# Logic Programming (PLP 11.3)

## Horn Clauses

## Introduction to Prolog: Resolution, Unification

Carlos Varela
Rennselaer Polytechnic Institute

November 9, 2006



# An Early (1971) "Conversation"

USER:
Cats kill mice.
Tom is a cat who does not like mice who eat cheese.
Jerry is a mouse who eats cheese.
Max is not a mouse.
What does Tom do?
COMPUTER:
Tom does not like mice who eat cheese.
Tom kills mice.
USER:
Who is a cat?
COMPUTER:
Tom.

USER:
What does Jerry eat?
COMPUTER:
Cheese.
USER:
Who does not like mice who eat cheese?
COMPUTER:
Tom.
USER:
What does Tom eat?
COMPUTER:
What cats who do not like mice who eat cheese eat.



# Another Conversation

USER:
Every psychiatrist is a person.
Every person he analyzes is sick.
Jacques is a psychiatrist in Marseille.
Is Jacques a person?
Where is Jacques?
Is Jacques sick?

COMPUTER:
Yes.
In Marseille.
I don't know.



# Logic programming

- A program is a collection of *axioms*, from which theorems can be proven.
- A *goal* states the theorem to be proved.
- A logic programming language implementation attempts to satisfy the goal given the axioms and built-in inference mechanism.



# Horn clauses

- A standard form for writing axioms, e.g.:

```
father(X,Y) ⇐ parent(X,Y), male(X).
```

- The Horn clause consists of:
  - A *head* or consequent term H, and
  - A *body* consisting of terms $B_i$

$$H \Leftarrow B_0 , B_1 , \ldots, B_n$$

- The semantics is:

« If $B_0$ , $B_1$ , …, $B_n$, then H »



# Terms

- Constants

```
rpi
troy
```

- Variables

```
University
City
```

- Predicates

```
located_at(rpi,troy)
pair(a, pair(b,c))
```

Can be nested.

# Resolution

- To derive new statements, Robinson's resolution principle says that if two Horn clauses:

$$H_1 \Leftarrow B_{11}, B_{12}, \ldots, B_{1m}$$
$$H_2 \Leftarrow B_{21}, B_{22}, \ldots, B_{2n}$$

are such that $H_1$ matches $B_{2i}$, then we can replace $B_{2i}$ with $B_{11}, B_{12}, \ldots, B_{1m}$:

$$H_2 \Leftarrow B_{21}, B_{22}, \ldots, B_{2(i-1)}, \underline{B_{11}, B_{12}, \ldots, B_{1m}}, B_{2(i+1)} \ldots, B_{2n}$$

- For example:

$$\begin{array}{l} C \Leftarrow A,B \\ D \Leftarrow C \\ \hline D \Leftarrow A,B \end{array}$$



# Resolution Example

```
father(X,Y) :- parent(X,Y), male(X).
ancestor(X,Y) :- father(X,Y).
---------------------------------------
ancestor(X,Y) :- parent(X,Y), male(X).
```

`:-` is Prolog's notation for $\Leftarrow$.



# Unification

- During *resolution*, free variables acquire values through *unification* with expressions in matching terms.
- For example:

```
male(carlos).
parent(carlos, tatiana).
father(X,Y) :- parent(X,Y), male(X).
-------------------------------------
father(carlos, tatiana).
```



# Unification Process

- A constant unifies only with itself.
- Two predicates unify if and only if they have
  - the same *functor*,
  - the same number of *arguments*, and
  - the corresponding arguments *unify*.
- A variable unifies with anything.
  - If the other thing has a *value*, then the variable is *instantiated*.
  - If it is an *uninstantiated variable*, then the two variables are *associated*.

✓ Just like compatible assignment of single-assignment variables in Oz.



# Prolog lists

- `[a,b,c ]` is syntactic sugar for:

  `.(a,.(b,.(c, [])))`

  where `[]` is the empty list, and `.` is a built-in cons-like functor.

- `[a,b,c ]` can also be expressed as:

  `[a | [b,c ]]` , or
  `[a, b | [c ]]` , or
  `[a,b,c | []]`



# Prolog lists append example

```
append([],L,L).
append([H|T ],A, [H,L ]) :- append(T,A,L).
```

# Backtracking

- *Forward chaining* goes from axioms forward into goals.
- *Backward chaining* starts from goals and works backwards to prove them with existing axioms.



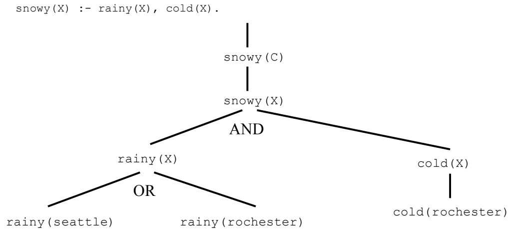
# Backtracking example

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```



# Backtracking example

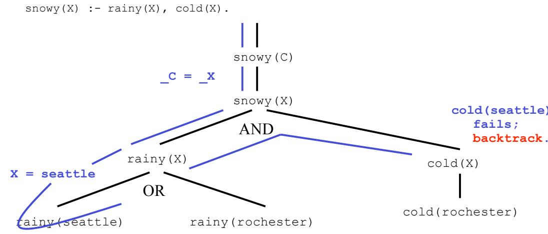
```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```






# Backtracking example

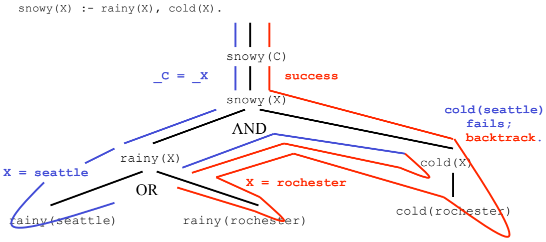
```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```






# Backtracking example

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```






# Propositional Logic

- Assigning truth values to logical propositions.
- Formula syntax:

| | | | |
|---|---|---|---|
| f | ::= | v | symbol |
| | \| | f ∧ f | and |
| | \| | f ∨ f | or |
| | \| | f ⇔ f | if and only if |
| | \| | f ⇒ f | implies |
| | \| | ¬f | not |

## Truth Values

- To assign a truth values to a propositional formula, we have to assign truth values to each of its atoms (symbols).
- Formula semantics:

| a | b | $a \wedge b$ | $a \vee b$ | $a \Leftrightarrow b$ | $a \Rightarrow b$ | $\neg a$ |
|---|---|---|---|---|---|---|
| False | False | F | F | T | T | T |
| False | True | F | T | F | T | T |
| True | False | F | T | F | F | F |
| True | True | T | T | T | T | F |



## Tautologies

- A *tautology* is a formula, true for all possible assignments.
- For example: $\neg\neg p \Leftrightarrow p$
- The contrapositive law:

$$(p \Rightarrow q) \Leftrightarrow (\neg q \Rightarrow \neg p)$$

- De Morgan's law:

$$\neg(p \wedge q) \Leftrightarrow (\neg p \vee \neg q)$$



## First Order Predicate Calculus

- Adds variables, terms, and (first-order) quantification of variables.
- Predicate syntax:

| | | | |
|---|---|---|---|
| a | ::= | $p(v_1, v_2, \ldots, v_n)$ | predicate |
| f | ::= | a | atom |
| | \| | $v = p(v_1, v_2, \ldots, v_n)$ | equality |
| | \| | $v_1 = v_2$ | |
| | \| | $f \wedge f$ \| $f \vee f$ \| $f \Leftrightarrow f$ \| $f \Rightarrow f$ \| $\neg f$ | |
| | \| | $\forall v.f$ | universal quantifier |
| | \| | $\exists v.f$ | existential quantifier |



## Predicate Calculus

- In mathematical logic, a *predicate* is a function that maps constants or variables to true and false.
- Predicate calculus enables reasoning about propositions.
- For example:

$$\forall C\ [\mathrm{rainy}(C) \wedge \mathrm{cold}(C) \Rightarrow \mathrm{snowy}(C)]$$

Quantifier
($\forall$, $\exists$)

Operators
($\wedge, \vee, \Leftrightarrow, \Rightarrow, \neg$)



## Quantifiers

- *Universal* ($\forall$) quantifier indicates that the proposition is true for **all** variable values.
- *Existential* ($\exists$) quantifier indicates that the proposition is true for **at least one** value of the variable.
- For example:

$$\forall A\ \forall B\ [\exists C\ [\ \mathrm{takes}(A,C) \wedge \mathrm{takes}(B,C)] \Rightarrow \mathrm{classmates}(A,B)\ ]$$



## Structural Congruence Laws

$$P_1 \Rightarrow P_2 \equiv \neg P_1 \vee P_2$$

$$\neg\exists X\ [P(X)] \equiv \forall X\ [\neg P(X)]$$
$$\neg\forall X\ [P(X)] \equiv \exists X\ [\neg P(X)]$$

$$\neg(P_1 \wedge P_2) \equiv \neg P_1 \vee \neg P_2$$
$$\neg(P_1 \vee P_2) \equiv \neg P_1 \wedge \neg P_2$$
$$\neg\neg P \equiv P$$

$$(P_1 \Leftrightarrow P_2) \equiv (P_1 \Rightarrow P_2) \wedge (P_2 \Rightarrow P_1)$$

$$P_1 \vee (P_2 \wedge P_3) \equiv (P_1 \vee P_2) \wedge (P_1 \vee P_3)$$
$$P_1 \wedge (P_2 \vee P_3) \equiv (P_1 \wedge P_2) \vee (P_1 \wedge P_3)$$

$$P_1 \vee P_2 \equiv P_2 \vee P_1$$

## Clausal Form

- Looking for a *minimal kernel* appropriate for theorem proving.
- Propositions are transformed into normal form by using structural congruence relationship.
- One popular normal form candidate is *clausal form*.
- Clocksin and Melish (1994) introduce a 5-step procedure to convert first-order logic propositions into clausal form.



## Clocksin and Melish Procedure

1. Eliminate implication (⇒) and equivalence (⇔).
2. Move negation (¬) inwards to individual terms.
3. *Skolemization*: eliminate existential quantifiers (∃).
4. Move universal quantifiers (∀) to top-level and make implicit , i.e., all variables are universally quantified.
5. Use distributive, associative and commutative rules of ∨, ∧, and ¬, to move into *conjuctive normal form*, i.e., a conjuction of disjunctions (or *clauses*.)



## Example

```
∀A [¬student(A) ⇒ (¬dorm_resident(A) ∧
                   ¬∃B [takes(A,B) ∧ class(B)])]
```

1. Eliminate implication (⇒) and equivalence (⇔).

```
∀A [student(A) ∨ (¬dorm_resident(A) ∧
                  ¬∃B [takes(A,B) ∧ class(B)])]
```

2. Move negation (¬) inwards to individual terms.

```
∀A [student(A) ∨ (¬dorm_resident(A) ∧
                  ∀B [¬(takes(A,B) ∧ class(B))])]

∀A [student(A) ∨ (¬dorm_resident(A) ∧
                  ∀B [¬takes(A,B) ∨ ¬class(B)])]
```



## Example Continued

```
∀A [student(A) ∨ (¬dorm_resident(A) ∧
                  ∀B [¬takes(A,B) ∨ ¬class(B)])]
```

3. *Skolemization*: eliminate existential quantifiers (∃).
4. Move universal quantifiers (∀) to top-level and make implicit , i.e., all variables are universally quantified.

```
student(A) ∨ (¬dorm_resident(A) ∧
              (¬takes(A,B) ∨ ¬class(B))
```

5. Use distributive, associative and commutative rules of ∨, ∧, and ¬, to move into *conjuctive normal form*, i.e., a conjuction of disjunctions (or *clauses*.)

```
(student(A) ∨ ¬dorm_resident(A)) ∧
(student(A) ∨ ¬takes(A,B) ∨ ¬class(B))
```



## Clausal Form to Prolog

```
(student(A) ∨ ¬dorm_resident(A)) ∧
(student(A) ∨ ¬takes(A,B) ∨ ¬class(B))
```

6. Use commutativity of ∨ to move negated terms to the right of each clause.
7. Use $P_1 \lor \neg P_2 \equiv P_2 \Rightarrow P_1 \equiv P_1 \Leftarrow P_2$

```
(student(A) ⇐ dorm_resident(A)) ∧
(student(A) ⇐ ¬(¬takes(A,B) ∨ ¬class(B))
```

8. Move Horn clauses to Prolog:

```
student(A) :- dorm_resident(A).
student(A) :- takes(A,B),class(B).
```



## Skolemization

```
∃X [takes(X,cs101) ∧ class_year(X,2)]
```

introduce a Skolem constant to get rid of existential quantifier (∃):

```
takes(x,cs101) ∧ class_year(x,2)
```

```
∀X [¬dorm_resident(X) ∨
    ∃A [campus_address_of(X,A)]]
```

introduce a Skolem function to get rid of existential quantifier (∃):

```
∀X [¬dorm_resident(X) ∨
    campus_address_of(X,f(X)]
```

## Limitations

- If more than one non-negated (positive) term in a clause, then it cannot be moved to a Horn clause (which restricts clauses to only one head term).
- If zero non-negated (positive) terms, the same problem arises (Prolog's inability to prove logical negations).
- For example:
  - « every living thing is an animal or a plant »

```
animal(X) ∨ plant(X) ∨ ¬living(X)
animal(X) ∨ plant(X) ⇐ living(X)
```



## Exercises

70. What is the logical meaning of the second Skolemization example if we do not introduce a Skolem function?
71. Download Prolog and execute the "`snowy(City)`" example. Use "tracing" to follow backtracking step by step.
72. PLP Exercise 11.21 (pg 654).
73. *PLP Exercise 11.23 (pg 654).