Prolog Details

February 8–9, 2012
CS 60: Principles of Computer Science

Assignment 3 due Monday: Prolog
Excerpts from simpsons.pl

%% Parent relation
parent(homer, bart).
parent(marge, bart).
parent(homer, lisa).
parent(marge, lisa).
parent(homer, maggie).
parent(marge, maggie).

%% Age relation
age(marge, 35).
age(homer, 38).
age(lisa, 8).
age(maggie, 1).
age(bart, 10).
age(gomer, 41).

%% Female predicate
female(marge).
female(jackie).
female(selma).
female(patty).
female(cher).
female(lisa).

%% Male predicate
male(homer).
male(gomer).
male(gemini).
male(glum).
male(bart).
male(millhouse).

%% Three rules about families
child(X, Y) :- parent(Y, X).

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

anc(X, Y) :- parent(X, Y).
anc(X, Y) :- parent(Z, Y), anc(X, Z).
“Quiz”

Define these three family-relationship predicates. (You must find a collection of facts that together make the relationship true.)

1. \textit{sib}(X,Y) — true when X and Y are siblings (or half-siblings)

   \textit{sib}(X,Y) :- \text{parent}(P,X), \text{parent}(P,Y), X \neq Y.

2. \textit{aunt}(A,N) — true when A is N’s aunt.

   \textit{aunt}(A,N) :- \text{parent}(P,N), \textit{sib}(A,P), \text{female}(A).

3. \textit{rel}(X,Y) — true when X and Y are related (by blood)

   \textit{rel}(X,Y) :- \text{anc}(Z, X), \text{anc}(Z, Y).
**Warning**

**OK:**

\[
\text{sib}(X, Y) :- \\
\text{parent}(A,X) \\
\text{parent}(A,Y), \\
X \neq Y.
\]

**OK:**

\[
\text{sib}(X, Y) :- \\
\text{parent}(A,X) \\
\text{parent}(A,Y), \\
X \neq Y.
\]

**Not OK:**

\[
\text{sib}(X, Y) :- \\
\text{parent}(A,X) \\
\text{parent}(A,Y), \\
X \neq Y.
\]
Prolog has one Algorithm: Depth-First Search

Who are Bart’s aunts?

`aunt(A, bart)`


parent(P, N)

- is there a parent?
  - parent(homer, bart)
    - YES, N = bart, P = homer
- other parents?
  - parent(marge, bart)
    - YES, N = bart, P = marge

sibling(A, P)

- is there a sibling?
  - sibling(gomer, homer)
    - YES, N = bart, P = homer, A = gomer
  - other siblings?
    - sibling(glum, marge)
      - YES, N = bart, P = marge, A = glum
    - other siblings?
      - sibling(selma, marge)
        - YES, N = bart, P = marge, A = selma

female(A)

- female(gomer) ?
  - NO (fails)
- female(glum) ?
  - NO
- female(selma)
  - SUCCESS, YES!

What space is Prolog searching through?

backtrack!

backtrack!

backtrack!
**PROLOG: The Reality**

We feed in a bunch of facts relevant to our problem:

- `pairs(apple, walnut).`
- `pairs(apple, honey).`
- `pairs(walnut, avocado).`
- `pairs(walnut, banana).`
- `pairs(X, X).`
- `pairs(X, coconut).`
- `pairs(X, Y) :- pairs(Y, X).`
- `pairs(X,Y) :- pairs(Y,X).`

We describe how to recognize a solution:

- `yummy_triple(X,Y,Z) :- pairs(X,Y), X \not= Y, pairs(Y,Z), Y \not= Z, pairs(X,Z), X \not= Z.`

Prolog then finds solution(s) for us(?)
We feed in a bunch of facts relevant to our problem:

pairs(apple, walnut).
pairs(apple, honey).
pairs(walnut, avocado).
pairs(walnut, banana).
%
% etc
pairs(X, X).
pairs(X, coconut).
pairs(X,Y) :- pairs(Y,X).

We describe how to recognize a solution:

yummy_triple(X,Y,Z) :- pairs(X,Y), X \= Y,
pairs(Y,Z), Y \= Z,
pairs(X,Z), X \= Z.

Prolog then finds solution(s) for us?
We feed in a bunch of facts relevant to our problem:

pairs(X, X).
pairs(X, coconut).
pairs(X,Y) :- pairs(Y,X).
pairs(apple, walnut).
pairs(apple, honey).
pairs(walnut, avocado).
pairs(walnut, banana).
%% etc

We describe how to recognize a solution:

yummy_triple(X,Y,Z) :- pairs(X,Y), X \= Y,
pairs(Y,Z), Y \= Z,
pairs(X,Z), X \= Z.

Prolog then finds solution(s) for us?

Conclusion: You often have to worry about the order of your facts and rules.
Unification means to take two expressions and make them the same, by giving values to unknown variables.

Fact: `pairs(X, coconut)`.

Query: `pairs(apple, coconut)`.

Result: Yes, because we can make the question and the answer unify, by setting `X = apple`. 
Equals Aren’t

Prolog has many different versions of (in)equalities.

= The two sides can and do unify
\= The two sides cannot be made to unify
== The two sides are already the same
\== The two sides are not already the same

is The LHS unifies with the value of the right-hand-side expression
=:= The value of the left is equal to the value of the right.
< The value of the left is strictly less than the value of the right.
Which of these queries succeed? What variables do they define?

% Unification
X = a. ✓
[X,a] = [b,Y]. ✓
X = [Y]. ✓
X = [X]. ✓
X = 5+2. ✓
[Root,L,R] = [42,[]]. ✓
[Root,L,R] = [42,[]]. ✓
[1,2] = [1,X]. ✓
X \= Y. X
X=3, Y=2, X \= Y. ✓

% Structure
1+2 == 1+2. ✓
1+2 == 2+1. X
[1,X] == [1,X]. ✓
[1,2] == [1,X]. X
X \= Y. ✓
X=3, Y=3, X \= Y. X

% Math
X is 5+2. ✓
X is Y+3. X
Y = 6, X is Y*7. ✓
1+2 is 2+1. X
1+2 =:= 2+1. ✓
X = 3, Y < X. X
1+2 < 3*4 ✓
This may be tempting, but is totally broken. Why?

```
fac(0) :- 1.
fac(X) :- X * fac(X-1).
```

We only have relations that succeed or fail; no return values!

```
fac(0,1).
fac(X,N) :-
```
**Using length and member in Prolog**

What should the following Prolog queries do?

- `length([a,b,c,d], 4). ✔`
- `length([a,b,c], 4). ✗`
- `length([a,b,c], N). ✔`
- `length(L, 0). ✔`
- `member(c, [a,b,c,d]). ✔`
- `member(e, [a,b,c,d]). ✗`
- `member(X, [a,b,c,d]). ✔`
Redefining \texttt{length} in \texttt{Prolog}

Base case:

\begin{verbatim}
length(L,N) :- L = [], N = 0.
\end{verbatim}

or more simply:

\begin{verbatim}
length([], 0).
\end{verbatim}
Redefining \textit{length} in Prolog

Recursive case:

\begin{verbatim}
length(L,N) :- L = [F|R], length(R,M), N is M+1.
\end{verbatim}

or more simply:

\begin{verbatim}
length([F|R], N) :- length(R,M), N is M+1.
\end{verbatim}
Redefining `append` and `member` in Prolog

member(E, [E|_]).
member(E, [_|R]) :- member(E, R).

% or
% member(E, L) :- L = [E|_].
% member(E, L) :- L = [_|R], member(E, R).

append([], M, M).
append([F|R], M, [F|RM]) :- append(R, M, RM).

% or
% append(L, M, LM) :- L = [], LM = M.
% append(L, M, LM) :- L = [F|R],
% append(R, M, RM),
% LM = [F|RM].
1. \texttt{lastof(E,L)} — true when \texttt{E} is the last element of list \texttt{L}.

\texttt{lastof(E, [E]).}
\texttt{lastof(E, [_|R]) :- lastof(E, R).}

2. \texttt{nnodes(BST,Y)} — true when \texttt{BST} is a binary tree (represented using nested lists) and \texttt{N} is the number of nodes.

\texttt{nnodes([], 0).}
\texttt{nnodes([Root,L,R],N) :-}
    \texttt{nnodes(L,LN), nnodes(R,RN), N is LN+RN+1.}
**Negation is Tricky**

How to discover Skugerina is 101?

1. `oldest(X) :- AX > AY, age(X,AX), age(Y,AY).`
   - *X* immediately chokes on comparing unknown *AX* and *AY*

2. `oldest(X) :- age(X,AX), age(Y,AY), AX > AY.`
   - *X* true for *X* if Prolog can find any *Y* who is younger; we’ve defined “not youngest”!

3. `notoldest(X) :- age(X,AX), age(Y,AY), AX < AY.`
   - `oldest(X) :- \+ notoldest(X).`
   - *X* works for specific queries like `oldest(bart)` and `oldest(skugerina)`, but not for the more general query `oldest(P)`. Why? Because `notoldest(P)` could succeed (with *P*=maggie among others) when *P* is unknown, and so `\+ notoldest(P)` does the opposite and fails.

4. `notoldest(X) :- age(X,AX), age(Y,AY), AX < AY.`
   - `oldest(X) :- person(X), \+ notoldest(X).`

**Conclusion:** put inequalities and negation as late as possible, when variables already have known values.
Another Example

Compare:

\[
\text{sib}(X,Y) :- \text{parent}(Z,X), \text{parent}(Z,Y), X \not= Y.
\]

✓ Works fine. \text{sib}(X,\text{bart}) will not suggest \(X = \text{bart}\).

\[
\text{sib}(X,Y) :- X \not= Y, \text{parent}(Z,X), \text{parent}(Z,Y).
\]

✗ Works correctly for specific questions like \text{sib}(\text{bart},\text{lisa})✓ and \text{sib}(\text{bart},\text{bart})✗

But, if you ask \text{sib}(X,\text{bart}), Prolog (1) confirms that \(X\) is not (yet!) identical to \(Y\) (\text{bart}), and then (2) finds an \(X\) that shares a parent with \text{bart}. At this point, \(X = \text{bart}\) will be returned. The check is useless.

Conclusion: put inequalities and negation as late as possible, when variables already have known values.
Another Example

Consider the following code:

\[
\text{count}(E, [], 0).
\text{count}(E, [E|R], N) :- \text{count}(E,R,M), N \text{ is } M+1.
\text{count}(E, [F|R], N) :- E \neq F, \text{count}(E,R,N).
\]

What will Prolog do with the query

\[
\text{count}(E, [\text{spam, oh, spam}], N).
\]

Because \text{E} and \text{spam} are not (yet!) identical, the third rule tells us that we can conclude \text{count}(E, [\text{spam,oh,spam}], N) if we can show \text{count}(E, [\text{oh,spam}], N). Thus, Prolog will report \text{E=spam}, N=1 as one of the possible solutions.

Conclusion: …