

---

## Prolog Overview

Yves Lespérance  
(Some material comes from Peter  
Roosen-Runge)

---

## Prolog idea

- ◆ programming language based on first-order Horn theories, SLD resolution
- ◆ search strategy is fixed: depth-first, left to right, top to bottom
- ◆ programmer uses this to order search, is responsible for efficiency and termination
- ◆ good for symbolic computing

## **syntax of terms**

---

- ◆ variables begin with upper-case letter or \_
- ◆ constants and functors (function and predicate symbols) begin with lower-case
- ◆ E.g. john, john\_smith, X, Node, \_person, 'CSE', fatherOf(paul), date(25,10,2005)
- ◆ compound terms are called structures, e.g. course(complexity,time(monday,9,11),lecturer(patrick,dymond),location('CSE',3311))

## **E.g. program: family relations**

---

- ◆ rules
  - parent(Parent, Child) :- mother(Parent, Child).
  - parent(Parent, Child) :- father(Parent, Child).
  - ancestor(X,Y) :- parent(X,Y).
  - ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
- ◆ facts
  - father('George', 'Elizabeth').
  - father('George', 'Margaret').
  - father('Paul', 'George').
  - mother('Mary', 'Elizabeth').
  - mother('Mary', 'Margaret').

## rules

---

- ◆ *rules* are definite clauses, or conditional statements.
- ◆ e.g.  
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).  
i.e.  $\forall x \forall y \forall z (\text{Ancestor}(z,y) \wedge \text{Parent}(x,z) \supset \text{Ancestor}(x,y))$  or  
[Ancestor(x,y),  $\neg$ Ancestor(x,y),  
 $\neg$ Ancestor(x,y)].
- ◆ , represents conjunction and :- represents implication.

## rules

---

- ◆ variables are universally quantified from outside; can think of variables that appear only in rule body as existentially quantified.
- ◆ a program is a set of rules/definite clauses.
- ◆ ; represents disjunction, e.g.  
parent(Parent, Child) :- mother(Parent, Child);  
father(Parent, Child).

## facts

---

- ◆ *facts* are a special case of rules, definite clauses with no negative literals, i.e. atomic formulas.
- ◆ e.g. `father('George', 'Elizabeth')`.

## queries

---

- ◆ a query asks whether a (conjunction of) atomic formula is entailed by the program.
- ◆ `?- parent(X, 'Elizabeth')`.  
`X = 'Mary'`  
Yes
- ◆ this asks whether  
`Program |=  $\exists x$  Parent(x, Elizabeth)` or  
`Program  $\cup$  { $\forall x \neg$ Parent(x, Elizabeth)} |- []`.
- ◆ variables in queries can be viewed as existentially quantified, can be used to retrieve information.

## simpler family relations e.g.

---

- ◆ rules
  - parent(Parent, Child) :- mother(Parent, Child).
  - parent(Parent, Child) :- father(Parent, Child).
- ◆ facts
  - father('George', 'Elizabeth'). father('George', 'Margaret').
  - mother('Mary', 'Elizabeth'). mother('Mary', 'Margaret').

## unification

---

- ◆ unification is used to match queries with facts or the head or rules
- ◆ no fixed input or output parameters
- ◆ ?- parent('Mary', X).  
X = 'Elizabeth'  
Yes

## finding all solutions

---

```
| ?- parent(Parent, Child).  
Parent = 'Mary',  
Child = 'Elizabeth' ;  
  
Parent = 'Mary',  
Child = 'Margaret' ;  
  
Parent = 'George',  
Child = 'Elizabeth' ;  
  
Parent = 'George',  
Child = 'Margaret' ;  
  
no
```

## search strategy/control

---

- ◆ Prolog searches to find a SLD resolution derivation of [] from the query.
- ◆ it works on the literals in the query from left to right.
- ◆ it resolves the first literal in the query against the first rules that matches, and the instantiated body of the rule replaces that literal in the query
- ◆ if eventually [] is derived, the query succeeds and the instantiation of the variables is returned.
- ◆ if at some point in the search no rule matches, the current query fails and Prolog backtracks to that last rule choice, and tries the next rule that matches.
- ◆ amounts to backward chaining, depth-first, left to right search.

## rules as procedures

---

- ◆ rule has form goal :- body
- ◆ goal or head is like name of procedure
- ◆ terms on the RHS are like the body of the procedure, the sub-goals that have to be achieved to show that the goal holds
- ◆ the sub-goals will be attempted left-to-right
- ◆ rule succeeds if all sub-goals succeed

## how prolog finds solutions

---

```
[trace] ?-
  parent(Parent, Child1),
  parent(Parent, Child2),
  not(Child1 = Child2).
Call: (8) parent(_G313, _G314)
? creep
Call: (9) mother(_G313, _G314) ?
creep
Exit: (9) mother('Mary',
'Elizabeth') ? creep
Exit: (8) parent('Mary', 'Elizabeth')
? creep
Exit: (9) mother('Mary', 'Elizabeth')
? creep
Exit: (8) parent('Mary', 'Elizabeth')
? creep
Call: (8) parent('Mary', _G317) ?
creep
Call: (9) mother('Mary', _G317) ?
creep
Exit: (9) mother('Mary',
'Elizabeth') ? creep
Exit: (8) parent('Mary', 'Elizabeth')
? creep
Redo: (9) mother('Mary', _G317) ?
creep
Exit: (9) mother('Mary',
'Margaret') ? creep
Exit: (8) parent('Mary', 'Margaret')
? creep
Parent = 'Mary'
Child1 = 'Elizabeth'
Child2 = 'Margaret'
```

## search control

---

- ◆ programmer can control search by ordering rules and goals in the body of rules.
- ◆ also can use ! (cut) as explained in textbook.
- ◆ not (negation as failure) can also be used to have a query succeed if another fails.

## arithmetic functions

---

- ◆ Prolog retains arithmetic functions as functions (more intuitive):  
?- X **is** exp(1). % exp(1) = e<sup>1</sup>  
X = 2.71828  
Yes  
?- X is (4 + 2) \* 5.  
X = 30  
Yes
- ◆ How does is compare with =, assignment?

## operators

---

- ◆ some functors are represented by *infix* or *prefix* or *postfix* operators
- ◆ some infix operators: `is`, `=`, `+`, `*`, `/`, `mod`, `>`, `>=`, `:-`, ``,``, etc.
- ◆ `+` and `-` are both prefix and infix
- ◆ `:-` as prefix is a command, used for declarations
- ◆ operators have precedence
- ◆ can define our own operators

## arithmetic examples

---

`factorial(0,1).`

`factorial(N,M):- K is N -1, factorial(K,L),  
M is N * L.`

`min(X,Y,X):- X =< Y, !.`

`min(X,Y,Y).`

## lists

---

- ◆ lists are a special kind of term that allows arbitrary number of components
- ◆ [] is the empty list
- ◆ .(a,b) is a dotted pair
- ◆ [a, b, c] = .(a,.(b,.(c,[]))) is a list of 3 components.
- ◆ the functor . builds binary trees (as in Lisp)
- ◆ can use display(X) to print internal representation of X

## lists

---

- ◆ can refer to the first and rest of a list using the notation: [First | Rest]
- ◆ e.g. ?- X = [a,b,c], X = [F|R].  
X = [a,b,c]  
F = a  
R = [b,c]
- ◆ E.g. X = [b], Y = a, Z = [Y|X].  
X = [b]  
Y = a  
Z = [a,b]

## e.g. append predicate

---

```
append([],L,L).  
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

```
?- append([a,b],[c],X).  
X = [a, b, c]
```

```
Yes  
?- append(X,[c],[a,b,c]).  
X = [a, b]
```

```
Yes  
?- append([a,b],[c],[a,b,d]).
```

```
No
```

## more append examples

---

```
?- append([a,b],X,Y).  
X = _G187  
Y = [a, b|_G187]  
Yes  
?- append(X,Y,Z).  
X = []  
Y = _G181  
Z = _G181 ;
```

```
X = [_G262]  
Y = _G181  
Z = [_G262|_G181] ;
```

```
X = [_G262, _G268]  
Y = _G181  
Z = [_G262, _G268|_G181]
```

append is an example of a reversible or **steadfast** predicate (Richard O'Keefe)

## building a knowledge base

---

- ◆ to be used in a computation, facts and rules must be stored in the (dynamic) database
- ◆ facts and rules get into the database through *assertion* and *consultation*
- ◆ consultation loads facts and rules from a file

## assertion

---

- ◆ ?- assert(human(ulyssus)).
- ◆ ?- human(X).  
X = ulyssus  
Yes
- ◆ assertion can be done dynamically
- ◆ also retract to remove facts and rules from the DB
- ◆ like assignment, change state; avoid when possible

## consultation

---

- ◆ ?- consult('family.pl').  
loads facts and rules from file family.pl
- ◆ ?- [family].  
does the same thing
- ◆ ?- [user].  
lets you enter facts and rules from the keyboard

## help is sometimes helpful

---

?- help(reverse).

reverse(+List1, -List2)

Reverse the order of the elements in List1 and unify the result with the elements of List2.

+arg: arg is input and should be instantiated.

-arg: arg is output and can be initially uninstantiated; if the query succeeds, the arg is instantiated with the "output" of the query.

?arg: arg can be either input or output

## online help

---

?- help(lists).

No help available for lists

Yes

?- apropos(lists).

merge/3

Merge two sorted lists

append/3

Concatenate lists

Section 11-1

"lists: List Manipulation"

Section 15-2-1

"lists"

Yes

?- help(append/3).

append(?List1, ?List2, ?List3)

Succeeds when List3 unifies with the concatenation of List1 and List2. The predicate can be used with any instantiation pattern (even three variables).

---

e.g. solving a logic puzzle

## the zebra puzzle

---

1. There are 5 houses, occupied by politically-incorrect gentlemen of 5 different nationalities, who all have different coloured houses, keep different pets, drink different drinks, and smoke different (now-extinct) brands of cigarettes.
2. The Englishman lives in a red house.
3. The Spaniard keeps a dog.
4. The owner of the green house drinks coffee.
- ...
6. The ivory house is just to the left of the green house.
- ...
11. The Chesterfields smoker lives next to a house with a fox.

Who owns the zebra and who drinks water?

## Prolog implementation

---

- ◆ represent the 5 houses by a structure of 5 terms  
house(Colour, Nationality, Pet, Drink, Cigarettes)
- ◆ create a partial structure using variables, to be filled by the solution process
- ◆ specify constraints to instantiate variables

## house building

---

```
makehouses(0,[]).
```

```
makehouses(N,[house(Col, Nat, Pet, Drk, Cig)|List])  
:- N>0, N1 is N - 1, makehouses(N1,List).
```

or more cleanly with anonymous variables:

```
makehouses(N,[house(_ , _ , _ , _ , _)|List])  
:- N>0, N1 is N - 1, makehouses(N1,List).
```

## the empty houses

---

```
?- makehouses(5, List).
```

```
List = [house(_G233, _G234, _G235, _G236, _G237),  
house(_G245, _G246, _G247, _G248, _G249),  
house(_G257, _G258, _G259, _G260, _G261),  
house(_G269, _G270, _G271, _G272, _G273),  
house(_G281, _G282, _G283, _G284, _G285)]
```

## constraints

---

- ◆ The Englishman lives in a red house.  
house(red, englishman, \_, \_, \_) on List,
- ◆ The Spaniard keeps a dog.  
house( \_, spaniard, dog, \_, \_) on List,
- ◆ The owner of the green house drinks coffee.  
house(green, \_, \_, coffee, \_) on List
- ◆ The ivory house is just to the left of the green house  
sublist2([house(ivy, \_, \_, \_, \_), house(green, \_, \_, \_, \_)], List),
- ◆ The Chesterfields smoker lives next to a house with a fox.  
nextto(house( \_, \_, \_, \_, chesterfields), house( \_, \_, fox, \_, \_), List),

## defining the on operator

---

- ◆ on is a user-defined infix operator that is a version of member/2
- ◆ :- op(100,zfy,on).  
X on List :- member(X,List).  
amounts to  
X on [X|\_].  
X on [\_|R]:- X on R.  
See /cs/dept/course/2005-06/F/3401/zebra.pl

## predicates for defining constraints

---

- ◆ “just to the left of”? “lives next to”?
- ◆ define sublist(S,L)  
sublist2([S1, S2], [S1, S2 | \_]) .  
sublist2(S, [\_ | T]) :- sublist2(S, T).
- ◆ define nextto predicate  
nextto(H1, H2, L) :- sublist2([H1, H2], L).  
nextto(H1, H2 ,L) :- sublist2([H2, H1], L).

## translating the constraints

---

- ◆ The ivory house is just to the left of the green house  
sublist2( [house(ivory, \_, \_, \_),  
house(green, \_, \_, \_)], List),
- ◆ The Chesterfields smoker lives next to a house with a fox.  
nextto(house( \_, \_, \_, chesterfields),  
house( \_, \_, fox, \_), List),

## looking for the zebra

---

- ◆ Who owns the zebra and who drinks water?  
find(ZebraOwner, WaterDrinker) :-  
  makehouses(5, List),  
  house(red, englishman, \_, \_, \_) on List,  
  ... % all other constraints  
  house( \_, WaterDrinker, \_, water, \_) on List,  
  house( \_, ZebraOwner, zebra, \_, \_) on List.
- ◆ solution is generated and queried in the same clause
- ◆ neither water or zebra are mentioned in the constraints

## solving the puzzle

---

```
?- [zebra].  
% zebra compiled 0.00 sec, 5,360 bytes  
  
Yes  
?- find(ZebraOwner, WaterDrinker).  
  
ZebraOwner = japanese  
WaterDrinker = norwegian ;  
  
No
```

## how Prolog finds solution

---

After first 8 constraints:

```
List = [  
house(red, englishman, snail, _G251, old_gold),  
house(green, spaniard, dog, coffee, _G264),  
house(ivory, ukrainian, _G274, tea, _G276),  
house(green, _G285, _G286, _G287, _G288),  
house(yellow, _G297, _G298, _G299, kools)]
```

## how Prolog solves the puzzle

---

Then need to satisfy "the owner of the third house drinks milk", i.e.

```
List = [_, _, house( _, _, _, milk, _),_, _],
```

Can't be done with current instantiation of List. So Prolog will **backtrack** and find another.

## how Prolog solves the puzzle

---

The unique complete solution is

```
L = [  
house(yellow, norwegian, fox, water, kools),  
house(blue, ukrainian, horse, tea, chesterfields),  
house(red, englishman, snail, milk, old_gold),  
house(ivory, spaniard, dog, orange, lucky_strike),  
house(green, japanese, zebra, coffee, parliaments)]  
See /cs/dept/course/2005-06/F/3401/zebra.pl
```