Some Prolog Examples

1 Family Relationships

Here are some selected family relationships assuming a database of facts for the predicates `parent/2, male/1 and female/1`

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

sibling(X,Y) :- parent(P,Y), parent(P,X), X \= Y.

sister(X,Y) :- sibling(X,Y), female(X).

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

greatgrandparent(X,Y) :- parent(P,Y), grandparent(X,P).

entwined(X,Y) :- parent(X,Child), parent(Y,Child), X \= Y.

uncle(Uncle, N) :- parent(Par,N), sibling(Uncle,Par), male(Uncle).
uncle(Uncle, N) :- parent(Par,N), sibling(Par, Aunt), entwined(Aunt, Uncle), male(Uncle).

cousin(X,Y) :- grandparent(GP, X), grandparent(GP, Y), X \= Y, \+sibling(X,Y).

ancestor(X,Y) :- parent(X,Y).                 % Base case.
ancestor(X,Y) :- parent(P,Y), ancestor(X,P).  % Recursive rule.
```

2 Recursive Rules

The relationship exploited in the ancestor example occurs over and over again in Prolog programs. Here's a simple example for river networks.

```
drains_to(X,Y) :- tributary(X,Y).
drains_to(X,Z) :-
    tributary(X,Y), drains_to(Y,Z).

tributary(mississippi, gulf_of_mexico).
tributary(missouri, missippi).
tributary(ohio, mississippi).
tributary(kanawha, ohio). ...
```

Here is the classic recursive program, factorial:

```
factorial(N, Fact) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, Fact1),
    Fact is N * Fact1.

factorial(0, 1).
```

Often this is done with a cut (!) instead. A cut always succeeds and commits Prolog to all the choices that have been made from the point where the rule was applied up through the cut.

```
factorial(0, 1) :- !.   % Commit to this choice and ignore clauses below.
factorial(N, Fact) :-

    % N > 0 due to cut in previous clause
    N1 is N - 1,
    factorial(N1, Fact1),
    Fact is N * Fact1.
```

`Caution:` Beginners tend to overuse cuts that ruin the logical reading of the program. Often cuts can be eliminated by better program structure. No single clause should ever have more than one cut.

This next version is tail recursive, so it gets translated internally into a loop. In `fact_iter/3` the first argument is the loop control variable that counts down from `N`. The second argument is an accumulator that represents the product as it is being built up. The net result is a faster algorithm.

```
factorial(N,Fact) :-

    fact_iter(N, 1, Fact).

fact_iter(0, SoFar, SoFar) :- !. % When you get to 0, hand back the result.
fact_iter(N, SoFar, Ans) :-

    % N > 0, due to cut above.
    N1 is N - 1,
    SoFar1 is N * SoFar,
    fact_iter(N1, SoFar1, Ans).
```

No set of simple recursive programs is complete without the towers of Hanoi.

```
hanoi(N) :-

    move(N, source, dest, spare).

move(1, Source, Dest, _) :-

    write('Move disk from '),
    write(Source),
    write(' to '),
    write(Dest),
    nl.

move(N, Source, Dest, Spare) :-

    N > 1,
    N1 is N - 1,
    move(N1, Source, Spare, Dest),
    move(1, Source, Dest, Spare),
    move(N1, Spare, Dest, Source).
```

3 Temperature Conversion

This section illustrates some techniques for using Prolog as a "regular" programming language. The running example is a simple temperature conversion program. This first version is just the conversion predicate. It can be used interactively.

```
% temperature conversion.
convert(Cel, Fahr):-

    Fahr is 9.0 / 5.0 * Cel + 32.
```

This version adds I/O and a warning for high and low temps.

```
run :-

    write('Enter a temp in degrees Celsius '),
    read(C),
    convert(C,F),
    write('The temp is '), write(F), write(' Degrees Fahrenheit'),
    nl,
    warning(F, Warning),
    write(Warning).

convert(Cel, Fahr) :-

    Fahr is 9.0 / 5.0 * Cel + 32.

warning(T, 'It"s really hot out') :- T > 90.
warning(T, 'Brass monkey danger!') :- T < 30.
warning(T, '') :- T >= 30, T =< 90. % Blank warning for normal temps
```

Notice how a decision (if) is implemented by defining a predicate with multiple clauses. The three clauses of `warning/2` essentially act like three separate ifs. Although only one of the three rules will succeed, Prolog will always try all three (upon backtracking for alternative solutions).

The next example makes use of the cut operation (!) to indicate that the rules are mutually exclusive. This causes the three clauses to act together like an if-elif structure.

```
run :-

    write('Enter a temp in degrees Celsius '), read(C),
    convert(C,F),
    write('The temp is '), write(F), write(' Degrees Fahrenheit'), nl,
    warning(F, Warning),
    write(Warning).

convert(Cel, Fahr):-

    Fahr is 9.0 / 5.0 * Cel + 32.

warning(T, 'It"s really hot out') :- T > 90, !.
warning(T, 'Brass monkey danger!') :- T < 30, !.
warning(_, ''). % Default case.
```

The next version adds a sentinel loop using recursion. This loop will end when the user inputs something that is not a number. If the non-number is the atom `quit` Prolog will respond "yes"; any other non-number input will produce "no".

```
run :- get_input(C), convert_loop(C).


get_input(In) :-

    write('\nEnter a temp in degrees Celsius '),
    read(In).

convert_loop(C) :-

    number(C), % ensure C is a number
    convert(C,F),
    write('The temp is '), write(F), write(' Degrees Fahrenheit'), nl,
    warning(F, W),
    write(W), nl,
    get_input(C1),
    convert_loop(C1).

convert_loop(quit). % Succeed if user types quit


warning(T, 'It"s really hot out') :- T > 90, !.
warning(T, 'Brass monkey danger!') :- T < 30, !.
warning(_, '').
```

This final version is a repeat-fail loop. The result is similar to the previous program, except that it will re-prompt when presented with a bad input.

```
run :-

    repeat,
    write('\nEnter a temp in degrees Celsius '),
    read(C),
    process(C),
    C = quit. % Force redo if C was not quit

process(quit) :- !. % Do nothing for quit
process(C) :-

    number(C),
    convert(C,F),
    write('The temp is '), write(F), write(' Degrees Fahrenheit'),
    nl,
    warning(F,W),
    write(W),
    nl.

warning(T, 'It"s really hot out') :- T > 90, !.
warning(T, 'Brass monkey danger!') :- T < 30, !.
warning(_, '').
```

4 Structures and Lists

Terms can be composed of other more primitive terms. For example, data about classes might be organized into a fact like this:

```
class(cs373, time(mwf, 10:45, 11:50), instructor(zelle, john), location(sc, 345) ).
```

Notice that the last three arguments of this fact are not atoms, but complex terms. The identifiers `time`, `instructor`, and `location` are *functors*. They serve as a sort of label of a record that groups the fields that follow. The `time` functor groups three aruments, and `instructor` and `location` each group two arguments. Here are some example rules that can extract information from a database of facts like these. Notice how variables can be used to match entire terms or just sub-terms.

```
teacher(Last, First) :-

    class(_, _, instructor(Last, First), _).

teaches(Instructor, Class) :-

    class(Class,_,Instructor,_).

start_time(Class, Time) :-

    class(Class, time(_,Time,_), _, _).
```

A certain kind of logical structure is so handy that Prolog provides a special "syntactic sugar" for it. That structure is the list. Lists are indicated using square brackets. Lists can contain arbitrary terms, including other lists. Here are some examples of lists:

```
[tom, dick, harry]
[ward, june, [beaver, wally]]
[[big, large, huge], [small, tiny, miniscule, dimunitve]]
```

One way of generating a list is by asking Prolog to find all the solutions to a goal using the built-in `findall/3`. To create a list of all the men in the family database, we could do this:

```
?- findall(X, male(X), Men).
```

Lists are composed of a first element (the head) and the rest of the elements as a list (the tail). Prolog uses the notation `[Head | Tail]` to denote the head and tail of the list. For example:

```
?- [X|Y] = [1,2,3,4].
X = 1
Y = [2, 3, 4]
```

Here `X` is bound to the head of the list, and `Y` to the tail. The following are definitions of some "classic" list predicates. These are actually built-ins, but they can be simply defined.

```
% linear search for membership in list.
member(X, [X|_]).
member(X, [_|Tail]) :- member(X,Tail).

% length of a list
length([], 0).
length([_|T], L) :-
    length(T, L1),
    L is L1 + 1.

% iterative version of length
length1(List, L) :- length_iter(List, 0, L).

length_iter([], Count, Count).
length_iter([_|T], SoFar, Length):-
    SoFar1 = SoFar + 1,
    length_iter(T, SoFar1, Length).

% Append two lists
append([], List, List).
append([H|T], List, [H|T1]) :- append(T, List, T1).

% Merge two sorted lists.
merge([], List, List) :- !.
merge(List, [], List) :- !.
merge([H1|T1], [H2|T2], [H1|T3]) :-
    H1 =< H2,
    !,
    merge(T1, [H2|T2], T3).
merge(List1, [H2|T2], [H2|T3]) :-
    merge(List1, T2, T3).
```

Here are some more example list predicates that are not built-ins.

```
% insert(X, List, Result): Result is List with X inserted somewhere.
insert(X, List, [X|List]).
insert(X, [H|T], [H|T1]) :- insert(X,T,T1).

% permutation(L1, L2): L2 is a permutation (rearrangement of L1) permutation([],[]).
permutation([H|T], Result) :-
    permutation(T, T1),
    insert(H, T1, Result).
```

```
% sorted(List): List is arranged in nondecreasing order.
sorted([]).
sorted([_]).
sorted([H1,H2|Rest]) :-
    H1 =< H2,
    sorted([H2|Rest]).


% permsort(List, Sorted): Sorted is a ordered rearrangement of List
permsort(List, Sorted):-
    permutation(List, Sorted),
    sorted(Sorted).


% insert_ordered(Item, List, Result): list is an ordered list, and Result
%           is List with Item inserted at the proper (ordered) position.
insert_ordered(Item, [], [Item]) :- !.
insert_ordered(Item, [H|T], [Item,H|T]):- Item =< H, !.
insert_ordered(Item, [H|T], [H|T1]) :- insert_ordered(Item, T, T1).

% insort(List, Sorted): Sorted is an ordered rearrangement of List insort([],[]).
insort([H|T], Result) :-
    insort(T, Result1),
    insert_ordered(H, Result1, Result).


%mergesort(List, Sorted): Sorted is an ordered rearrangement of List mergesort([],[]) :- !.
mergesort([X], [X]):- !.
mergesort(List, Sorted):-
    split(List, List1, List2),
    mergesort(List1, Sort1),
    mergesort(List2, Sort2),
    merge(Sort1, Sort2, Sorted).


%split(L, List1, List2): List1 and List2 contain alternating elements of L.
split([], [], []).
split([H], [H], []).
split([H1,H2|Rest], [H1|T1], [H2|T2]) :- split(Rest, T1, T2).
```

5 Built-In Predicates from These Examples

Predicate	Meaning
`X \= Y`	`X` does not match `Y`
`\+ Goal`	Negation: Succeed only if Goal fails
`X is Expression`	Evaluate Expression and match the result with `X`
!	Cut: Succeed and commit to choices since matching this rule
`write(Term)`	Print `Term` on the standard output
nl	Print a newline to standard output
`read(C)`	Allows user to type a term and then match it with `C`
`repeat`	Succeeds initially and re-succeeds on each backtrack
`findall(Var, Goal, List)`	`List` is all instantiations of `Var` found in all demonstrations of `Goal`
`member(X, List)`	`X` is in List
`length(List, N)`	`N` is the number of elements in List
`append(L1, L2, L3)`	Appending `L2` onto `L1` produces `L3`
`merge(L1,L2,L3)`	`L3` is the merge of sorted lists `L1` and `L2`