combinations:
  - symptomList is a list of symptoms:
    - e.g. function: symptomList(symptom, SList)
    - or using Prolog notation: [aches, fever, chills]
  - treatmentList similar:
    - \([tmt(mudwort,tincture), tmt(echinacea,capsule)]\)

What Relations?

- Relations depend on functionality desired
- If we ask \(?treatment(john, T)\), we need information about john in KB (e.g., symptoms)
  - e.g.: symptoms(john,fever), symptom(john,chills).
  - or: symptoms(john,[fever,chills]).
  - or maybe symptoms are relations themselves and not individuals: fever(john), chills(john).
- Maybe we don’t even discuss individual clients:
  - e.g., we only ask: \(?treatment(SList,TList)\).
- Different choices influence how you express your knowledge: some make life easy, or difficult!
Facts and Rules

- Once we’ve decided on suitable relations we need to populate our KB with suitable facts and rules
  - facts/rules should be correct
  - facts/rules should cover all relevant cases (which depends on the task at hand)
  - try to keep facts/rule concise (only relevant facts)
- For example: we can often express a zillion facts using one or two simple rules

Some Example Facts/Rules

- Facts about individual patients

  Specific Visit Facts (enter into KB during exam):
  - musclepain(mary,shoulders)
  - slow_digetion(john)
  - fever(john)

  Semi-permanent Facts (persist in KB):
  - arthritis(ming)
  - hypertension(john)
  - relaxed_disposition(mary)

- Rules relating treatments to symptoms

  We can relate treatments to symptoms directly:
  - remedy(X,chinacea) :- fever(X) & cough(X) & sniffles(X)
  - remedy(X,chinacea) :- chills(X) & cough(X) & sniffles(X)

  Or relate treatments to diseases, and diseases to symptoms:
  - remedy(X,chinacea) :- has_cold(X)
  - has_cold(X) :- fever(X) & cough(X) & sniffles(X)
  - has_cold(X) :- chills(X) & cough(X) & sniffles(X)

- Design choice for relations, individuals can have impact on ability to prove certain things (easily)
- Suppose we want to find a treatment list for john:
  - list should cover each symptom john exhibits (in KB)
  - but how do we “collect” all the facts from the KB of the form fever(john), slow_digetion(john), etc.
  - (actually Prolog has some hacks, but SLD doesn’t)
- Thus we make our lives easier by thinking of symptoms as individuals, and relating patients to a list of all symptoms
  - symptoms(john, [fever, aches, slow_digetion]).

Some Example Facts/Rules

- We might even have more general rules
  - Appropriate level of generality can make KB expression more concise

  We might have general problems:
  - general_digetion(X) :- slow_digetion(X)
  - general_digetion(X) :- heartburn(X) & relaxed_disposition(X)
  - general_digetion(X) :- gastritis(X)

  and relate treatments to such classes of problems:
  - remedy(X,clove) :- general_digetion(X)
  - remedy(X,meadowsweet) :- gastritis(X)

Example Facts/Rules

- Let’s attempt to define treatment(S,T): treatment list T is satisfactory for symptom list S
  - Note: it suggests new relations to specify/define
  - Is this definition correct? complete? efficient? for what types of queries will it work?

  treatment([],[S],T) :-
  - safe([T],[S])

  treatment([S],[],T) :-
  - treatment(T,S)

  treatment([S],[],T) :-
  - safe([T],[S])
Example Facts/Rules

treatment([], []).
treatment([S1 | Rest S], [T1 | Rest T]) :-
treats(T1, T2),
treatment(Rest S, Rest T),
safe([T1 | Rest T]).

• ?treatment([aches, fever], T): is this defn OK?
• ?treatment([aches, fever], [ech, mudwort]): OK?
  • what if ech treats fever and mudwort treats aches?
  • must rewrite to make order-independent

• Final Tlist is safe if no nasty interactions:
  • why is this definition inefficient?
  • why prove for each sublist? how would you rewrite it?
  • could proving it each time make sense (for Prolog)?
  • Exercise: define a version of the safe predicate

KB Design: The Moral

• There are many design choices
• The queries you plan to ask influence the way you break the world into individuals and relations
• Even with fixed functionality, there are often several ways to approach the problem
• Different approaches lead to more or less natural, efficient, and compact KBs