POPL 2(2020-04-13)

Srijith P K

List

```
list([]).
list([X|Xs]) :- list(Xs).


natlist([]).
natlist([N|Ns]) :- nat(N), natlist(Ns).
```

List operations

$$\frac{}{\mathsf{member}(X, \mathsf{cons}(X, Ys))}$$

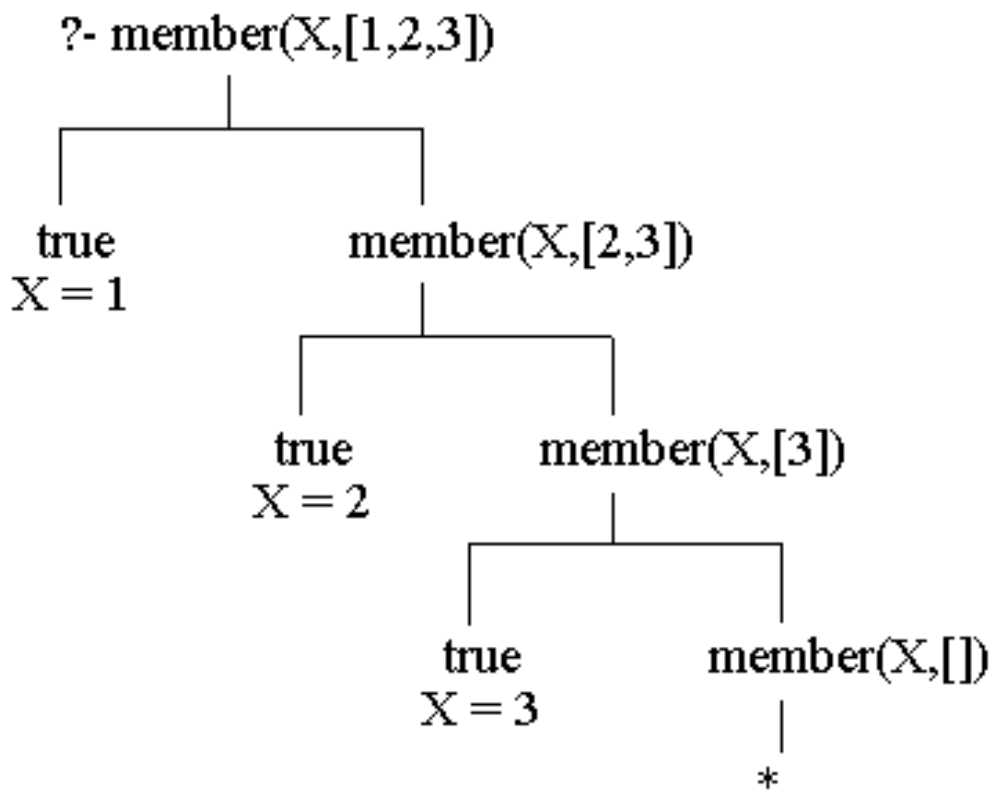
$$\frac{\mathsf{member}(X, Ys)}{\mathsf{member}(X, \mathsf{cons}(Y, Ys))}$$

```
member(X, [X|Ys]).
member(X, [Y|Ys]) :- member(X, Ys).
```

```
?- member(a, [a,b,a,c]).
```

```
?- member(X, [a,b,a,c]).
```

List membership



List operations

- Member predicate is inefficient : when we find the first matching element there is no need to traverse the remainder of the list, although the member predicate above will always do so.
- to only check membership, or find the first occurrence of an element in a list

List operations

- Member predicate is inefficient : when we find the first matching element there is no need to traverse the remainder of the list, although the member predicate above will always do so.
- to only check membership, or find the first occurrence of an element in a list

$$\frac{}{\mathsf{member}(X, \mathsf{cons}(X, Ys))} \qquad \frac{X \neq Y \quad \mathsf{member}(X, Ys)}{\mathsf{member}(X, \mathsf{cons}(Y, Ys))}$$

```
member1(X, [X|Ys]).
member1(X, [Y|Ys]) :- X \= Y, member1(X, Ys).
```

```
?- member1(a, [a,b,a,c]).
```

```
?- member1(X, [a,b,a,c]).
```

List operations

```
prefix([], Ys).
prefix([X|Xs], [X|Ys]) :- prefix(Xs, Ys).

?- prefix(Xs,[a,b,c,d]).
```

List operations

```
prefix([], Ys).
prefix([X|Xs], [X|Ys]) :- prefix(Xs, Ys).
```

```
?- prefix(Xs,[a,b,c,d]).
Xs = [];
Xs = [a];
Xs = [a,b];
Xs = [a,b,c];
Xs = [a,b,c,d].
```

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

```
prefix2(Xs, Ys) :- append(Xs, _, Ys).
```

Cut

- The **cut**, in Prolog, is a goal, written as !, which always succeeds, but cannot be backtracked.
- It is best used to prevent unwanted backtracking,
- on backtracking we do not attempt to use the second clause, it will be used only if first clause fails

```
minimum(X, Y, X) :- X =< Y.
minimum(X, Y, Y) :- X > Y.
```

```
minimum(X, Y, Z) :- X =< Y, !, Z = X.
minimum(X, Y, Y).
```

Conditionals

- Conditional construct in Prolog

```
minimum(X, Y, X) :- X =< Y.
minimum(X, Y, Y) :- X > Y.
```

- If A succeeds solve B else solve C

```
A -> B ; C
```

```
minimum(X, Y, Z) :- X =< Y -> Z = X ; Z = Y.
```

Conditionals as cuts

```
A -> B ; C
```

```
if_then_else(A, B, C) :- A, !, B.
if_then_else(A, B, C) :- C.
```

- A when it succeeds for the first time, and also
- commits to the first clause of if_then_else. B will create
- choice points and backtrack as usual, except when it fails the second clause of if_then_else will never be tried.
- A fails before the cut, then the second clause will be tried.

Conditionals as cuts

```
minimum(X, Y, X) :- X =< Y, !.
minimum(X, Y, Y).
```

Conditionals as cuts

```
minimum(X, Y, X) :- X =< Y, !.
minimum(X, Y, Y).
```

```
?- minimum(5,10,10).
```

Green cuts and Red cuts

- Green cuts are merely for efficiency, to remove redundant choice points, while red cuts change the meaning of the program entirely
- Green Red

```
minimum(X, Y, Z) :- X =< Y, !, Z = X.
minimum(X, Y, Y) :- X > Y.
```

```
minimum(X, Y, Z) :- X =< Y, !, Z = X.
minimum(X, Y, Y).
```

```
gamble(X) :- gotmoney(X),!.
gamble(X) :- gotcredit(X), \+ gotmoney(X).
```

```
gamble(X) :- gotmoney(X),!.
gamble(X) :- gotcredit(X).
```

Negation as Failure (NAF)

- To derive not A
- to negate A one tries to prove A (just executing it), and if A is proved, then its negation, not A fails during execution. If A fails during execution, then not A will succeed.

```
\+(A) :- A, !, fail.
\+(A).
```

- if A fails then \+(A) will succeed

```
unmarried_student(X):-
    not(married(X)), student(X).

student(joe).
married(john).
```

```
?- unmarried_student(joe).

?- unmarried_student(john).
```

Sorting

- Quicksort
- Comparison operation will fail if X or X0 are uninstantiated or not integers
- Mode and Type restrictions

```
quicksort([], []).
quicksort([X0|Xs], Ys) :-
        partition(Xs, X0, Ls, Gs),
        quicksort(Ls, Ys1),
        quicksort(Gs, Ys2),
        append(Ys1, [X0|Ys2], Ys).
```

Comparison

```
partition([], _, [], []).
partition([X|Xs], X0, [X|Ls], Gs) :-
        X =< X0, partition(Xs, X0, Ls, Gs).
partition([X|Xs], X0, Ls, [X|Gs]) :-
        X > X0, partition(Xs, X0, Ls, Gs).
```

Sorting

- Quicksort
- Comparison operation will fail if X or X0 are uninstantiated or not integers
- Mode and Type restrictions
- Mode means arguments to be ground on invocation and not variables
- Type means arguments have to be integers

```
quicksort([], []).
quicksort([X0|Xs], Ys) :-
        partition(Xs, X0, Ls, Gs),
        quicksort(Ls, Ys1),
        quicksort(Gs, Ys2),
        append(Ys1, [X0|Ys2], Ys).
```

```
partition([], _, [], []).
partition([X|Xs], X0, [X|Ls], Gs) :-
        X =< X0, partition(Xs, X0, Ls, Gs).
partition([X|Xs], X0, Ls, [X|Gs]) :-
        X > X0, partition(Xs, X0, Ls, Gs).
```