Searching in PROLOG

Recursion is the primary control mechanism for Prolog programming, and the list structure is the primary data structure used for representing complex data.

Recursion Example

```
ancestor_of(Ancest,Descend) :-  parent_of(Ancest,Descend).  /* base case */
ancestor_of(Ancest,Descend) :-  parent_of(Z,Descend),      /* recursive case */
                                ancestor_of(Ancest,Z).
```

In-class Exercise: Trace `ancestor_of(X,d)` using the following facts.

```
parent_of(a,b).
parent_of(b,c).
parent_of(c,d).
parent_of(d,e).
```

In-class Exercise: Define a recursive `descendant_of(X,Y)`.

Prolog List Notation

In Prolog list elements are enclosed by brackets and separated by commas.

```
[1,2,3,4]
[[mary,joe],[bob,carol,ted,alice]]
[]
```

Another way to represent a list is to use the **head/tail notation** `[H|T]`. Here the head of the list, H, is separated from the tail of the list, T, by a vertical bar. The tail of a list is the original list with its first element removed. The tail of a list is always a list, even if it's the empty list.

In Prolog, the H|T notation is used together with unification to combine and break up lists. For example, suppose we have the following list:

```
[bob,carol,ted,alice]
```

Here's the various matches we would obtain using H|T:

```
[X|Y]        matches with X=bob Y=[carol,ted,alice]
[X,Y|Z]      matches with X=bob, Y=carol, Z=[ted,alice]
[X,Y,Z|W]    matches with X=bob, Y=carol, Z=ted W=[alice]
[X,Y,Z,W|V]  matches with X=bob, Y=carol, Z=ted,  W=alice and V=[]
[X,Y,Z,Y]    won't match because Y=carol and carol != alice
[X,Y,Z,W,V|U] won't match because the list does not contain 5 elements
```

We can also build lists using unification and H|T notation. Suppose L unifies with [X|Y] and X=bob and Y=[carol,ted,alice]. Then L=[bob,carol,ted,alice].

Recursive List Examples

In some Prolog environments the `member` predicate is not a built-in predicate and must be defined within your program. It takes the form `member(Element,List)` and evaluates to true if and only if Element is a member of List. The underscore (_) can be used as a **anonymous** or **don't care** variable, meaning we don't care what value it has. It's there solely for pattern-matching (unification) purposes.

```
member(X,[X|_]).          /* 1. Base case:      X is a member of the list headed by X */
member(X,[Y|L]) :-        /* 2.  Recursive case: X is a member of the list headed by Y */
   member(X,L).           /*                      if X is a member of that list's tail (L) */
```

Here's a trace of the member() predicate on the query: member(c,[a,b,c]).

```
member(c,[a,b,c]).
    call 1 (base case). fails, since c != a.
    call 2 (recursive case). X=c, Y=a, L=[b,c], member(c,[b,c]) ?
      call 1 (base). fails, since c != b.
      call 2 (recursive). X=c, Y=b, L=[c], member(c,[c]) ?
         call 1. Success, c = c.
      Yes to call 2. (backing out of recursion)
    Yes to call 2.   (backing out of recursion)
Yes.                 (original query succeeds)
```

Here's a trace of the member() predicate on the query: member(c,[a,b]).

```
member(c,[a,b]).
    call 1 (base case). fails, since c != a.
    call 2 (recursive case). X=c, Y=a, L=[b], member(c,[b]) ?
      call 1 (base). fails, since c != b.
      call 2 (recursive). X=c, Y=b, L=[], member(c,[]) ?
         call 1. fails, since [] does not match [X|_].
         call 2. fails, since [] does not match [Y|L].
      No to call 2. (backing out of recursion)
    No to call 2.   (backing out of recursion)
No.                 (original query succeeds)
```

The following predicate writes each element of a list using Prolog's built-in `write()` predicate and built-in `nl` (newline) predicate:

```
writelist([]).                                /* Base case: An empty list */
writelist([H|T]) :- write(H),nl,writelist(T).  /* Recursive case: */
```

The following predicate writes a list in reverse order:

```
reverse_writelist([]).                                          /* Base case: An empty list */
reverse_writelist([H|T]) :- reverse_writelist(T),write(H),nl.  /* Recursive case: */
```

The Knight's Tour

Suppose we represent the squares of a 3 x 3 chess board with the following notation:

```
1  2  3
4  5  6
7  8  9
```

Then the following list of predicates describe all of the legal moves that a knight can make on such a chess board:

```
move(1,6).   move(3,4).    move(6,7).   move(8,3).
move(1,8).   move(3,8).    move(6,1).   move(8,1).
move(2,7).   move(4,3).    move(7,6).   move(9,4).
move(2,9).   move(4,9).    move(7,2).   move(9,2).
```

Suppose you wanted to determine whether a path exists from one square to another using just the legal knight moves. Here's a recursive predicate for path:

```
path(Z,Z).
path(X,Y) :- move(X,W),not(been(W)),assert(been(W)),path(W,Y).
```

This version of `path()` uses the `assert()` predicate to maintain a list of visited states. This will prevent looping.

An alternative design would be to use a list to represent the visited states, and just carry the list along as a third parameter:

```
path(Z,Z,L).
path(X,Y,L) :- move(X,Z),not(member(Z,L)),path(Z,Y,[Z|L]).
```

The third parameter maintains a list of visited states. Note how the state Z is added to the list in the recursive call.

Exercise

Trace the following query: path(1,3,[1]).

The Cut Operator

The **cut** operator, which is represented by the exclamation point, !, is used to cut off backtracking. The syntax for cut is that of a goal with no arguments. It has two important side-effects:

- The cut operator always succeeds.
- If it is "failed back to", during backtracking, it causes the entire goal to fail.

One of the best uses of the cut operator is in the definition of Prolog's `not()` predicate, which recall represents **negation as failure**.

```
not(P) :- call(P),!,fail.     /* Call P. If it succeeds, fail.          */
not(P).                       /* If Call P fails, then not(P) succeeds. */
```