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 (excl. 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: \( \text{yes} \leftarrow . \)
SLD-Resolution: Algorithm (no vars)

Given query \(?q_1 & \ldots & q_m\) and a KB

1. Construct answer clause \(yes \leftarrow q_1 & \ldots & q_m\)

2. Until no KB-clause choosable or AC is an answer
   (a) Select an atom \(a_i\) from the current AC
       \(yes \leftarrow a_1 & \ldots & a_k\)
   (b) Choose a clause \(a_i \leftarrow b_1 & \ldots & b_n\) from KB
       whose head matches selected atom
   (c) Replace \(a_i\) in AC with body to obtain new AC
       \(yes \leftarrow a_1 & \ldots a_{i-1} & b_1 & \ldots & b_n & a_{i+1} \ldots & a_k\)
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

KB: (1) a <- b & c.
    (2) b <- d & e.
    (2') b <- c.
    (3) b <- g & e.
    (4) c <- e.
    (5) d.
    (6) e.
    (7) f <- a & g.

Query: ?a

Derivation Attempt #1

yes <- a.
yes <- b & c.  Select a; choose (1)
yes <- g & e & c.  Select b; choose (3)
yes <- g & c.  Select e; choose (6)

Select g: FAIL! no choosable clause
## SLD: Example

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

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

**Derivation Attempt #2**

- \( \text{yes} \leftarrow a. \)
- \( \text{yes} \leftarrow b \& c. \)
- \( \text{yes} \leftarrow d \& e \& c. \)
- \( \text{yes} \leftarrow e \& c. \)
- \( \text{yes} \leftarrow c. \)
- \( \text{yes} \leftarrow e. \)
- \( \text{yes} \leftarrow . \)

- Select a; choose (1)
- Select b; choose (2)
- Select d; choose (5)
- Select e; choose (6)
- Select c; choose (4)
- Select e; choose (6)

**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

\[ \frac{a \lor b, \neg b \lor c}{a \lor c} \]

Query: yes \leftarrow g \land h \quad \text{equivalent to} \quad \neg g \lor \neg h \lor \text{yes}

Rule: \quad h \leftarrow a \land b \land c \quad \text{equivalent to} \quad h \lor \neg a \lor \neg b \lor \neg c

\[ \neg g \lor \neg h \lor \text{yes}, \quad h \lor \neg a \lor \neg b \lor \neg c \]

\[ \frac{\neg g \lor \neg h \lor \text{yes}, h \lor \neg a \lor \neg b \lor \neg c}{\neg g \lor \neg a \lor \neg b \lor \neg c \lor \text{yes}} \]

Resolvent: \quad \neg g \lor \neg a \lor \neg b \lor \neg c \lor \text{yes} \quad \text{equiv. to} \quad \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 \( t \)) s.t. \( q(t) \) holds
  - with no functions, terms are just constants

Example:
1. rich(joan).
2. mother(linda,joan).
3. mother(mary,linda).
4. rich(X) <- mother(X,Y) & rich(Y).

Query:
? rich(linda).
  yes

? rich(X).
  joan, linda, mary
SLD: Queries with no vars

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

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

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

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

Query:
? rich(linda).

Derivation:
yes <- rich(linda).
yes <- mother(linda,Y) & rich(Y).
   How: Select rich(linda); resolve with (4) using {X/linda}
yes <- rich(joan).
   How: Select mother(linda,Y); resolve with (2) using {Y/joan}
yes <- .
   How: Select rich(joan); resolve with (1) using { }
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: 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:
? rich(Z).

Derivation:
yes(Z) <- rich(Z).
yes(Z) <- mother(Z,Y) & rich(Y).
   Select rich(Z); resolve with (4) using {X/Z}
yes(Z) <- mother(Z,joan).
   Select rich(Y); resolve with (1) using {Y/joan}
yes(linda) <- .
   Select mother(Z,joan); resolve with (2) using {Z/linda}
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:
? rich(Z).

A Different Derivation:
yes(Z) <- rich(Z).
yes(joan) <- .

Select rich(Z); resolve with (1) using {Z/joan}

Different derivations can give different answers;
Exercise: construct derivation that gives the answer “mary”.
SLD with Variables

- To recap, we’ve seen SLD with:
  - variables in KB, but ground queries
  - variables in KR and variables in query

- Basic idea: we need to make appropriate substitutions of our variables in order to make atoms in answer clause match heads of KB rules

- Let’s look at one more example, sticking with the “intuitive” definition of a substitution

- Then we’ll formalize unifiers and MGUs
Example Derivation #1

KB:
1. busy(Z) <- teaches(Z,X) & teaches(Z,Y) & distinct(X,Y).
2. busy(Z) <- teaches(Z,148).
3. teaches(craig, 384).
4. teaches(craig, 2534).
5. teaches(kyros, 384).
6. teaches(kyros, 2501).
7. teaches(suzanne, 148).
8. distinct(2534,384).
9. distinct(2501,384).

distinct...

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/Z}
yes(suzanne) <- .

Select t(Z,148): resolve with (2) using {Z/P}

Answer: suzanne
(others: craig, kyros... show!)

Could have used {Z/P} instead; as long as vars match
SLD-Resolution: Algorithm (w/ vars)

Given query \( ?q_1 \& \ldots \& q_m \) with vars \( x_1 \ldots x_n \) and a KB

1. Construct answer clause \( yes(x_1 \ldots x_n) \leftarrow q_1 \& \ldots \& q_m \).

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

   (a) Select an atom \( a_i \) from the current AC \( yes \leftarrow a_1 \& \ldots \& a_k \)

   (b) Choose a clause \( h_i \leftarrow b_1 \& \ldots \& b_n \) from KB

   and a substitution \( \sigma \) that unifies the head \( h_i \) of the KB clause with the selected atom \( a_i \) (i.e., that when applied to \( h_i \) and \( a_i \) makes them the same)

   (c) apply \( \sigma \) to AC and KB clause to obtain \( AC\sigma, KB\sigma \)

   (d) Replace \( a_i \sigma \) in \( AC\sigma \) with body of \( KB\sigma \) to obtain new AC

\[
(yes(x_1 \ldots x_n) \leftarrow a_1 \& \ldots \& a_{i-1} \& b_1 \& \ldots \& b_n \& a_{i+1} \ldots \& a_k) \sigma
\]
Example Derivation #2

KB:
1. busy(Z) <- teaches(Z,X) &
   teaches(Z,Y) & distinct(X,Y).
2. busy(Z) <- teaches(Z,148).
3. teaches(craig, 384).
4. teaches(craig, 2534).
5. teaches(kyros, 384).
6. teaches(kyros, 2501).
7. teaches(suzanne, 148).
8. distinct(2534,384).
9. distinct(2501,384).

Derivation:
yes(P) <- busy(P).
yes(P) <- t(P,X) & t(P,Y) & d(X,Y).
   busy(P); (1): {Z/P}
yes(craig) <- t(craig,Y) & d(384,Y).
   t(P,X); (3): {P/craig, X/384}
yes(craig) <- d(384,2534).
   t(c,Y); (4): {X/2534}

FAILS! Nothing will unify with d(384,2534).

Problem lies in KB. We didn’t axiomatize domain correctly.
Add distinct(384,2534), etc...
or add rule: distinct(C,D) <- distinct(D,C).

Same query: ?busy(P).
Example Derivation #3

Assume KB fixed with rule: $12. \text{distinct}(C,D) \leftarrow \text{distinct}(D,C)$.

Derivation:

```
yes(P) <- busy(P).
yes(P) <- t(P,X) & t(P,Y) & d(X,Y).
  busy(P); (1); \{Z/P\}
yes(craig) <- t(craig,Y) & d(384,Y).
  t(P,X); (3); \{P/craig, X/384\}
yes(craig) <- d(384,2534).
  t(c,Y); (4); \{X/2534\}
yes(craig) <- d(2534,384).
  d(384,2534); (12); \{C/384, D/2534\}
yes(craig) <- .
  d(2534,384); (8); {}```

CSC 384 Lecture Slides (c) 2002, C. Boutilier
Substitutions

- **Defn:** A substitution $\sigma$ is any assignment of terms to variables
  - we write it like as $\sigma = \{X/t_1, Y/t_2, \ldots\}$
  - constant substitution is a special case; terms can be any terms (nonground included)
    - without functions, only terms are constants, vars
  - e.g., $\sigma = \{X/craig, Y/father(craig), Z/P, W/father(X)\}$

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 $related(mother(craig), father(X))$
Unifiers

- **Defn**: A substitution unifies two expressions $e_1$ and $e_2$ iff $e_1\sigma$ 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
  - $\{X/Y, Z/a\}$: gives $p(Y,f(a))$ for both expressions
  - $\{X/Z, Y/Z, Z/a\}$: gives $p(Z,f(a))$ for both expressions

- Unifier $\sigma$ is a **most general unifier (MGU)** of $e_1$ and $e_2$ iff $e_1\sigma'$ 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.
MGUs: Examples

- Let $e_1 = \text{busy}(X)$, $e_2 = \text{busy}(Y)$

- Unifier $\sigma_1$: \{X/kyros, Y/kyros\}
  - result: $e_1\sigma_1 = e_2\sigma_1 = \text{busy}(kyros)$

- Unifier $\sigma_2$: \{X/craig, Y/craig\}
  - 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 $\sigma_3$ an MGU of expressions; not $\sigma_1$, $\sigma_2$
  - $e_1\sigma_3$ unifies with result of any other unifier
  - $e_1\sigma_1 = \text{busy}(kyros)$ **cannot** (e.g., cannot unify $e_1\sigma_1$ with $e_2\sigma_1 = \text{busy}(craig)$)
Notes on General SLD Resolution

- Generally insist that 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 into 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).

Query:
? rich(mary).

Derivation:
yes <- rich(mary).
yes <- mother(mary,Y) & rich(Y).
rich(mary); (4); \{X/mary\}
yes <- mother(mary,X) & mother(X,X) & 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_i) <- mother(X_i,Y_i) & rich(Y_i). i-th time you use this clause in a derivation
Renaming of Variables: Example

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

Query:
? rich(mary).

Derivation:
yes <- rich(mary).
yes <- mother(mary,Y₁) & rich(Y₁).
   rich(mary); (4); {X₁/mary}
yes <- mother(mary,X₂) & mother(X₂,Y₂) & rich(Y₂).
   rich(Y₁); (4) using {Y₁/X₂}
etc... (no conflict now)
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)` or `[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

(A1) \text{append}([], Z, Z).

(A2) \text{append}([E1 \mid R1], Y, [E1 \mid \text{Rest}]) \leftarrow \text{append}(R1, Y, \text{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], [b,c,d]).
   Resolve with (A2) using { E1/a, R1/[b], Y/[c,d], Rest/[b,c,d] }
yes <- append([], [c,d], [c,d]).
   Resolve with (A2) using { E1/b, R1/[], Y/[c,d], Rest/[c,d] }
yes <- .
   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], [f,b,c,d]).

No append rule can unify with this atom
  (convince yourself: look at E1)

Answer: no
Proving the Append Relation #3

Query: \( ? \text{append}(L, M, [a,b,c,d]). \)

(A1) \text{append}([], Z, Z).

(A2) \text{append}([E1 | R1], Y, [E1 | Rest]) \iff \text{append}(R1, Y, Rest).

Derivation:
yes(L,M) \iff \text{append}(L, M, [a,b,c,d]).
yes([a|R1], M) \iff \text{append}(R1, M, [b,c,d]).
    Resolve with (A2) using \{ L/[a|R1], Y/M, E1/a, Rest/[b,c,d] \}
yes([a], [b,c,d]) \iff .
    Resolve with (A1) using \{ R1/[], M/[b,c,d], 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:** \(? \text{ append}(L, M, [a,b,c,d]).\)
  - \(L = [], M = [a,b,c,d]\)
  - \(L = [a], M = [b,c,d]\)
  - \(L = [a,b], M = [c,d]\)
  - \(L = [a,b,c], M = [d]\)
  - \(L = [a,b,c,d], M = []\)
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) \(?\text{treatment}(\text{john}, T)\).
  b) \(?\text{treatment}(\text{symptom}, T)\).
  c) \(?\text{treatment}(\text{combination-of-symptoms}, T)\).
  d) \(?\text{safe}(\text{combination-of-treatments})\).
  e) \(?\text{medical\_records}(\text{john}, R)\).
  f) \(?\text{paid\_bills}(\text{john})\).

- and so on
What Individuals Do We Need?

- What constants/functions will I need?
- Clients (people), other entities:
  - constants: joan, ming, gabrielle, greenshield…
  - functions: insurer(X), etc.
- Symptoms (constants): fever, aches, chills, …
- Treatments:
  - constants: echinacea, mudwort, feverfew, …
  - or maybe function: tmt(feverfew,capsule),
    tmt(mudwort, tincture), where we have a treatment
    requires a substance and a preparation
  - then we need constants for substances, preparations
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: \( \text{symList}(\text{symptom}, \text{SList}) \)
  - or using Prolog notation: \([\text{aches}, \text{fever}, \text{chills}]\)
- treatmentList similar:
  - \([\text{tmt}(<\text{mudwort}, \text{tincture}>>, \text{tmt}(<\text{echinacea}, \text{capsule}>>)\]

CSC 384 Lecture Slides (c) 2002, C. Boutilier
What Relations?

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

  **Semi-permanent Facts (persist in KB):**
  - arthritis(ming).
  - hypertensive(john).
  - relaxed_disposition(mary).
Some Example Facts/Rules

- Rules relating treatments to symptoms

We can relate treatments to symptoms directly:
  remedy(X, echinacea) :- fever(X) & cough(X) & sniffsles(X).
  remedy(X, echinacea) :- chills(X) & cough(X) & sniffsles(X).

Or relate treatments to diseases, and diseases to symptoms:
  remedy(X, echinacea) :- has_cold(X).

  has_cold(X) :- fever(X) & cough(X) & sniffsles(X).
  has_cold(X) :- chills(X) & cough(X) & sniffsles(X).
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:

```prolog
general_digit_problems(X) :- slow_digestion(X).
general_digit_problems(X) :- heartbeat(X) & relaxed_disposition(X).
general_digit_problems(X) :- gastritis(X).
```

and relate treatments to such classes of problems:

```prolog
remedy(X,clove) :- general_digit_problems(X).
remedy(X,meadowsweet) :- gastritis(X).
```
Some Example Facts/Rules

- 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_digestion(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_digestion]).*
Example Facts/Rules

- Let’s attempt to define \( \text{treatment}(S, T) \): 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?

\[
\begin{align*}
treatment( [], [] ). \\
treatment( [S1 | \text{Rest}S], [T1 | \text{Rest}T] ) :&- \\
\quad \text{treats}(T1,S1), \\
\quad \text{treatment}(\text{Rest}S, \text{Rest}T), \\
\quad \text{safe}( [T1 | \text{Rest}T] ).
\end{align*}
\]
Example Facts/Rules

```
treatment([], []).
treatment([S1 | RestS], [T1 | RestT]) :-
  treats(T1, S1),
  treatment(RestS, RestT),
  safe([T1 | RestT]).
```

- `?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