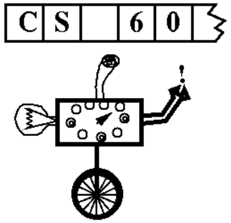


Prolog

November 19, 2001

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

Enumeration of the *lives* relation in Prolog:

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

called “clauses”

Enumeration of the *tutors* relation in Prolog:

```
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 ~:1> 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).
```

variable

individual

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

comma in this context is ***and***.

Variables occuring 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. ∃) 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).
```

This is like a proposition variable.
It is equivalent to a predicate with
no arguments, and **parens are not used**.

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:

```
prepared_for_exam(X) :-
    read_book(X),
    attended_lectures(X),
    \+ slept_during_lectures(X),
    worked_problems(X).
```

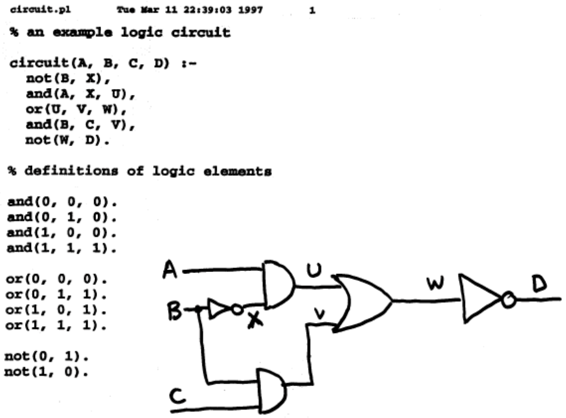
Note: \+ is Prolog’s version of *not*.

Who passes the exam?

```
person(mary).
person(john).
person(tom).
person(sally).
person(fred).

read_book(fred).
read_book(mary).

worked_problems(fred).
worked_problems(mary).
```

```
attended_lectures(fred).
attended_lectures(mary).

slept_during_lectures(fred).

knows_it_all(tom).

tutored_by(john, mary).
tutored_by(sally, john).
```

Simulating a Logic Circuit



```
circuit.pl       Tue Mar 11 22:39:03 1997       1

% an example logic circuit

circuit(A, B, C, D) :-
  not(B, X),
  and(A, X, U),
  or(U, V, W),
  and(B, C, V),
  not(W, D).

% definitions of logic elements

and(0, 0, 0).
and(0, 1, 0).
and(1, 0, 0).
and(1, 1, 1).

or(0, 0, 0).
or(0, 1, 1).
or(1, 0, 1).
or(1, 1, 1).

not(0, 1).
not(1, 0).
```

Goal-Oriented (Procedural) Interpretation of Prolog

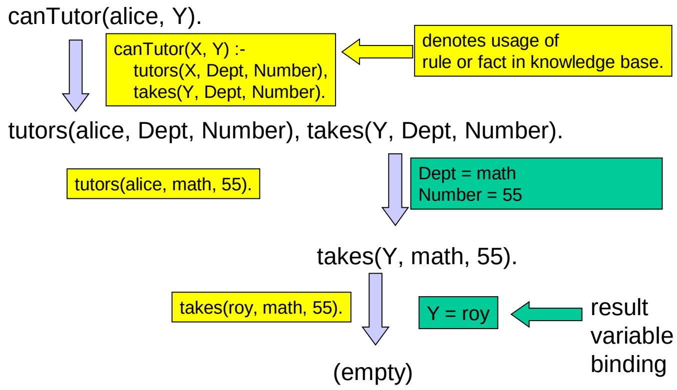
- The execution of Prolog is actually a form of **depth-first search**.
- A Prolog query is composed of a list of **goals** (the individual predicate expressions).
- Prolog tries to solve these goals by finding individuals that **satisfy** the predicates, as determined by the knowledge base.
- During the solving process, a goal is replaced by other goals, according to the **rules**, until there are no unsolved goals left.

Previous Example Prolog KB

```
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(albert,  cs,   60).
takes(roy,     math, 55).
takes(naima,   math, 55).
takes(alice,   math, 70).
takes(toshiko, math, 80).
takes(albert,  math, 55).

% tutors(N, D, C) means that person named N tutors course C in department D
tutors(john,   cs,    5).
tutors(naima,  cs,    5).
tutors(roy,    math,  3).
tutors(alice,  math, 55).
tutors(albert, math,  4).
```

Goal Succession:

Depth-First Execution in Prolog



canTutor(alice, Y).

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

denotes usage of
rule or fact in knowledge base.

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

tutors(alice, math, 55).

Dept = math
Number = 55

takes(Y, math, 55).

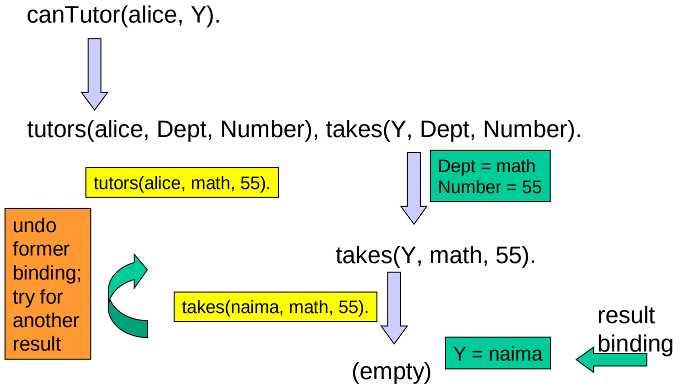
takes(roy, math, 55).

Y = roy

result
variable
binding

(empty)

Backtracking in Depth-First Search



canTutor(alice, Y).

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

tutors(alice, math, 55).

Dept = math
Number = 55

undo
former
binding;
try for
another
result

takes(Y, math, 55).

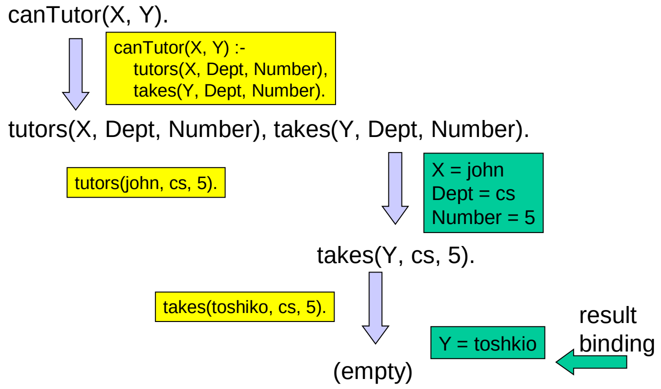
takes(naima, math, 55).

Y = naima

result
binding

(empty)

Deeper Backtracking



canTutor(X, Y).

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

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

tutors(john, cs, 5).

X = john
Dept = cs
Number = 5

takes(Y, cs, 5).

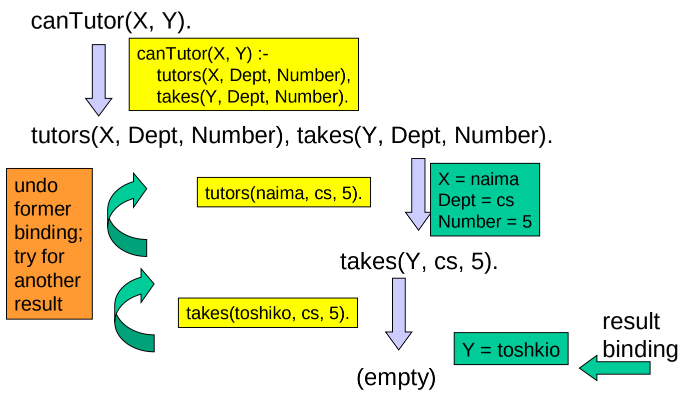
takes(toshiko, cs, 5).

Y = toshkio

result
binding

(empty)

Deeper Backtracking



canTutor(X, Y).

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

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

undo former binding; try for another result

tutors(naima, cs, 5).

X = naima
Dept = cs
Number = 5

takes(Y, cs, 5).

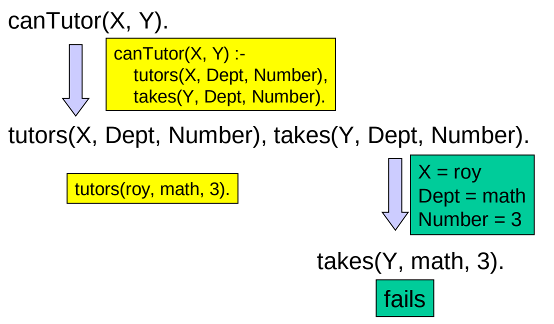
takes(toshiko, cs, 5).

Y = toshkio

result binding

(empty)

Deeper Backtracking



canTutor(X, Y).

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

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

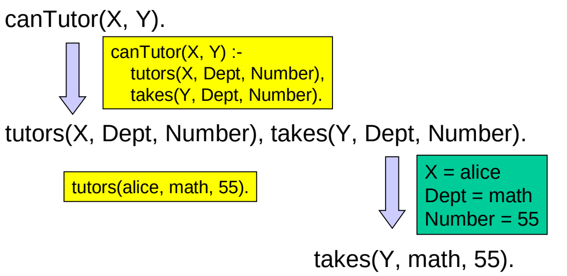
tutors(roy, math, 3).

X = roy
Dept = math
Number = 3

takes(Y, math, 3).

fails

Deeper Backtracking



canTutor(X, Y).

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

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

tutors(alice, math, 55).

X = alice
Dept = math
Number = 55

takes(Y, math, 55).

etc.

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).