CSE 341 - Programming Languages - Autumn 2012

Language Metaphors:

- Algol family: Von Neumann machine
- functional programming: function definition and application
- object-oriented programming: simulation
- logic programming: theorem proving

Very brief Prolog history

Prolog's roots are in predicate calculus and theorem proving
Alain Colmerauer (Marseilles) invented Prolog in the early 70's
David Warren from the University of Edinburgh implemented a compiler, and also defined the WAM (Warren Abstract Machine), a well-known technique for implementing Prolog

the Fifth Generation project in Japan (1984) popularized Prolog
lots of offshoots: constraint logic programming: CLP languages, CHIP, Prolog III, Trilogy, HCLP, concurrent logic programming, etc.

Prolog Programs

A Prolog program is a collection of facts and rules (like axioms). A query is in effect a theorem to be proved.

Two modes: enter assertions; make queries

Suppose we have the following Prolog program in a file named `basics.pl`

```
likes(fred,beer).
likes(fred,cheap_cigars).
likes(fred,monday_night_football).

likes(sue,jogging).
likes(sue,yogurt).
likes(sue,bicycling).
likes(sue,amy_goodman).

likes(mary,jogging).
likes(mary,yogurt).
likes(mary,bicycling).
likes(mary,rush_limbaugh).

health_freak(X) :-
   likes(X,yogurt),
   likes(X,jogging).

left_wing(X) :-
   likes(X,amy_goodman).

right_wing(X) :-
   likes(X,rush_limbaugh).

low_life(X) :-
  likes(X,cheap_cigars).
```

File these in by saying

```
| ?- consult(basics).
```

Make queries:

```
| ?- likes(fred,beer).
```

```
true.

| ?- likes(fred,yogurt).
false

| ?- likes(fred,X).
X = beer
```

However, Fred likes other things besides beer. We can reject an answer by typing a semicolon, and get more by backtracking:

```
| ?- likes(fred,X).
X = beer ;
X = cheap_cigars ;
X = monday_night_football.

| ?- health_freak(Y).
Y = sue ;
Y = mary.

/* is there anyone who is both left wing and a lowlife (known in our
   database, that is)? */
| ?- left_wing(X), low_life(X).
false
```

How about right wing and a lowlife? right wing and a health freak?

Recursive Rules

```
/* Some CSE majors courses and their prerequisites.  This simplifies
   the actual CSE curriculum by assuming courses have at most one
   direct prerequisite. */

prerequisite(cse142,cse143).

prerequisite(cse143,cse311).

prerequisite(cse311,cse312).

prerequisite(cse143,cse331).

prerequisite(cse143,cse341).

/* take_before(A,B) succeeds if you must take A before B */
take_before(X,Z) :- prerequisite(X,Z).
take_before(X,Z) :- prerequisite(X,Y),
                    take_before(Y,Z).
```

Then we can issue queries such as:

```
take_before(cse142,cse341).
take_before(cse341,cse311).
take_before(X,cse341).
prerequisite(X,Y).
```

Key concepts in Prolog:

- logic variables (scope rules: variables locally scoped within a fact, rule, or query)
- unification (two-way pattern matching)
- depth-first search; backtracking
- dual declarative and procedural reading of Prolog program

Prolog data types:

- variables -- begin with capital letter
  `X, Y, Fred, A_very_long_variable_name`
  anonymous variable: _
- integers, reals
- atoms -- begin with lower-case letter
  `x, y, fred, a_very_long_atom_name`

lists

```
[]    /* the empty list */
[10]
[10,11,12]
 [ [squid,octopus,clam], dolphin]
```

- structures

```
point(10,30)
line(point(10,30),point(99,100))
```

Some list queries:

```
A = [4,5,6], B=[3|A].   /* then B =[3,4,5,6]  */
A = [4,5,6], B=[3,A].   /* then B =[3,[4,5,6]]  */
```

The list notation is just shorthand for a set of structures, where "." plays the role of `cons`

```
.(4, .(5, []))
```

is the same as `[4,5]`

point, line, . are *unevaluated function symbols*

Unification

two terms unify:

- if they are identical
- if the variables in the terms can be instantiated to objects such that after the substitutions, the terms are identical

examples:

fred unifies with fred
X unifies with fred (by substituting fred for X)
X unifies with Y (by substituting Y for X, or substituting 3 for X and 3 for Y)
point(A,10) unifies with point(B,C)
clam(X,X) unifies with clam(Y,3)

When Prolog unifies two terms, it picks the **most general unification**
point(A,A) unifies with point(B,C) by substituting A for B and A for C

Unification algorithm

to unify S and T,

- if S and T are constants, they unify only if they are the same object.
- if S is a variable and T is anything, then they match. Substitute T for S.
- if T is a variable and S is anything, then they match. Substitute S for T.
- if S and T are structures, S and T must have the same principal functor, and the corresponding components must unify.

Note that if you substitute T for S, the substitution must be carried out throughout the structure.

Nit: the logical definition of unification also includes the "occurs check": a variable is not allowed to unify

with a structure containing that variable. For example, X is not allowed to unify with f(X). However, most practical implementations of Prolog skip the occurs check for efficiency.

Unification can also be viewed as constraint solving, where the constraints are limited to equations over Prolog's data structures.

Prolog programs have both a declarative and a procedural meaning:

- declarative: WHAT
- procedural: HOW

Prolog clauses are either facts and rules.

Declarative meaning

a rule such as

```
P :-  Q1, Q2, Q3.
```

means: if Q1 and Q2 and Q3 are true, then P is true.

a fact such as:

```
P.
```

means P is true.

A goal G is satisfiable if there is a clause C such that

- there is a clause instance I of C such that
- the head of I is identical to G and
- all the goals in the body of I are satisfiable

In the declarative meaning of a Prolog program, the order of the clauses, and the order of the goals in the body of a rule, doesn't matter.

another way of describing this: variables in rules are UNIVERSALLY QUANTIFIED:

```
low_life(X) :- likes(X,cheap_cigars).
```

means for every X, if likes(X,cheap_cigars) is true, then low_life(X) is true

variables in goals are EXISTENTIALLY QUANTIFIED:

```
?- likes(fred,X).
```

means prove that there exists an X such that likes(fred,X)

A goal is satisfiable if it can be proven from the clauses.

Procedural meaning

Given a list of goals G1, ... Gm

- If the list is empty, terminate with success
- Otherwise look for the first clause in the program whose head matches G1.
- If none, terminate with failure.
- If yes, replace G1 with the goals in the body of the clause, making the same unifications that were

done to make G1 match the head of the clause. Recursively try to satisfy the list of goals. If this fails, look for another clause whose head matches G1.

Notice that for the procedural meaning, the order of clauses in the program, and the order of goals in the body of a rule, affects the execution of the program.

Dangers of infinite search (infinite looping):

The take_before rule was written as:

```
take_before(X,Z) :- prerequisite(X,Z).
take_before(X,Z) :- prerequisite(X,Y),
                    take_before(Y,Z).


Suppose instead it was written as:
take_before(X,Z) :- take_before(Y,Z),
                    prerequisite(X,Y).
```

Declaratively this is fine, but procedurally a take_before goal would get stuck in an infinite search.

Some list manipulation programs:

```
/* Definition of append (name changed to myappend to avoid colliding with
   built-in append rule) */
myappend([],Ys,Ys).
myappend([X|Xs],Ys,[X|Zs]) :- myappend(Xs,Ys,Zs).

/* SAMPLE GOALS */
| ?- myappend([1,2],[3,4,5],Q).

| ?- myappend([1,2],M,[1,2,3,4,5,6]).

| ?- myappend(A,B,[1,2,3]).

| ?- myappend(A,B,C).

/* DEFINITION OF MEMBER */
mymember(X,[X|_]).
mymember(X,[_|Ys]) :- mymember(X,Ys).


/* SAMPLE GOALS */
| ?- mymember(3,[1,2,3,4]).

| ?- mymember(X,[1,2,3,4]).

| ?- mymember(1,X).
```

Prolog warts:

- arithmetic
- cut
- negation
- assert, retract
- evaluable predicates

Arithmetic

In Prolog the operators +, *, and so forth just build up tree structures. So for example

```
X = 3+4, Y = 5*X.
```

succeeds with `X=3+4` and `Y=5*(3+4)`. Or try:

```
X = 3+4+5, A+B=X.
```

If you want to *evaluate* one of these tree structures, in other words do arithmetic, use the "is" operator. For example:

```
X is 3+4, Y is X*X.
```

A little later we'll see how to do this in a cleaner and more general way using one of constraint libraries for SWI Prolog.

Some simple arithmetic examples:

```
fahrenheit(C,F) :- F is 1.8*C+32.0.

myabs(X,X) :- X>=0.
myabs(X,X1) :- X<0, X1 is -X.

mymax(X,Y,X) :- X>=Y.
mymax(X,Y,Y) :- X<Y.

length of a list:

mylength([],0).
mylength([_|Xs],N1) :- mylength(Xs,N), N1 is N+1.

factorial(0,1).
factorial(N,F) :-
   N>0,
   N1 is N-1,
   factorial(N1,F1),
   F is N * F1.
```

More list examples

```
/* CLAUSES TO FIND ALL PERMUTATIONS OF A LIST */

permute([],[]).

permute([H|T],L) :-
  permute(T,U),
  insert(H,U,L).

/* insert an element X somewhere in list L */

insert(X,L,[X|L]).

insert(X,[H|T],[H|U]) :-
  insert(X,T,U).

/* inefficient sort */

badsort(L,S) :- permute(L,S), sorted(S).

sorted([]).
sorted([_]).
sorted([A,B|R]) :-
  A=<B,
  sorted([B|R]).

quicksort([],[]).
quicksort([X|Xs],Sorted) :-
  partition(X,Xs,Smalls,Bigs),
  quicksort(Smalls,SortedSmalls),
  quicksort(Bigs,SortedBigs),
  myappend(SortedSmalls,[X|SortedBigs],Sorted).

partition(_,[],[],[]).
partition(Pivot,[X|Xs],[X|Ys],Zs) :-
   X =< Pivot,
   partition(Pivot,Xs,Ys,Zs).
partition(Pivot,[X|Xs],Ys,[X|Zs]) :-
   X > Pivot,
   partition(Pivot,Xs,Ys,Zs).
```