Introduction

- Prolog is a logical programming language and stands for PROgramming in LOGic
- Created around 1972
- Preferred for AI programming and mainly used in such areas as:
  - Theorem proving, expert systems, NLP, …
- Logical programming is the use of mathematical logic for computer programming.

Introduction (Cont'd)

- For symbolic, non-numeric computation
- e.g. : parent (tom, bob).
- Parent is a relation between its parameters: tom and bob
- The whole thing is called a clause
- Each clause declares one fact about a relation

Prolog

- Prolog has an interactive interpreter
- After starting SWI-Prolog, the interpreter can start reading your Prolog files and accept your queries.
- To exit Prolog simply type the command ‘halt.’ (Notice the full-stop)
- Prolog program files usually have the extension .pl or .pro

Statements

- There are three categories of statements in Prolog:
  - Facts: Those are true statements that form the basis for the knowledge base.
  - Rules: Similar to functions in procedural programming (C++, Java…) and has the form of if/then.
  - Queries: Questions that are passed to the interpreter to access the knowledge base and start the program.

Facts

- A fact is a one-line statement that ends with a full-stop.
  - parent (john, bart).
  - parent (barbara, bart).
  - male (john).
  - male (bart).
  - female (barbara).

Rules

- A Rule consists of
  - a condition part (right-hand side) → body of clause
  - a conclusion part (left-hand side) → head of clause
  - They are separated by ':-' which means 'if'
- offspring relation
  - offspring (X, Y) : X is an offspring of Y
  - ∀ X,Y (offspring (X, Y) ← parent (Y, X))
  - offspring (X, Y) :- parent (Y, X).

head body

Rules (Cont'd)

- Variables in head of rules are universally quantified
- Variables appearing only in the body are existentially quantified
- Rules vs. Facts
  - A Fact is something unconditionally true
  - Rules specify things that are true if some condition is satisfied

Queries

- Queries are questions
- The engine tries to entail the query (goal) using the Facts and Rules in KB
- There are two kinds of answer
  - Yes/No: parent (tom, bob).
  - Unified Answer/No: parent (X, bob).
- Other possible answer(s) can be found using semicolon (return for stopping)
- For example : parent (X, Y). →

X=pam
Y=bob;

X=tom
Y=bob;

X=tom
Y=liz;

no

Queries (Cont'd)

- Q: Who is a grandparent of Jim? (using parent relationship)
  - Who is a parent of Jim? Assuming "Y"
  - Who is a parent of "Y"? Assuming "X"
  - ?- parent (Y, jim) , parent (X, Y).
  - If we change the order of them the logical meaning remains the same
- Q: Who are Tom's grandchildren?
- Q: Are Ann and Pat siblings?

Where the program is written?

- Facts and Rules are stored in one or more files forming our Knowledge Base
- Files containing KB are loaded into the interpreter
- After changing these files, the files should be loaded again to be effective
- Queries are asked in the interactive mode in front of the question prompt: ?-

```prolog
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Y) :- parent(X,Y), ancestor(Y,Z).
sibling(X,Y) :- mother(M,X), mother(M,Y),
                father(F,X), father(F,Y), X \= Y.
cousin(X,Y) :- parent(U,X), parent(V,Y), sibling(U,V).

father(albert, jeffrey).
mother(alice, jeffrey).
father(albert, george).
mother(alice, george).
father(john, mary).
mother(sue, mary).
father(george, cindy).
mother(mary, cindy).
father(george, victor).
mother(mary, victor).
```

SWI Prolog

```
?- [kinship].
% kinship compiled 0.00 sec, 3,016 bytes
Yes

?- ancestor(X, cindy), sibling(X, jeffrey).
X = george ↵
Yes

?- grandparent(albert, victor).
Yes

?- cousin(alice, john).
No

?- sibling(A,B).
A = jeffrey, B = george ; ↵
A = george, B = jeffrey ; ↵
A = cindy, B = victor ; ↵
A = victor, B = cindy ; ↵
No
```

Examples

- mother (X, Y) :- parent (X, Y) , female (X).
- sister (X, Y) :-

  parent (Z, X) ,

  parent (Z, Y) ,

  female (X).
- What is wrong with this rule?
- Any female is her own sister
- Solution?

Comments

- Multi-line :

```
/* This is a comment
  This is another comment */
```

- Short :

```
% This is also a comment
```

Reading Files

- consult (filename).
  - Reads and compiles a Prolog source file
  - consult ('/home/user/prolog/sample.pl').
- reconsult (filename).
  - Reconsult a changed source files.
  - reconsult('/home/user/prolog/sample.pl').
- [‘filename’].
  - ['/home/user/prolog/sample.pl'].
- make.
  - Reconsult all changed source files.

Prolog Syntax

- Terms in Prolog:
  - Simple
    - Constants:
      - Atoms
      - Numbers
        - Integer
        - Real
    - Variables
  - Complex Structures

Atoms

- They should consist of the following set of characters:
  - The *upper-case letters*
  - The *lower-case letters*
  - The *digits*
  - The *special characters:* +, -, *, /, <, >, =, :, ., &, ~, _
- Atoms should not start with upper-case letters or underscore and can be followed by any set of characters.
- The scope of an atom is the whole program

Examples of Atoms

- anna, x30, x_, x___y, miss_Jones
- <---> , ==>, … , .:. , ::= (except reserved ones like :-)
- ‘Tom’ , ‘Sarah Jones’ (Useful for having an atom starting with a capital letter)

Numbers

- Integer
  - limited to an interval between some smallest and some largest number permitted by a particular Prolog implementation
  - e.g. : 1, 1001 , 0 , -98
- Real
  - Not frequently used
  - e.g. : 3.14 , -0.0035 , 100.2

Variables

- Consists of letters, digits and '_'
- Starting with an upper-case or an '_'
- The variable '_' (a single underscore character) is a special one. It's called the anonymous variable.
- The scope of a variable is its clause
  - If the name X15 occurs in two clauses, it represents two different variables.
  - Each occurrence of X15 within the same clause means the same variable

Structures

- Compound Objects
- Each constituent is a simple object or structure.
- e.g. : date (1, jan, 2007)
- Components can be variables.
- Any day in Jan 2007 → date (Day, jan, 2007)

Conjunction and Disjunction

- Conjunction → ,
- Disjunction → ;
  - P :- Q ; R.
  - P :- Q
  - P :- R
- ‘,’ has more priority
  - P :- Q , R ; S , T , U .
  - P :- (Q , R) ; (S , T , U) .

Recursion

- Define ancestor relation based on parent relation.
- ancestor (X, Z) :-

  parent (X, Z).
- ancestor (X, Z) :-

  parent (X,Y) , parent (Y, Z).
- ancestor (X, Z) :-

  parent (X, I) , parent (I, Y) , parent (Y, Z).
- Solution is Recursion

Recursion

- Remember from functional programming languages

```
void func (int a , int b)
{
      //base case
        if (condition)
          return;
      …
      // recursion
       func (x, y);
      …
}
```

Recursion

- Rules in Prolog are like functions in procedural programming languages
- For recursion we should define the ancestor relation in terms of itself
- Base Case :
  - ancestor(X, Z) :- parent (X, Z).
- Recursion Step :
  - ancestor (X, Z) :- parent (X, Y) , ancestor (Y, Z).

How Prolog Answers Questions

- Instead of starting with simple facts given in the program, prolog starts with the goals. In fact, Prolog does goal driven search.
- Using rules, Prolog substitutes the current goals (which matches a rule head) with new sub-goals (the rule body), until the new sub-goals happen to be simple facts.
- Prolog returns the first answer matching the query. When prolog discovers that a branch fails or if you type ';' to get other answers, it backtracks to the previous node and tries to apply an alternative rule at that node.

Example

- Facts:
  - parent (pam, bob). parent (tom, bob). parent (tom, liz).
  - parent (bob, ann). parent (bob, pat). parent (pat, jim).
- Rules:
  1. ancestor (X, Z) :- parent (X, Z).
  2. ancestor (X, Z) :- parent (X, Y) , ancestor (Y, Z)
- ?- ancestor (tom, pat). (goal)
- The rule that appears first, is applied first
- Unifying: {tom/X} , {pat/Z}
  - The goal is replaced by : parent (tom, pat). (sub-goal)
- Fails $\Rightarrow$ backtracking

Example (Cont'd)

- Applying the next rule
  2. ancestor (X, Z) :- parent (X, Y) , ancestor (Y, Z)
- Unifying: {tom/X} , {pat/Z}
  - New Goal: parent (tom, Y) , ancestor (Y, pat)
  - Prolog tries to satisfy them in order in which they are written
  - The first one matches one of the facts {bob/Y}
  - Second sub-goal: ancestor (bob, pat)
  - The same steps should be done for this sub-goal

Orders of Clauses and Goals

```
    ancestor (X, Z) :- parent (X, Z).
    ancestor (X, Z) :- parent (X, Y) , ancestor (Y, Z).

2.  ancestor (X, Z) :- parent (X, Y) , ancestor (Y, Z).
    ancestor (X, Z) :- parent (X, Z).

3.  ancestor (X, Z) :- parent (X, Z).
    ancestor (X, Z) :- ancestor (Y, Z) , parent (X, Y).

4.  ancestor (X, Z) :- ancestor (Y, Z) , parent (X, Y).
    ancestor (X, Z) :- parent (X, Z).
```

Orders of Clauses and Goals

- It turns out that :
  - The first and second variations are able to reach and answer for ancestor.
  - The third sometimes can and sometimes can't
  - And the forth can never reach and answer (infinite recursion)
- "Try simple things first".

Lists

- How do you represent the list 1,2,3,4?
- Use a structured term:
  cons(1, cons(2, cons(3, cons(4, nil))))
- Prolog lets you write this more prettily as [1,2,3,4]

cons(1, cons(2, cons(3, cons(4, nil))))

- if X=[3,4], then [1,2|X]=[1,2,3,4]

cons(3,cons(4,nil)) cons(1,cons(2,X))

Lists

- How do you represent the list 1,2,3,4?
- Use a structured term:
  cons(1, cons(2, cons(3, cons(4, nil))))
- Prolog lets you write this more prettily as [1,2,3,4]

- $\overbrace{[1,2,3,4]}^{\text{cons(1, cons(2, cons(3, cons(4, nil))))}}=\underbrace{[1,2|X]}_{\text{cons(1,cons(2,X))}} \rightarrow \underbrace{X=[3,4]}_{\text{cons(3,cons(4,nil))}}$ by unification

Lists

- How do you represent the list 1,2,3,4?
- Use a structured term:
  cons(1, cons(2, cons(3, cons(4, nil))))
- Prolog lets you write this more prettily as [1,2,3,4]
- $\overbrace{[1,2]}^{\text{cons(1, cons(2, nil))}} \quad = \underbrace{[1,2|X]}_{\text{cons(1,cons(2,X))}} \rightarrow \underbrace{X=[\,]}_{\text{nil}}$

Decomposing lists

- first(X,List) :- ...?

- first(X,List) :- List=[X|Xs].
  - Traditional variable name:
    "X followed by some more X's."
- first(X, [X|Xs]).
  - Nicer: eliminates the single-use variable List.
- first(X, [X|_]).
  - Also eliminate the single-use variable Xs.

Decomposing lists

- first(X,   [X|_]).
- rest(Xs, [_|Xs]).

- Query: first(8, [7,8,9]).
  - Answer: no
- Query: first(X, [7,8,9]).
  - Answer: X=7
- Query: first(7, List).
  - Answer: List=[7|Xs]
    (will probably print an internal var name like _G123 instead of Xs)
- Query: first(7, List), rest([8,9], List).
  - Answer: List=[7,8,9].
  - Can you draw the structures that get unified to do this?

Decomposing lists

- In practice, no one ever actually defines rules for "first" and "rest."

- Just do the same thing by pattern matching: write things like [X|Xs] directly in your other rules.

List processing: member

- member(X,Y) should be true if X is any object, Y is a list, and X is a member of the list Y.

- member(X, [X|_]).   % same as "first"
- member(X, [Y|Ys]) :- member(X,Ys).
- Query: member(giraffe, [beaver, ant, steak(giraffe), fish]).
  - Answer: no   (why?)
  - It's recursive, but where is the base case???
    - if (list.empty()) then return "no"   % missing in Prolog??
      else if (x==list.first()) then return "yes"   % like 1st Prolog rule
      else return member(x, list.rest())   % like 2nd Prolog rule

question thanks to Michael J. Ciaraldi and David Finkel

Cut! 🎬

- ‘!’: Discard choice points of parent frame and frames created after the parent frame.

- Always is satisfied.
- Used to guarantee termination or control execution order.

- i.e. in the goal `:- p(X,a), !`
  - Only produce the 1st answer to X
  - Probably only one X satisfies *p* and trying to find another one leads to an infinite search!

- i.e. in the rule `color(X,red) :- red(X), !.`
  - Don’t try other choices of red (mentioned above) and color if X satisfies red
  - Similar to *then* part of a if-then-elseif

Red-Green Cuts (!)

- A ‘green’ cut
  - Only improves efficiency
  - e.g. to avoid additional unnecessary computation

- A ‘red’ cut
  - e.g. block what would be other consequences of the program
  - e.g. control execution order (procedural prog.)

Fisher, J.R., Prolog Tutorial,
http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html

Negative Facts

- How to define `nonsibling`? Logically…

```
nonsibling(X,Y) :- X = Y.
nonsibling(X,Y) :- mother(M1,X), mother(M2,Y), M1 \= M2.
nonsibling(X,Y) :- father(F1,X), father(F2,Y), F1 \= F2.
```

- But if parents of X or Y are not in database?
  - What is the answer of nonsibling? Can be solved by…

```
nonsibling(X,Y) :- no_parent(X).
nonsibling(X,Y) :- no_parent(Y).
```

  - How to define `no_parent`?

Negative Facts (cont.)

- Problem: There is no *positive* fact expressing the *absence* of parent.

- Cause:
  - Horn clauses are limited to
  - C :- P1,P2,…,Pn ≡ C holds if P1^P2^…^Pn hold.
  - No conclusion if P1^P2^…^Pn don't hold!
  - If, *not* iff

Cut-fail

Solutions:

- Stating *all* negative facts such as no_parent
  - Tedious
  - Error-prone
  - Negative facts about sth are usually much more than positive facts about it

- *"Cut-fail"* combination
  - nonsibling(X,Y) is satisfiable if sibling(X,Y) is not (i.e. sibling(X,Y) is **unsatisfiable**)
  - nonsibling(X,Y) :- sibling(X,Y), !, fail.
  - nonsibling(X,Y).
  - how to define 'fail' ?!

negation :- unsatisfiablility

- 'not' predicate
  - not(P) is satisfiable if P is not (i.e. is **unsatisfiable**).
  - not(P) :- call(P), !, fail.
  - not(P).
  - nonsibling(X,Y) :- not(sibling(X,Y)).
- Is 'not' predicate the same as 'logical negation'?