CSC 324: Principles of Programming Languages

Logic Programming

READING: Sebesta Chapter 15

Part V: Logic Programming and Prolog

Logic programming languages are not procedural or functional.

- Specify *relations* between objects
  - `larger(3,2)`
  - `father(tom,jane)`
- Separate logic from control:
  - Programmer declares **what** facts and relations are true.
  - System determines **how** to use facts to solve problems.
  - System **instantiates** variables in order to make relations true!
- Computation engine: theorem-proving and recursion (Unification, Resolution, Backward Chaining, Backtracking)
  - Higher-level than imperative languages

Basic Facts and Relations

Start with the basic facts...

```
> cat family.pl
male(albert). <------------------------- a fact
male(edward).            Facts are put in a file.
female(alice).
female(victoria).
parent(albert,edward).
parent(victoria,edward).
parent(albert,alice).
parent(victoria,alice).
> pl
?- [family]. <--------- loads file family.pl
yes
?- male(albert). <--------------------- a query
yes
?- male(alice).
no
?- parent(albert,edward).
yes
?- parent(bullwinkle,edward).
no
```

Limited use: need variables and deductive rules.

Variables and Unification

```
?- female(X).
X = alice
yes

?- female(X).
X = alice ; <------------ ';' means look further
X = victoria ;
no
```

X is **unified** to all possible values that make the query female(X) true.

```
?- parent(P,edward).
P = albert ;
P = victoria ;
no
```

P is unified to all possible values that make the query parent(P,edward) true.

⇒ search with pattern matching

In Prolog:

- All variables are capitalized.
- All constants are in lower case.
- All predicates are in lower case.

Prolog Queries

A query is a proposed fact that is to be proven.

- If the query has no variables, returns yes/no.
- If the query has variables, returns appropriate values of variables (called a substitution).

Horn Clauses (Rules)

A Horn Clause is: $c \leftarrow h_1 \wedge h_2 \wedge h_3 \wedge \ldots \wedge h_n$

- Antecedents: conjunction of zero or more conditions which are atomic formulae in predicate logic
- Consequent: an atomic formula in predicate logic

Meaning of a Horn clause:

- “The consequent is true if the antecedents are all true”
- c is true if h_1, h_2, h_3, . . . , and h_n are all true

Horn Clause Terminology

- Horn Clause = Clause
- Consequent = Goal = Head
- Antecedents = Subgoals = Body
- Horn Clause with No Body = Fact
- Horn Clause with Body = Rule

In Prolog, a Horn clause

$$c \leftarrow h_1 \wedge \ldots \wedge h_n$$

is written

c :- h$_1$,   , h$_n$.

Syntax elements: ‘:-’ ‘,’ ‘.’

Prolog Horn Clause Examples

A Horn clause with no body:

```
male(albert).
```

I.e., a fact: albert is a male dependent on no other conditions

A Horn clause with a body:

```
father(albert,edward):-
   male(albert), parent(albert,edward).
```

I.e., a rule: albert is the father of edward if albert is male and albert is a parent of edward's.

Horn Clauses with Variables

Variables may appear in the antecedents **and** consequent of a Horn clause:

- c(X_1,. . . ,X_n) :- h(X_1,. . . ,X_m).

  "For all values of X_1,. . . ,X_n, the formula c(X_1,. . . ,X_n) is true if the formula h(X_1,. . . ,X_n) is true"

- c(X_1,. . . ,X_n) :- h(X_1,. . . ,X_m,Y_1,. . . ,Y_k).

  "For all values of X_1,. . . ,X_n, the formula c(X_1,. . . ,X_n) is true if there exist values of Y_1,. . . ,Y_k such that the formula h(X_1,. . . ,X_n,Y_1,. . . ,Y_k) is true"

Examples:

```
father(X,Y):- male(X), parent(X,Y).

?- father(F,edward).
F = albert ;
no

child_of(C,P):- parent(P,C).

?- child_of(C,P).
C = edward, P = albert ;
C = edward, P = victoria ;
C = alice, P = albert ;
C = alice, P = victoria ;

no
```

NOTE: Always use ';' when debugging, and always test all combinations of variables and constants.

Example Prolog Clauses

```
sibling(X,Y):- parent(P,X), parent(P,Y).

?- sibling(alice,A).
A = edward ;
A = alice ;
A = edward ;
A = alice ;
no
```

Rule Ordering and Unification

1. rule ordering used in search
2. unification requires two instances of the same variable in the same rule to get the same value
3. unification does not require differently named variables to get different values: hence, sibling(edward,edward).
4. all rules searched if requested by ';'

```
?- sibling(A,alice).
A = edward ;
A = edward ;
A = alice ;
A = alice ;
no
```

Note: arguments are interchangeable, but ordering affects order of search.

```
brother(X,Y):- ...

sister(X,Y):- ...

uncle(X,Y):- ...

aunt(X,Y):- ...

nephew(X,Y):- ... OR ...??

niece(X,Y):- ... OR ...??
```

Note: "disjunction" (logical-or) is achieved by defining multiple clauses.

Transitive Relations

```
parent(sally,jane).     parent(bob,jane).
parent(sally,john).     parent(bob,john).
parent(mary,sally).     parent(al,sally).
parent(ann,bob).        parent(mike,bob).
parent(jean,al).        parent(joe,al).
parent(ruth,mary).      parent(jim,mary).
parent(esther,ruth).    parent(mick,ruth).

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

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

```
?- grandparent(Y,jane).
Y = mary ;
Y = al ;
Y = ann ;
Y = mike ;
no

?- ancestor(X,jane).
X = sally ;
X = bob ;
X = mary ;
X = al ;
X = ann ;
X = mike ;
X = jean ;
X = joe ;
X = ruth ;
X = jim ;
X = esther ;
X = mick ;
no
```

Procedural Semantics of Prolog

```
uncle(X,Z) :- male(X), sibling(X,Y),
              parent(Y,Z).
```

```
?- uncle(X,jane). % a query
```

In order to find an X to make $uncle(X, jane)$ true:

1. Set Z to jane.
2. Find an X to make $male(X)$ true.
3. Find a Y such that $sibling(X, Y)$ is true.
4. Check that $parent(Y, jane)$ is true.

Procedural Semantics of Prolog

Notice the recursion in this algorithm: "find" calls "find". This reasoning is recursively applied until we reach rules that are facts.

This process is called **Backward Chaining**.

Logic Programming vs. Prolog

```
cousin(X,Y) :- parent(W,X), sister(W,Z),
               parent(Z,Y).

cousin(X,Y) :- parent(W,X), brother(W,Z),
               parent(Z,Y).
```

```
?- cousin(X,jane). % a query
```

Rule and Goal Ordering:

- There are two rules for cousin
- Which rule do we try first?
- Each rule for cousin has several subgoals
- Which subgoal do we try first?

Logic Programming vs. Prolog

Logic Programming: *Nondeterministic*

- Arbitrarily choose rule to expand first
- Arbitrarily choose subgoal to to explore first
- Results don't depend on rule and subgoal ordering

Prolog: *Deterministic*

- Expand first rule first
- Explore first subgoal first
- Results may depend on rule and subgoal ordering

(Minimal) Prolog Syntax

```
<rule> ::= <head> :- <body> . |
           <fact> .

<head> ::= <predicate>

<fact> ::= <predicate>

<body> ::= <predicate> { , <predicate> }

<predicate> ::=

 <functor> '(' <term> { , <term> } ')'

<term> ::= <integer> | <atom> |
            <variable> | <predicate>

<query> ::= ?- <predicate>.
```

Lists in Prolog

- '[' List Elems Separated by Commas ']'
- '[' First Elem '|' Rest of List ']'

List	**head**	**tail**	**equivalent**
[a,b,c]	a	[b,c]	
[a]	a	[]	
[X,[cat],Y]	X	[[cat],Y]	
[[a,b],c,d]	[a,b]	[c,d]	

Unifying Lists

[X, Y, Z]	[john, likes, fish]
[cat]	[X | Y]
[1, 2]	[X, Y]
[1 | 2]	[X | Y]
[1 | 2]	[X, Y]
[a,b,c]	[X | Y]
[a,b|Z]	[X | Y]
[X, abc, Y]	[X, abc | Y]
[[the | Y] | Z]	[[X, hare] | [is, here]]

"Appending" Lists

```
append([],A,A).
append([A|B],C,[A|D]) :- append(B,C,D).

Build a list:
        ?- append([a],[b],Y).
        Y=[a,b]
        yes
Break a list up:
        ?- append(X,[b],[a,b]).
        X=[a]
        yes
        ?- append([a],Y,[a,b]).
        Y=[b]
        yes
        ?- append(X,Y,[a,b]).
        X=[],Y=[a,b] ;
        X=[a],Y=[b] ;
        X=[a,b],Y=[] ;
        no
```

NOTE: There is a built-in append function!!

List Membership

member(A,[A|_]).

member(A,[_|C]) :- member(A,C).

```
?- member(a,[a,b]).
yes
?- member(a,[b,c]).
no
?- member(X,[a,b,c]).
X=a ;
X=b ;
X=c ;
no
?- member(a,[c,b,X]).
X=a ;
no
```

NOTE: There is a built-in member function!!

Avoiding Irrelevant Unification

A graph:

```
arc(a,b).
arc(a,c).
arc(b,c).
arc(b,d).
```

Rule for nodes of graph:

```
node(X):- arc(X,Y).
node(X):- arc(Y,X).
```

On reading these in:

```
++Warning: Singleton variable Y in a clause of node/1
++Warning: Singleton variable Y in a clause of node/1
```

Avoiding Irrelevant Unification

Above has irrelevant unification of Y ⇒ wasted effort.

Anonymous variable:
"don't care" symbol _ which means the argument exists but doesn't signify anything.

```
node(X):- arc(X,_).
node(X):- arc(_,X).
```

Removes useless unification.

Built-in Predicates

There are "built-in" predicates such as: member, append, length.

```
?- member(2,[1,2,3]).
Yes

?- member(X,[1,2]).
X = 1 ;
X = 2 ;
No

?- append([1,2],[3,4],L).
L = [1, 2, 3, 4] ;
No

?- append([1,2], L, [1,2,3,4]).
L = [3, 4] ;
No

?- length([1,2,3],L).
L = 3 ;
No
```

Generate lists:

?- append(X,[b],Z).

X=[],Z=[b] ;

X=[_98],Z=[_98,b] ;

X=[_98,_102],Z=[_98,_102,b] ;

. . .

?- member(X,Y).

X=_72, Y=[_72|_73] ;

X=_74, Y=[_72,_74|_75] ;

X=_76, Y=[_72,_74,_76|_77] ;

. . .

Lazy evaluation of potentially infinite data structures

Accessing More Than One Initial Element

Definition of `swap_first_two`...

```
?- swap_first_two([a,b], [b,a]).
yes
?- swap_first_two([a,b], [b,c]).
no
?- swap_first_two([a,b,c], [b,a,c]).
yes
?- swap_first_two([a,b,c], [b,a,d]).
no
?- swap_first_two([a,b,c], X).
X = [b,a,c];
no
?- swap_first_two([a,b|Y], X).
Y = _56, X = [b,a|_56];
no
?- swap_first_two([],X).
no
?- swap_first_two([a],X).
no
?- swap_first_two([a,b],X).
X = [b,a];
no
```

Computing the Length of a List

length([],0).

length([_ | Y],L) :- length(Y,M), L is M + 1.

```
?- length([a,b,c],L).
L = 3

?- length([],L).
L = 0

?- length(X,3).
X = [_66,_68,_70]

?- length(X,0).
X = []
```

NOTE: There is a built-in length function!!

Lists of a Specified Length

```
list_of_elem(L,E,N) :-
       all_elem(L,E),
       length(L,N).
all_elem([],_).
all_elem([F|R],F) :-
all_elem(R,F).
```

```
?- list_of_elem(X,b,3).
X = [b,b,b];
ERROR: Out of global stack

?- list_of_elem(X,Y,2).
X = [_50,_50]
Y = _50;
ERROR: Out of global stack
```

Lists of a Specified Length

```
 working_list_elem(L,E,N) :-
        length(L,N),
        all_elem(L,E).

?- working_list_elem(X,b,3).
X = [b,b,b];
no

?- working_list_elem(X,Y,2).
X = [_50,_50]
Y = _50;
no
```

Arithmetic in Prolog

```
?- AGE is 1995 - 1956.
AGE = 39

?- DATE is 1956 + 39.
DATE = 1995

?- 39 is DATE - 1956.
ERROR: Arguments are not sufficiently instantiated

?- 1995 is 1956 + AGE.
ERROR: Arguments are not sufficiently instantiated
```

Arithmetic in Prolog

At the time Prolog begins processing a goal of the form: `X is <Exp>`, `<Exp>` must be a fully instantiated arithmetic expression, i.e., it cannot have any unbound variables

⇒ Arithmetic programs are not always invertible.

Factorial Function

```
factorial(0,1).
factorial(X,Y) :- W is X-1,
                  factorial(W,Z),
                  Y is Z*X.
```

This calculates X! if X is bound to an integer. Otherwise it aborts in the first "is" clause.

```
?- factorial(3,6).
yes
?- factorial(5,Z).
Z = 120
yes
?- factorial(Y,6).
ERROR: Arguments are not sufficiently instantiated
^  Exception: (7) _G296 is _G227-1 ?
```

Factorial Function

NOTE: Problem with asking for multiple solutions.

?- factorial(5,Z).
Z = 120;
ERROR: Out of local stack

Adding up the Numbers on a List

```
sumlist([],0).
sumlist([X|Y],S) :- sumlist(Y,T),
                    S is X + T.
```

```
?- sumlist([1,2,3],X).
X = 6;
no

?- sumlist([],X).
X = 0;
no

?- sumlist(X,3).
ERROR: Arguments are not sufficiently instantiated
^  Exception: (7) 3 is _G271+0 ? creep
```

Execution of Prolog Programs

- **Unification**: (variable bindings)
  Specializes general rules to apply to a specific problem.
- **Backward Chaining**:
  Reduces a goal to one or more subgoals.
- **Backtracking**:
  Systematically searches for all possible solutions that can be obtained via unification and backchaining.

Prolog Search Trees

Encapsulate unification, backward chaining, and backtracking.

- Internal nodes are ordered list of subgoals.
- Leaves are success nodes or failures, where computation can proceed no further.
- Edges are labeled with variable bindings that occur by unification.

Describe *all possible computation* paths.

- There can be many success nodes.
- There can be infinite branches.

Prolog Execution Example 1

```
male(tom).               %Male Clause 1.
male(fred).              %Male Clause 2.

female(jane).            %Female Clause 1.
female(betty).           %Female Clause 2.
female(mary).            %Female Clause 3.

sibling(betty,fred).     %Sibling Clause 1.
sibling(fred,betty).     %Sibling Clause 2.
sibling(mary,fred).      %Sibling Clause 3.
sibling(fred,mary).      %Sibling Clause 4.

parent(tim,jane).        %Parent Clause 1.
parent(mary,jane).       %Parent Clause 2.

uncle(X,Z) :- male(X),   %Uncle Clause.
              sibling(X,Y),
              parent(Y,Z).
```

Search Tree for Example 1

```
?- uncle(X,Z).
```

Prolog Execution Trace for Example 1

```
?- trace.
yes
[trace]
  ?- uncle(X,Z).
   Call:  (  7) uncle(_G129, _G130) ? creep
   Call:  (  8) male(_G129) ? creep
   Exit:  (  8) male(tom) ? creep
   Call:  (  8) sibling(tom, _L142) ? creep
   Fail:  (  8) sibling(tom, _L142) ? creep
   Redo:  (  8) male(_G129) ? creep
   Exit:  (  8) male(fred) ? creep
   Call:  (  8) sibling(fred, _L142) ? creep
   Exit:  (  8) sibling(fred, betty) ? creep
   Call:  (  8) parent(betty, _G130) ? creep
   Fail:  (  8) parent(betty, _G130) ? creep
   Redo:  (  8) sibling(fred, _L142) ? creep
   Exit:  (  8) sibling(fred, mary) ? creep
   Call:  (  8) parent(mary, _G130) ? creep
   Exit:  (  8) parent(mary, jane) ? creep
   Exit:  (  7) uncle(fred, jane) ? creep

X = fred
Y = jane ;

No
```

Prolog Execution Example 2

```
flight(ny,chicago).                      %Clause 1.
flight(ny,miami).                        %Clause 2.
flight(miami,austin).                    %Clause 3.

trip(X,Z) :- flight(X,Z).                %Clause 1.
trip(X,Z) :- flight(X,Y),trip(Y,Z).      %Clause 2.
```

Prolog Search Tree for Example 2

```
?- trip(ny,austin).
```

Prolog Execution Trace

```
[trace]
?- trip(ny,austin).
   Call:  (  7) trip(ny, austin) ? creep
   Call:  (  8) flight(ny, austin) ? creep
   Fail:  (  8) flight(ny, austin) ? creep
   Redo:  (  7) trip(ny, austin) ? creep
   Call:  (  8) flight(ny, _L140) ? creep
   Exit:  (  8) flight(ny, chicago) ? creep
   Call:  (  8) trip(chicago, austin) ? creep
   Call:  (  9) flight(chicago, austin) ? creep
   Fail:  (  9) flight(chicago, austin) ? creep
   Redo:  (  8) trip(chicago, austin) ? creep
   Call:  (  9) flight(chicago, _L163) ? creep
   Fail:  (  9) flight(chicago, _L163) ? creep
   Fail:  (  8) trip(chicago, austin) ? creep
   Redo:  (  8) flight(ny, _L140) ? creep
   Exit:  (  8) flight(ny, miami) ? creep
   Call:  (  8) trip(miami, austin) ? creep
   Call:  (  9) flight(miami, austin) ? creep
   Exit:  (  9) flight(miami, austin) ? creep
   Exit:  (  8) trip(miami, austin) ? creep
   Exit:  (  7) trip(ny, austin) ? creep

Yes
```

Negation as Failure

No equivalent of logical not in Prolog:

- Prolog can only assert that something is true.
- Prolog **cannot** assert that something is false.
- Prolog can assert that the given facts and rules do not allow something to be proven true.

Negation as Failure

Assuming that something unprovable is false is called **negation as failure**.

(Based on a **closed world assumption**.)

The goal \+(G) succeeds whenever the goal G fails.

```
?- member(b,[a,b,c]).
yes
?- \+(member(b,[a,b,c])).
no
?- \+(member(b,[a,c])).
yes
```

Example: Disjoint Sets

```
overlap(S1,S2) :- member(X,S1),member(X,S2).

disjoint(S1,S2) :- \+(overlap(S1,S2)).

?- overlap([a,b,c],[c,d,e]).
yes
?- overlap([a,b,c],[d,e,f]).
no
?- disjoint([a,b,c],[c,d,e]).
no
?- disjoint([a,b,c],[d,e,f]).
yes
?- disjoint([a,b,c],X).
no   %<---------Not what we wanted
```

Proper use of Negation as Failure

\+(G) works properly only in the following cases:

1. When G is fully instantiated at the time prolog processes the goal \+(G).

   (In this case, \+(G) is interpreted to mean “goal G does not succeed”.)

2. When all variables in G are unique to G, i.e., they don’t appear elsewhere in the same clause.

   (In this case, \+(G(X)) is interpreted to mean “There is no value of X that will make G(X) succeed”.)

Adding Subgoals in Search Trees

```
male(tom).               %Male Clause 1.
male(fred).              %Male Clause 2.

female(jane).            %Female Clause 1.
female(betty).           %Female Clause 2.
female(mary).            %Female Clause 3.

sibling(betty,fred).     %Sibling Clause 1.
sibling(fred,betty).     %Sibling Clause 2.
sibling(mary,fred).      %Sibling Clause 3.
sibling(fred,mary).      %Sibling Clause 4.

parent(tim,jane).        %Parent Clause 1.
parent(mary,jane).       %Parent Clause 2.

uncle(X,Z) :- male(X),        %Uncle Clause.
              sibling(X,Y),
              parent(Y,Z).

niece(X,Y):- female(X), uncle(Y,X). %Niece Clause.
```

Another Example Search Tree

```
?- niece(X,Y).
```

Problems with Multiple Solutions

```
isa_mother(X) :- female(X),parent(X,_).
isa_father(X) :- male(X),parent(X,_).

parent(fred,sue).
parent(janet,sue).
parent(fred,tim).
parent(janet,tim).

male(fred).
female(janet).

?- isa_mother(X).
X = janet ;
X = janet ;
no
```

Suppressing Multiple Solutions with Cut

The cut symbol **!** is prolog syntax for:

- a goal that always succeeds.
- with the side effect of suppressing backtracking.
  - Prolog will not attempt to find additional solutions to any goals to the left of the cut.

⇒ Cut prunes the search tree that prolog generates.

Suppressing Multiple Solutions with Cut

```
isa_mother(X) :- female(X),parent(X,_),!.
isa_father(X) :- male(X),parent(X,_),!.

parent(fred,sue).
parent(janet,sue).
parent(fred,tim).
parent(janet,tim).

male(fred).
female(janet).

?- isa_mother(X).
X = janet ;
no
```

Negation by Failure Revisited

```
not(X):- X, !, fail.
not(_).
```

Summary

Cuts are:

+ very powerful.

+ can help make programs more efficient.

– difficult to use safely.

– make for difficult to understand programs.

Summary of Part V: Logic Programming

- Programming with relations
- Separate logic from control
- Horn clauses, facts, rules
- Prolog
- Unification
- Backward chaining
- Prolog lists and list unification
- Prolog arithmetic
- Backtracking
- Prolog search trees
- Cuts
- Negation by failure