Prolog Details

February 8–9, 2012

CS 60: Principles of Computer Science

Assignment 3 due Monday: Prolog

Excerpts from simpsons.pl

```
%% Parent relation         %% Age relation    %% Female predicate   %% Male predicate
parent(homer, bart).       age(marge, 35).    female(marge).        male(homer).
parent(marge, bart).       age(homer, 38).    female(jackie).       male(gomer).
parent(homer, lisa).       age(lisa, 8).      female(selma).        male(gemini).
parent(marge, lisa).       age(maggie, 1).    female(patty).        male(glum).
parent(homer, maggie).     age(bart, 10).     female(cher).         male(bart).
parent(marge, maggie).     age(gomer, 41).    female(lisa).         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).
```

Name: "Quiz"

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

1. `sib(X,Y)` — true when X and Y are siblings (or half-siblings)

   ```
   sib(X,Y) :- parent(P,X), parent(P,Y), X =▯ Y.
   ```

2. `aunt(A,N)` — true when A is N's aunt.

   ```
   aunt(A,N) :- parent(P,N), sib(A,P), female(A).
   ```

3. `rel(X,Y)` — true when X and Y are related (by blood)

   ```
   rel(X,Y) :- anc(Z, X), anc(Z, Y).
   ```

Warning

OK:

```
sib(X, Y) :-  parent(A,X)
              parent(A,Y),
              X \== Y.
```

OK:

```
sib(X, Y) :-  parent( A , X )
              parent( A , Y ),
              X \== Y.
```

Not *OK:*

```
sib(X, Y) :-  parent (A,X)
              parent (A,Y),
              X \== Y.
```

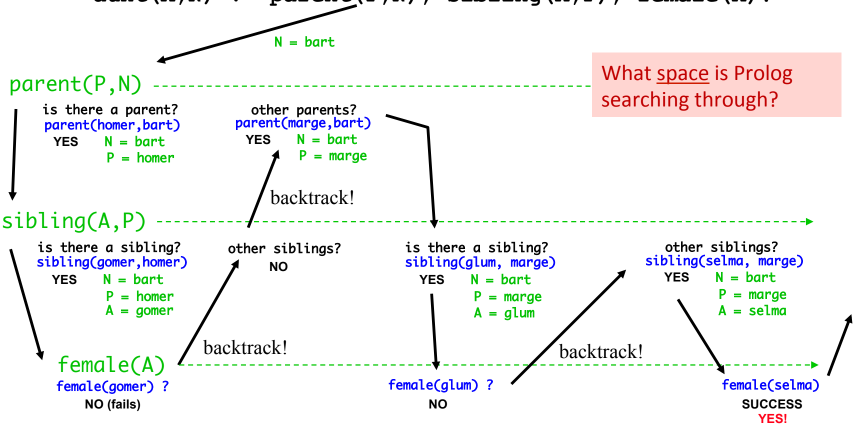
How does Prolog Work?

Prolog has ***one*** Algorithm: Depth-First Search

Who are Bart's aunts?

`aunt(A,bart)`

```
aunt(A,N) :- parent(P,N), sibling(A,P), female(A).
```



What space is Prolog searching through?

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).
%% 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(?)

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).
%% 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?

Prolog: The Reality

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: Prolog's Main Tool

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.

Exercise

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=3, Y=2, X \= Y. ✔
```

```
% Structure
1+2 == 1+2. ✔
1+2 == 2+1. ✗
[1,X] == [1,X]. ✔
[1,2] == [1,X]. ✗
X \== Y. ✔
X=3, Y=3, X \== Y. ✗
```

```
% Math
X is 5+2. ✔
X is Y+3. ✗
Y = 6, X is Y*7. ✔
1+2 is 2+1. ✗
1+2 =:= 2+1. ✔
X = 3, Y < X. ✗
1+2 < 3*4 ✔
```

Math in Prolog

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 `length` IN PROLOG

Base case:

```
length(L,N) :- L = [], N = 0.
```

or more simply:

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

Redefining length in Prolog

Recursive case:

```
length(L,N) :- L = [F|R], length(R,M), N is M+1.
```

or more simply:

```
length([F|R], N) :- length(R,M), N is M+1.
```

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].
```

Name: "Quiz"

1. `lastof(E,L)` — *true when* `E` *is the last element of list* `L`*.*

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

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

```
nnodes([], 0).
nnodes([Root,L,R],N) :-
    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).`

   ✗ Immediately chokes on comparing unknown **AX** and **AY**

2. `oldest(X) :- age(X,AX), age(Y,AY), AX > AY.`

   ✗ 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).
   ```

   ✗ 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:

```
sib(X,Y) :- parent(Z,X), parent(Z,Y), X \== Y.
```

✔ Works fine. `sib(X,bart)` will *not* suggest `X = bart`.

```
sib(X,Y) :- X \== Y, parent(Z,X), parent(Z,Y).
```

✗ Works correctly for specific questions like `sib(bart,lisa)`✔ and `sib(bart,bart)`✗

But, if you ask `sib(X,bart)`, Prolog (1) confirms that `X` is not (yet!) identical to `Y` (`bart`), and then (2) finds an `X` that shares a parent with `bart`. At this point, `X = 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:

```
count(E, [], 0).
count(E, [E|R], N) :- count(E,R,M), N is M+1.
count(E, [F|R], N) :- E \== F, count(E,R,N).
```

What will Prolog do with the query

```
count(E, [spam, oh, spam], N).
```

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

Conclusion: ...