Principles of Programming Languages VIII

Wael Aboelsaadat
wael@cs.toronto.edu
http://www.dgp.toronto.edu/~wael/324.html
Today

- Logic Programming
- Prolog I
LP: resolution in predicate calculus

- We would like to infer new propositions (e.g. facts) from some existing set of propositions.

- An inference rule that can be applied atomically is called a *resolution*
  - E.g.
    
    Given: \[ P_1 \leftarrow P_2, Q_1 \leftarrow Q_2 \]
    \[ P_1 \equiv Q_2 \]
    Alternatively: \[ T \leftarrow P_2, Q_1 \leftarrow T \]
    New fact: \[ Q_1 \leftarrow P_2 \]

- Resolution gets more complex if variables/values are involved:
  - To use resolution with variables, we will need to find values for variables that allow matching to proceed.
  - E.g.
    
    Given: \[ F(X,Y) \leftarrow P_2(Y,X) \]
    \[ Q_1(foo) \leftarrow F(foo, bar) \]
    New fact: \[ Q_1(foo) \leftarrow P_2(bar, foo) \]
LP: horn clause

• Logic programming is heavily based on horn clauses:

\[ c \leftarrow h_1 \land h_2 \land h_3 \land \ldots \land h_n \]

  – Antecedents (h's): conjunction of zero or more conditions which are atomic constructs in predicate logic.
  – Consequent(c): an atomic construct in predicate logic

• Meaning of a horn-clause:
  – The consequent is true if the antecedents are all true
  – c is true if \( h_1, h_2, h_3, \ldots \) are all true

• A horn clause can capture most, but not all, logical statements/implications, why?
LP: specifying non-horn rules

- Many non-horn rules can be transformed to horn form using one of two methods:
  - logical equivalence
  - Skolemization

- Logical equivalence:
  - Uses the following logical laws:
    - Negation: \( \neg \neg A \equiv A \)
    - De Morgan’s Law: \( \neg (A \lor B) \equiv \neg A \land \neg B \)
      \( \neg (A \land B) \equiv \neg A \lor \neg B \)
    - Distributive Property: \( A \lor (B \land C) \equiv (A \lor B) \land (A \lor C) \)
      \( A \land (B \lor C) \equiv (A \land B) \lor (A \land C) \)
    - Absorption Law: \( A \lor (A \land B) \equiv A \)
      \( A \land (A \lor B) \equiv A \)
    - Implication Laws: \( A \iff B \equiv (A \implies B) \land (B \implies A) \)
      \( A \iff B \equiv A \lor \neg B \)
LP: specifying non-horn rules

• Logical equivalence rules:
  • Negation \( \neg \neg A \equiv A \)
  • De Morgan’s Law \( \neg (A \lor B) \equiv \neg A \land \neg B \)
  \( \neg (A \land B) \equiv \neg A \lor \neg B \)
  • Distributive Property \( A \lor (B \land C) \equiv (A \lor B) \land (A \lor C) \)
  \( A \land (B \lor C) \equiv (A \land B) \lor (A \land C) \)
  • Absorption Law \( A \lor (A \land B) \equiv A \)
  \( A \land (A \lor B) \equiv A \)
  • Implication Laws \( A \iff B \equiv (A \rightarrow B) \land (B \rightarrow A) \)
  \( A \iff B \equiv A \lor \neg B \)

• Examples:
  • \( \neg A \iff \neg B \equiv \neg A \lor \neg (\neg B) \)
    \[ \equiv \neg A \lor B \]
    \[ \equiv B \lor \neg A \]
    \[ \equiv B \iff A \quad \text{(horn-clause)} \]

  • \( A \iff (B \lor C) \equiv A \lor (\neg B \land \neg C) \)
    \[ \equiv A \lor (\neg B \land \neg C) \]
    \[ \equiv (A \lor \neg B) \land (A \lor \neg C) \]
    \[ \equiv (A \iff B) \land (A \iff C) \quad \text{(horn-clause)} \]
LP: specifying non-horn rules – cont’d

- Logical equivalence rules:
  - Negation
    \[ \neg \neg A \equiv A \]
  - De Morgan’s Law
    \[ \neg (A \lor B) \equiv \neg A \land \neg B \]
    \[ \neg (A \land B) \equiv \neg A \lor \neg B \]
  - Distributive Property
    \[ A \lor (B \land C) \equiv (A \lor B) \land (A \lor C) \]
    \[ A \land (B \lor C) \equiv (A \land B) \lor (A \land C) \]
  - Absorption Law
    \[ A \lor (A \land B) \equiv A \]
    \[ A \land (A \lor B) \equiv A \]
  - Implication Laws
    \[ A \leftrightarrow B \equiv (A \Rightarrow B) \land (B \Rightarrow A) \]
    \[ A \leftrightarrow B \equiv A \lor \neg B \]

- Examples:
  - \[ A \leftrightarrow (B \leftrightarrow C) \equiv A \lor \neg (B \leftrightarrow C) \]
    \[ \equiv A \lor \neg (B \lor \neg C) \]
    \[ \equiv A \lor (\neg B \land \neg \neg C) \]
    \[ \equiv A \lor (\neg B \land C) \]
    \[ \equiv (A \lor \neg B) \land (A \lor C) \]
    \[ \equiv (A \leftrightarrow B) \land (A \lor C) \quad (non-horn) \]
  - In general, rules of the form
    \[ (\forall X)[(A_1 \lor \ldots \lor A_n) \leftrightarrow (B_1 \land \ldots \land B_m)] \]
    cannot be converted to horn-clause
LP: specifying non-horn rules – cont’d

• Horn clause
  – What are we going to do about quantifiers?

• Skolemization:
  – Variables bound by existential ($\exists X$) quantifiers which are not inside the scope of universal quantifiers can simply be replaced by constants:
    • ($\exists X$) [$X < 3$] becomes $c < 3$
    • ($\exists X$) mother(john,X) becomes mother(john,m)

  – When the existential quantifier ($\exists X$) is inside a universal quantifier ($\forall Y$), the bound variable must be replaced by a function of the variables bound by universal quantifier ($\forall Y$).
    • ($\forall X$) [$X=0 \lor \exists(Y) [X=Y+1]$ becomes ($\forall X$) [$X=0 \lor X = f(X) + 1$]
    • ($\forall X$) [person(X) $\Rightarrow$ ($\exists Y$) mother(X,Y)] becomes ($\forall X$) [person(X) $\Rightarrow$ mother(X,m(X))]

LP: specifying non-horn rules – cont’d

• Skolemization:
  – Non horn formulas like $(\exists X) A(X)$ can be converted to horn-clause by introducing a skolem constant and/or skolem function. The resulting clause is almost the same thing.

• Why does skolemization works?
  – We only need $\exists X$ because we don’t have a name for X. By creating artificial names (skolem names), we can eliminate many $\exists$’s and convert many formulas to horn clause.
LP: horn clause made easy!

- Horn clause can include more complex terms:

  \[ p(X) \leftarrow q(X,Y) \land r(X,Y) \land s(X,Y) \]
  \[ p(X) \leftarrow \ldots \]
  \[ p(X) \leftarrow \ldots \]
  \[ p(X) \leftarrow \ldots \]
  \[ p(X) \leftarrow \ldots \]
  \[ p(X) \leftarrow \ldots \]

- We can assume the following when writing horn-clauses:
  - \( p \) is the program name
  - \( q,r,s \) are the subprogram names
  - \( X \) is a parameter of the program
  - \( Y \) is a local variable
Prolog I
The first and most popular logic programming language

- Invented by Alain Colmerauer and Phillipe Roussel at the University of Aix-Marseille in 1971 (France)

Characteristics:
- Is very weakly typed
- Has no data abstraction
- Has no functional abstraction!
- Has no mutable state
- Has no explicit control flow

So, how do you program?
- Load facts/rules into interpreter
- Make queries to see if a fact is:
  - in the knowledge-base or
  - can be implied from existing facts or rules

Prolog is really an engine to prove theorems
**Prolog: data types – quick intro**

- **Simple**
  - Constants:
    - Numbers: integer, floating point,…
    - Atoms: alphabetic sequence **starting with a lower case letter** (e.g. apple)
  - Variables:
    - Variables start with **capital letters** or underscore

- **Complex**
  - Lists
  - Structures
Prolog: horn clauses

• Recall

\[ c \leftarrow h_1 \land h_2 \land h_3 \land \ldots \land h_n \]

• Syntax:

\[ <\text{head}> :\!-\! <\text{body}> \]

  – You can conclude that <head> is true, if you can prove that <body> is true
  – The symbol :\!-\! is read as *if*

• 3 types of clauses:

  – Facts
  – Rules
  – Queries
Prolog: facts

• A fact is a clause with an empty body

• Syntax

  <head>.

• What makes a fact a fact?

• Examples
  – Exams
  – Assignments
  – Taxes
  – The earth is round.
  – The sky is blue.
  – The sun is hot.
  – Mary is a female.
  – Beethoven lived between 1770 & 1827.
Prolog: facts – cont’d

• **Facts about facts:**
  – Full stop “.” at the end of every fact.
  – The number of arguments in a fact is called **arity**.
    • E.g. `female(mary).` is an instance of `female/1` (functor `female`, arity 1)
  – Facts with different number of arguments are distinct
    • E.g. `female(mary,may).` is different from `female(mary).`
Prolog: rules

• A rule in Prolog is a full horn clause:
  \[ c \leftarrow h_1 \land h_2 \land h_3 \land \ldots \land h_n \]

• Syntax:
  \[ \text{rel}_1 :\!- \text{rel}_2, \text{rel}_3, \ldots \text{rel}_n. \]

  - If I know that all those relations (those in the body) hold, then I also know that this LHS relation (in the head) holds.

• Examples:
  – If there is smoke there is fire
    \[ \text{fire} :\!- \text{smoke}. \]

  – If the course is boring, I leave
    \[ \text{leave(i)} :\!- \text{boring(course)}. \]

  – Joe is going to kill the teacher if he fails CSC324.
    \[ \text{kills(joe, X)} :\!- \text{fails(joe,csc324), teaches(X,csc324)}. \]
• **Examples:**
  – X is female if X is the mother of anyone.
    
    female(X) :- mother(X, _).

  – X is the sister of Y, if X is female and X's parents are M and F, and Y's parents are M and F
    
    sister_of(X, Y) :- female(X), parents(X, M, F), parents(Y, M, F).

• **When to use rules?**
  – Use rules to say that a particular fact depends on a group of facts.
  – Use rules to deduce new facts from existing ones.

• **Rules of rules:**
  – The head of the rule consist of at most one predicate
  – The body of the rule is a finite sequence of literals separated by , or conjunction (*and*)
  – Rules always end with a period “.”
Prolog: queries

- A query is a clause with an empty head.

  \[ \leftarrow h_1 \land h_2 \land h_3 \land \ldots \land h_n \]

- Syntax

  \[ ? \langle \text{body} \rangle. \]

  - Try to prove that \langle body \rangle is true
  - The goal is represented to the interpreter as a question.

- Examples

  \[ ? \text{-round(earth)}. \]
  - is it true that the earth is round?
    (or simpler than that: is the earth round?)

  \[ ? \text{-round(X)}. \]
  - is it true that there are entities which are round?
    (or simpler than that: what entities are round?)
Prolog: queries – cont’d

• Examples

|?-composer(beethoven,1770,1827). | - is it true that beethoven was a composer who lived between 1770 and 1827 |

|?-owns(john,book). | - is it true that john owns a book? |
| (simpler: does john own a book?) |

|?-owns(john,X). | - is it true that john owns something? |
| (simpler: does john own something?) |
Prolog: simple types - constants

• There are two types of constants: *atoms* and *numbers*.

• Atoms:
  – Alphanumeric atoms: *alphabetic sequence starting with a lower case letter*
    • E.g.: apple a1 apple_cart
  – Special atoms
    • E.g ! ; [ ] {}
  – Symbolic atoms: *sequence of symbolic characters*
    • E.g. & < > * - + >>
  – Quoted atoms: *sequence of characters surrounded by single quotes*
    • Can make anything an atom by enclosing it in single quotes.
    • E.g ‘Apple’ ‘hello world’

• Numbers:
  – Integers and Floating Point numbers
    • E.g. 0 1 9821 -10 1.3 -1.3E102
Prolog: simple types - variables

- Variables start with **capital letters** or **underscore**

- **There are no global variables** (*assert* and *retract*, will see them later…)

- **Instantiated vs. un-instantiated:**
  - if the object a variable stands for is already determined, var is **instantiated**
  - if the object a variable stands for is not yet determined, var is **un-instantiated**

- An instantiated variable in Prolog cannot change its value

- Variables are limited in scope to the clause they appear in (**local vars**)
  - E.g.
    - `grandParent(X,Z) :- parent(X,Y), parent(Y,Z). % The Xs here are the same var`
    - `Parent(X,Y) :- mother(X,Y). % But not the same as those here`

- **There is a special anonymous variable “_” which is used to denote “don’t care”**
  - E.g
    - `Parent(X) :- mother(X, _).`
    - `married(X) :- husband(X, _).`
  - Note that every use of _ is considered a separate variable
# Prolog: example 1

## Facts

| likes(eve, pie). | food(pie). |
| likes(al, eve). | food(apple). |
| likes(eve, tom). | person(tom). |
| likes(eve, eve). |          |

## Query

```
?-likes(al, pie).
no

?-likes(al, eve).
yes

?-likes(eve, al).
no

?-likes(person, food).
no
```

Variables:

- `?`: Represents a variable.
- `Who`: A variable that can be bound to a value.

**Answer with variable binding**:

```
?‑likes(al, Who).
Who=eve

?-likes(eve, W).
W=pie ;
W=tom ;
W=eve
```

**Force search for more answers**:

```
no
```
Prolog: example 1 – cont’d

Facts

likes(eve, pie).  food(pie).
likes(al, eve).  food(apple).
likes(eve, tom).  person(tom).
likes(eve, eve).

?-likes(A,B).
A=eve, B=pie ; A=al, B=eve ; ...  
?-likes(D,D).
D=eve ; no
?-likes(eve,W), person(W).
W=tom
?-likes(al,V), likes(eve,V).
V=eve ; no
Prolog: proof procedure

- Two main processes:
  - Unification
  - Top-down reasoning
Prolog: unification

- First step in proof procedure

- Prolog tries to satisfy a query by *unifying* it with some conclusion and see if it is true!

- Process of finding these suitable "assignments" of values to variables is called *unification*
  - It is really a process of pattern matching to make statements identical
  - Somewhat similar to variable bindings in imperative world and to pattern matching in ML.
### Prolog: unification – cont’d

- **Rules of unification:**

<table>
<thead>
<tr>
<th>Object 1</th>
<th>Object 2</th>
<th>example</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>free var.</td>
<td>4</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X=4</td>
</tr>
<tr>
<td>bound variable</td>
<td>free variable</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Y gets the value of X</td>
</tr>
<tr>
<td>free variable</td>
<td>bound variable</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X gets the value of Y</td>
</tr>
<tr>
<td>bound variable</td>
<td>constant</td>
<td>X</td>
<td>“b”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fails if X has a value different then “b”</td>
</tr>
<tr>
<td>compound object</td>
<td>compound object</td>
<td>f(X,Y)</td>
<td>f(2,3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X=2, Y=3</td>
</tr>
<tr>
<td>compound object</td>
<td>compound object</td>
<td>f(q(2,X),3)</td>
<td>f(P,3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>succeeds if P is free, and P=q(2,X) . (.. more possibilities )</td>
</tr>
<tr>
<td>compound object</td>
<td>compound object</td>
<td>f(3,X)</td>
<td>q(3,X)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fails, due to different functors (p is not q)</td>
</tr>
</tbody>
</table>
• Rules of unification:

  – A constant unifies only with itself, it cannot unify with any other constant.

  – Two structures unify iff they have the same name, number of arguments and all the arguments unify.

  – Unification requires all instances of the same variable in a rule to get the same value.
Prolog: unification – cont’d

- Examples:

  \[
  a(b, C, d, E) \\
  \text{with } x(\ldots) \quad \text{doesn’t unify: } a \text{ and } x \text{ differ}
  \]

  \[
  a(b, C, d, E) \\
  a(_, _, _) \quad \text{no: different # of args}
  \]

  \[
  a(b, C, d, E) \\
  a(j, f, G, H) \quad \text{no: } b \neq j
  \]

  \[
  a(b, C, d, E) \\
  a(b, f, G, H) \quad \text{yes: by either } \{C \mapsto f, G \mapsto d, H \mapsto E\} \text{ or } \{C \mapsto f, G \mapsto d, E \mapsto H\}
  \]

  \[
  a(\text{pred}(X, j)) \\
  a(\text{pred}(k, j)) \quad \text{yes: } \{X \mapsto k\}
  \]

  \[
  a(\text{pred}(X, j)) \\
  a(\text{pred}(X, j)) \quad \text{yes: } \{B \mapsto \text{pred}(X, j)\}
  \]
• **Examples:**
  – Does $p(X,X)$ unify with $p(b,b)$?
  – Does $p(X,X)$ unify with $p(b,c)$?
  – Does $p(X,b)$ unify with $p(Y,Y)$?
  – Does $p(X,Z,Z)$ unify with $p(Y,Y,b)$?
  – Does $p(X,b,X)$ unify with $p(Y,Y,c)$?
    - To make the third arguments equal, we must replace $X$ by $c$.
    - To make the second argument equal, we must replace $Y$ by $b$.
    - So, $p(X,b,X)$ becomes $p(c,b,c)$, and $p(Y,Y,c)$ becomes $p(b,b,c)$.
    - However, $p(c,b,c)$ and $p(b,b,c)$ are not syntactically identical.
Prolog: example 2

- **Facts & rules:**

```prolog
link(a,b), link(b,c), link(a,d), link(d,c).
path(N, N).
path(L, M) :- link(L, X), path(X, M).
```

- **Posing queries:**

Based on our logical encoding of the graph, we can then write queries:

```prolog
?- path(a,c)
yes

?- path(c,a)
no

?- path(a,X), path(X,c)
X = a
X = b
X = c
X = d
```

Notice that we didn’t write a graph traversal algorithm, and we didn’t hard code the set of questions we can ask in advance. We just define what a graph is…
Prolog: proof procedure - revisited

- **Two main processes:**
  - Unification
  - Top-down reasoning
Prolog: reasoning

• Given a set of facts and rules, we need a mechanism to deduce new facts and/or prove that a given rule is true or false or has no answer

• There are two techniques to do this:
  – Bottom-up reasoning
  – Top-down reasoning
Prolog: bottom-up reasoning

- **Bottom-up** (or forward) reasoning: starting from the given facts, apply rules to infer everything that is true.

  *e.g.,* Suppose the fact \( B \) and the rule \( A \leftarrow B \) are given. Then infer that \( A \) is true.

**Example**

Rule base:

\[
p(X,Y,Z) \leftarrow q(X), q(Y), q(Z).
q(a1).
q(a2).
\ldots
q(an).
\]

Bottom-up inference derives \( n^3 \) facts of the form \( p(a_i,a_j,a_k) \):

\[
p(a_1, a_1, a_1)
p(a_1, a_1, a_2)
p(a_1, a_2, a_3)
\ldots
\]

A rule base:

\[
A \leftarrow B \quad (1)
B \leftarrow C \quad (2)
C \quad (3)
\]

A bottom-up proof:

\[
\text{infer } A
\quad \text{rule (1)}
\]

\[
\text{infer } B
\quad \text{rule (2)}
\]

\[
\text{infer } C
\quad \text{rule (3)}
\]

\[
\text{start}
\]

So, \( A \) is proved.
Top-down (or backward) reasoning: starting from the query, apply the rules in reverse, attempting only those lines of inference that are relevant to the query.

e.g., Suppose the query is $A$, and the rule $A \leftarrow B$ is given. Then to prove $A$, try to prove $B$.

A rule base:

\[
\begin{align*}
A & \leftarrow B \quad (1) \\
B & \leftarrow C \quad (2) \\
C & \quad (3)
\end{align*}
\]

A top-down proof:

\[
\begin{align*}
goal \ A & \\
\quad \downarrow & \quad \text{rule (1)} \\
goal \ B & \\
\quad \downarrow & \quad \text{rule (2)} \\
goal \ C & \\
\quad \downarrow & \quad \text{rule (3)} \\
\text{success} & \\
\end{align*}
\]

So, $A$ is proved
Multiple rules and multiple premises:

- A fact may be inferred by many rules
  - E.g. \[
  \begin{align*}
  E & \leftarrow B \\
  E & \leftarrow C \\
  E & \leftarrow D
  \end{align*}
  \]

- A rule may have many premises
  - E.g. \[
  \begin{align*}
  E & \leftarrow B \land C \land D
  \end{align*}
  \]

In top-down inference, such rules give rise to

- Inference trees
- Backtracking
Prolog: top-down reasoning – cont’d

- Example: *multiple premises*

**Rule base:**

(1) \( A \leftarrow B_1 \land B_2 \)
(2) \( B_1 \leftarrow C_1 \land C_2 \)
(3) \( B_2 \leftarrow C_3 \land C_4 \)

\( C_1 \quad C_2 \quad C_3 \quad C_4 \)

**Query:** Is \( A \) true?

\[ \text{Goal } A \]
\[ \downarrow \text{Rule (1)} \]
\[ B_1 \land B_2 \]
\[ \downarrow \text{Rule (2)} \]
\[ C_1 \land C_2 \]
\[ \quad \downarrow \text{success} \]
\[ \quad \text{Goal } C_1 \]

\[ \downarrow \text{Rule (3)} \]
\[ C_3 \land C_4 \]
\[ \quad \downarrow \text{success} \]
\[ \quad \text{Goal } C_4 \]

\[ \downarrow \text{success} \]
\[ \text{Goal } C_2 \]
\[ \quad \downarrow \text{success} \]
\[ \quad \text{Goal } C_3 \]

\[ \quad \downarrow \text{success} \]
\[ \text{Goal } B_1 \]

So, goal \( A \) is proved. (all paths must succeed)
Prolog: top-down reasoning – cont’d

- **Example:** *multiple rules*

**Rule base:**

\[
\begin{align*}
A & \leftarrow B_1 & B_1 & \leftarrow C_1 & B_2 & \leftarrow C_3 \\
A & \leftarrow B_2 & B_1 & \leftarrow C_2 & B_2 & \leftarrow C_4 \\
& & & C_4 \\
\end{align*}
\]

**Query:** Is A true?

- So, goal A is proved. (only one path must succeed)
Prolog: backtracking

- Prolog uses this algorithm for proving a goal by recursively breaking goal down into sub-goals and try to prove these sub-goals until facts are reached.

- To satisfy a goal:
  - Try to unify with conclusion of first rule in database
  - If successful, apply substitution to first premise, try to satisfy resulting sub-goals
  - Then apply both substitutions to next sub-goal (premise), and so on...
  - If not successful, go on to the next rule in database
  - If all rules fail, try again (backtrack) to a previous sub-goal
Rule base:

\[
p(X) \leftarrow q(X), r(X).
q(d), \quad q(e), \quad q(f), \quad q(g).
\]

\[
r(e), \quad r(g).
\]

Query: Find \(x\) such that \(p(x)\) is true.

\[
p(X)
\]

\[
q(X), \ r(X)
\]

\[
X=d \quad r(d) \text{ fail}
\]

\[
X=e \quad r(e) \text{ success (print ”X=e”)}
\]

\[
X=f \quad r(f) \text{ fail}
\]

\[
X=g \quad r(g) \text{ success (print ”X=g”)}
\]
Prolog: backtracking example 2

Rule base:

\[ p(X) : = q(X), r(X,Y), s(Y). \]
\[ q(a). \quad r(a,b). \quad r(c,b). \quad s(c). \]
\[ q(c). \quad r(a,c). \quad r(c,c). \]
\[ r(a,d). \]

Query: Find \( x \) such that \( p(x) \) is true.
Prolog: backtracking example 3

?- located_in(toronto, north_america).

matches [8] under X=toronto

?- located_in(toronto, usa).

matches [5] under X=toronto

?- located_in(toronto, georgia)

No Matches
Fail

[1] located_in(atlanta, georgia).
[5] located_in(X, usa) :- located_in(X, georgia).
[6] located_in(X, usa) :- located_in(X, colorado).
[8] located_in(X, north_america) :- located_in(X, usa).
[9] located_in(X, north_america) :- located_in(X, canada).

?- located_in(toronto, north_america).
Prolog: backtracking example 3 – cont’d

[1] located_in(atlanta, georgia).
[5] located_in(X, usa) :- located_in(X, georgia).
[6] located_in(X, usa) :- located_in(X, colorado).
[8] located_in(X, north_america) :- located_in(X, usa).
[9] located_in(X, north_america) :- located_in(X, canada).

?- located_in(toronto, north_america).
Prolog: backtracking example 3 – cont’d

```prolog
[1] located_in(atlanta, georgia).
[5] located_in(X, usa) :- located_in(X, georgia).
[6] located_in(X, usa) :- located_in(X, colorado).
[8] located_in(X, north_america) :- located_in(X, usa).
[9] located_in(X, north_america) :- located_in(X, canada).

?- located_in(toronto, north_america).
```

Diagram:

```
?- located_in(toronto, north_america).
    matches [9] under X=toronto
      |- matches [3] under X=toronto
      |  |- ?- located_in(toronto, usa).
      |  matches [5] under X=toronto
      |  |- ?- located_in(toronto, georgia).
      |     No Matches
      |     Fail
      |  matches [6] under X=toronto
      |  |- ?- located_in(toronto, colorado).
      |     No Matches
      |     Fail
      |- matches [7] under X=toronto
      |- ?- located_in(toronto, canada).
      matches [4] as is
      no body => success
```
Prolog: top-down vs. bottom-up reasoning

- Prolog uses top-down inference, although some other logic programming systems use bottom-up inference (e.g. Coral).

- Each has its own advantages and disadvantages:
  - Bottom-up may generate many irrelevant facts
  - Top-down may explore many lines of reasoning that fail.

- Top-down and bottom-up inference are logically equivalent
  - i.e. they both prove the same set of facts.

- However, only top-down inference simulates program execution
  - i.e. execution is inherently top down, since it proceeds from the main procedure downwards, to subroutines, to sub-subroutines, etc...
Prolog: complex types - structures

- **Recall:** what’s a function term?
  \[ \text{functor}(\text{some-parameters}) \quad \text{e.g. woman(marry)} \]

- **We can construct complex data structures using nested function terms.**
  - Represents a statement about the world

- **Example:**
  - A person has; name: first name, last name - birth date: day, month, year & occupation

\[
\text{person(name(michael, jordan), birth\_date(17, february, 1963), occupation('NBA player'))}
\]
Prolog: complex types - structures

Database:

owns(john, car(red, corvette))
owns(john, cat(black, siamese, sylvester))
owns(elvis, copyright(song, "jailhouse rock"))
owns(tolstoy, copyright(book, "war and peace"))
owns(elvis, car(red, cadillac))

Query:

“Retrieve everything that John owns.”

i.e., Find x such that owns(john, x) is true.

answers: \[ X = \text{car(red, corvette)} \]
          \[ X = \text{cat(black, siamese, sylvester)} \]

Query:

“Retrieve the colour and make of John’s car.”

i.e., owns(john, car(\text{Colour, Make}))

answer: Colour = \text{red}
        Make = \text{corvette}
Prolog: complex types - structures

Same Database:

owns(john, car(red, corvette))
owns(john, cat(black, siamese, sylvester))
owns(elvis, copyright(song,"jailhouse rock"))
owns(tolstoy, copyright(book,"war and peace"))
owns(elvis, car(red, cadillac))

Query: "Who owns a red car?"
\[ i.e., \text{Find values for who so that} \]
\[ \exists \text{Make } owns(Who, car(red, Make)) \text{ is true.} \]

answers: Who = john
           Who = elvis
Prolog: complex types - structures

Same Database:

owns(john, car(red,corvette))
owns(john, cat(black,siamese,sylvester))
owns(elvis, copyright(song,"jailhouse rock"))
owns(tolstoy, copyright(book,"war and peace"))
owns(elvis, car(red,cadillac))

Query: "Who owns a copyright?"

i.e., Find values for Who so that

\[ \exists X, Y \text{ owns(Who, copyright(X,Y)) is true.} \]

answers: Who = elvis
          Who = tolstoy