Prolog Tutorial

• Relations can be expressed in two ways:
  – Enumeration
  – Rules
  – Combinations of both are possible

• **Highly case-sensitive**
  – Predicates and constants are in lower-case, unless quoted with ‘…’
  – Variables are in upper-case
  – _ is a variable (does not match others)

Prolog Tutorial

```
lives(john, east).
lives(naima, south).
lives(alice, west).
lives(toshiko, east).
lives(roy, north).
lives(albert, south).
```

```
tutors(john, cs, 5).
tutors(naima, cs, 5).
tutors(roy, math, 3).
tutors(alice, math, 55).
tutors(albert, math, 4).
```

Prolog Tutorial

• Typical developmental execution (as opposed to complete application) scenario:
  – Knowledge Base (= Database+Rules) is loaded ("consulted").
  – Queries are posed based on loaded database.
Prolog on Turing
(text in red is typed by user)

```
turing -l prolog
Quintus Prolog Release 3.2 (Sun 4, SunOS 5.3)
Copyright (C) 1994, Quintus Corporation. All rights reserved.
Licensed to Harvey Mudd College, CS Dept.
| ?- consult(tutors).
  % tutors.pl contains the database
  % compiling file /home/keller/tutors.pl
| ?- lives(john, X).
  % Where does john live?
  X = east
| ?- lives(X, east).
  % Who lives in east?
  X = john ;
  X = toshiko ;
  no
```

Prolog Rule Syntax

- A clause such as

  ```
lives(john, east).
  ```

  is called a unit clause or fact; it refers to one piece of information.

- Non-unit clauses typically are in the form of reverse implications:

  ```
  Consequent :- Antecedent.
  ```

  which stands for

  Antecedent implies Consequent

Consequent :- Antecedent

- Can be read as any of the following:

  Antecedent implies Consequent

  Consequent is implied by Antecedent

  Consequent if Antecedent

  Consequent provided Antecedent

- Called the logical interpretation of a clause.

Non-Unit Clause Examples

- `livesInEast(X) :- lives(X, east).`

  variables

- `canTutor(X, Y) :-
  tutors(X, Dept, Number),
  takes(Y, Dept, Number).`

  comma in this context is and

Variables occurring in the consequent are implicitly universally (i.e. $\forall$ ) quantified.
Existential Variables

A variable occurring on the right but not on the left is implicitly existentially (i.e. $\exists$) quantified.

knows(X, Y) :-
    lives(X, Z),
    lives(Y, Z).

Means "(For all X, Y), if there is some Z such that lives(X, Z) and lives(Y, Z), then knows(X, Y)."

Multiple Clauses for One Predicate

Expressing two different ways for X to know Y:

knows(X, Y) :-
    lives(X, Z),
    lives(Y, Z).

knows(X, Y) :-
    takes(X, Dept, Number),
    tutors(Y, Dept, Number).

Logic of Passing an Exam

- There are two ways for a person X to pass an exam:
  - X is adequately prepared, or
  - the exam is extremely easy

pass_exam(X) :- prepared_for_exam(X).

pass_exam(X) :- easy_exam, person(X).

Preparing for an Exam (1&2)

- There are three ways for X to be prepared:
  - X knows it all
    
    prepared_for_exam(X) :-
    knows_it_all(X).

  - X was tutored by Y, who was prepared for the exam:
    
    prepared_for_exam(X) :-
    tutored_by(X, Y),
    prepared_for_exam(Y).
Preparing for an Exam (3)

- X read the book, attended classes (without sleeping), and worked the problems:

\[
\text{prepared_for_exam}(X) \leftarrow \\
\quad \text{read_book}(X), \\
\quad \text{attended_lectures}(X), \\
\quad \text{\textbackslash+ slept_during_lectures}(X), \\
\quad \text{worked_problems}(X).
\]

Note: \textbackslash+ is Prolog’s version of \textit{not}.

Who passes the exam?

\begin{itemize}
  \item \text{person(mary)}.
  \item \text{person(john)}.
  \item \text{person(tom)}.
  \item \text{person(sally)}.
  \item \text{person(fred)}.
  \item \text{read_book(fred)}.
  \item \text{read_book(mary)}.
  \item \text{attended_lectures(fred)}.
  \item \text{attended_lectures(mary)}.
  \item \text{slept_during_lectures(fred)}.
  \item \text{slept_during_lectures(mary)}.
  \item \text{knows_it_all(tom)}.
  \item \text{tutored_by(john, mary)}.
  \item \text{tutored_by(sally, john)}.
  \item \text{worked_problems(fred)}.
  \item \text{worked_problems(mary)}.
\end{itemize}

Simulating a Logic Circuit

circuit.pl  Tue Dec 13 21:30:01 1989

% an example logic circuit

\[
circuit(A, B, C, D) \leftarrow \\
\quad \text{not}(B, X), \\
\quad \text{and}(A, X, Y), \\
\quad \text{or}(Y, V, W), \\
\quad \text{and}(W, C, V), \\
\quad \text{not}(W, D).
\]

% definitions of logic elements

\[
\text{and}(0, 0, 0). \\
\text{and}(0, 1, 0). \\
\text{and}(1, 0, 0). \\
\text{and}(1, 1, 1). \\
\text{or}(0, 0, 0). \\
\text{or}(0, 1, 1). \\
\text{or}(1, 0, 1). \\
\text{or}(1, 1, 1). \\
\text{not}(0, 1). \\
\text{not}(1, 0).
\]

\begin{tikzpicture}
  \node [draw] (A) at (0,0) {A};
  \node [draw] (B) at (1,0) {B};
  \node [draw] (X) at (2,0) {X};
  \node [draw] (Y) at (3,0) {Y};
  \node [draw] (U) at (2,1) {U};
  \node [draw] (V) at (2,0) {V};
  \node [draw] (W) at (3,1) {W};
  \node [draw] (D) at (4,0) {D};
  \node [draw] (C) at (0,-1) {C};

  \draw [->] (A) -- (U);
  \draw [->] (B) -- (X);
  \draw [->] (X) -- (U);
  \draw [->] (V) -- (W);
  \draw [->] (W) -- (D);
  \draw [->] (C) -- (V);
\end{tikzpicture}

Goal-Oriented (Procedural) Interpretation of Prolog

- The execution of Prolog is actually a form of \textit{depth-first search}.

- A Prolog query is composed of a list of \textbf{goals} (the individual predicate expressions).

- Prolog tries to solve these goals by finding individuals that \textbf{satisfy} the predicates, as determined by the knowledge base.

- During the solving process, a goal is replaced by other goals, according to the \textbf{rules}, until there are no unsolved goals left.
canTutor(X, Y) :-
    tutors(X, Dept, Number),
    takes(Y, Dept, Number).

% lives(N, D) means that person named N lives in dorm D
lives(john, east).
lives(naima, south).
lives(alice, west).
lives(toshiko, east).
lives(roy, north).
lives(albert, south).

% takes(N, D, C) means that person named N takes course C in department D
takes(john, cs, 60).
takes(naima, cs, 60).
takes(alice, cs, 60).
takes(toshiko, cs, 5).
takes(roy, math, 55).
takes(naima, math, 55).
takes(alice, math, 55).
takes(toshiko, math, 5).
takes(albert, math, 55).

tutors(alice, math, 55).
tutors(alice, cs, 5).
tutors(roy, math, 3).
tutors(naima, cs, 5).
tutors(alice, math, 55).
tutors(albert, math, 4).

canTutor(alice, Y).

tutors(alice, Dept, Number),
    takes(Y, Dept, Number).

tutors(alice, math, 55).
takes(Y, math, 55).
Y = naima
(undo)

Deeper Backtracking

canTutor(X, Y) :-
    tutors(X, Dept, Number),
    takes(Y, Dept, Number).

tutors(X, Dept, Number),
    takes(Y, Dept, Number).

tutors(john, cs, 5).
takes(Y, cs, 5).
Y = toshiko
(undo)

 Goal Succession:

Depth-First Execution in Prolog

canTutor(alice, Y).

tutors(alice, Dept, Number),
    takes(Y, Dept, Number).

tutors(alice, math, 55).
takes(Y, math, 55).
Y = roy
(undo)

Backtracking in Depth-First Search

canTutor(alice, Y).

tutors(alice, Dept, Number),
    takes(Y, Dept, Number).

tutors(alice, math, 55).
takes(Y, math, 55).
Y = naima
(undo)
Deeper Backtracking

\[ \text{canTutor}(X, Y). \]

\[ \text{canTutor}(X, Y) : \]
\[ \text{tutors}(X, \text{Dept}, \text{Number}), \]
\[ \text{takes}(Y, \text{Dept}, \text{Number}). \]

\[ \text{tutors}(X, \text{Dept}, \text{Number}), \]
\[ \text{takes}(Y, \text{Dept}, \text{Number}). \]

\[ \text{tutors}(\text{naima}, \text{cs}, 5). \]
\[ \text{takes}(\text{Y}, \text{cs}, 5). \]
\[ \text{Y} = \text{toshiko} \]
\[ \text{(empty)} \]

\[ \text{result binding} \]

\[ \text{X} = \text{naima} \]
\[ \text{Dept} = \text{cs} \]
\[ \text{Number} = 5 \]

\[ \text{undo former binding \}; try for another result} \]

\[ \text{tutors}(\text{toshiko}, \text{cs}, 5). \]
\[ \text{result binding} \]

\[ \text{Deeper Backtracking} \]

\[ \text{canTutor}(X, Y). \]

\[ \text{canTutor}(X, Y) : \]
\[ \text{tutors}(X, \text{Dept}, \text{Number}), \]
\[ \text{takes}(Y, \text{Dept}, \text{Number}). \]

\[ \text{tutors}(X, \text{Dept}, \text{Number}), \]
\[ \text{takes}(Y, \text{Dept}, \text{Number}). \]

\[ \text{tutors}(\text{roy}, \text{math}, 3). \]
\[ \text{takes}(\text{Y}, \text{math}, 3). \]
\[ \text{fails} \]

\[ \text{Deeper Backtracking} \]

\[ \text{canTutor}(X, Y). \]

\[ \text{canTutor}(X, Y) : \]
\[ \text{tutors}(X, \text{Dept}, \text{Number}), \]
\[ \text{takes}(Y, \text{Dept}, \text{Number}). \]

\[ \text{tutors}(X, \text{Dept}, \text{Number}), \]
\[ \text{takes}(Y, \text{Dept}, \text{Number}). \]

\[ \text{tutors}(\text{alice}, \text{math}, 55). \]
\[ \text{takes}(\text{Y}, \text{math}, 55). \]
\[ \text{X} = \text{alice} \]
\[ \text{Dept} = \text{math} \]
\[ \text{Number} = 55 \]

\[ \text{etc.} \]

\[ \text{Summary of Backtracking} \]

- Given a goal, Prolog tries, in order of occurrence, the first rule, the consequent of which matches the goal.

- If the rule has sub-goals, the sub-goals are satisfied in order of occurrence, resulting in bindings at each stage.

- If a goal or sub-goal fails, Prolog retries to satisfy it using the next available option (e.g. the next rule).