List Processing

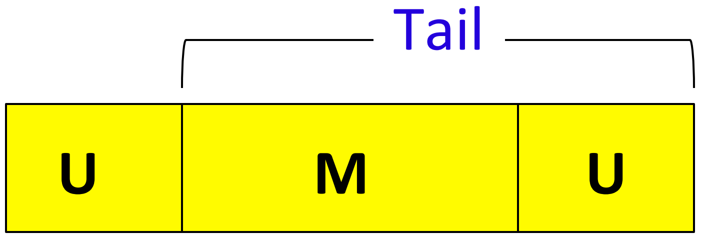
List processing

Lists are the most commonly-used data objects in Prolog, and relations on them usually require recursive programs.

EXAMPLE

To define a palindrome:

```
palin([ ]).
palin([U | Tail]) :-   append(M, [U], Tail), palin(M).
```



see: pt2_s2.pl

List processing: Palindrome Example

Alternatively:

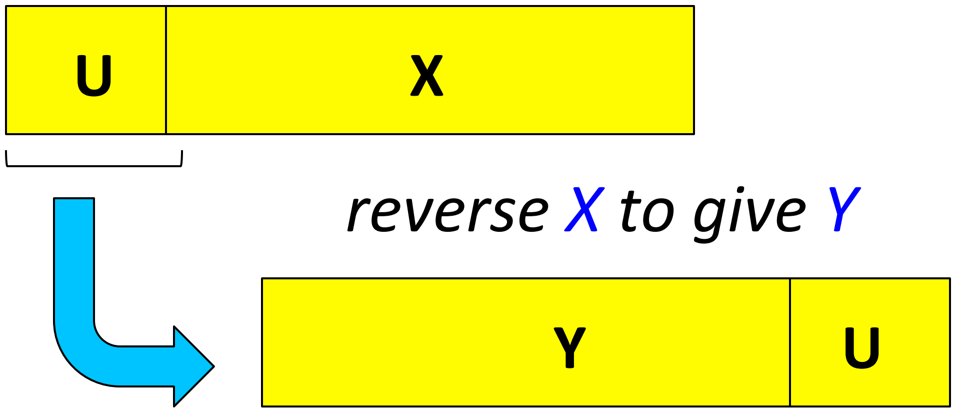
```
palin([ ]).
palin(L) :-    first(L, U), last(L, U),
               middle(L, M), palin(M).

first([U | _], U).

last([U], U).
last([_ | Tail], U) :- last(Tail, U).

middle([ ], [ ]).
middle([_], [ ]).
middle([_ | Tail], M) :- append(M, [_], Tail).
```

see: pt2_s3.pl

List processing: Reverse Example

To reverse a list:

```
reverse([ ], [ ]).
reverse([U | X], R) :-    reverse(X, Y),
                          append(Y, [U], R).
```



see: pt2_s4.pl

List processing: Reverse Example

Note that the program just seen is not *tail-recursive*.

If we try to force it to be so, by reordering the calls thus:

```
reverse([U | X], R) :-
        append(Y, [U], R),
        reverse(X, Y).
```

then the evaluation is likely to go infinite for some modes.

see: pt2_s5.pl

List processing: Membership

member(X, L) : X is a member of list L.
e.g.,

```
    member(2,[2,3])
    member(3,[2,3])
```

member/2 is provided in the Sicstus list primitives library.

Internally, defined like this:

```
    member(E, [E|T]).
    member(E, [_|T]):- member(E, T).
```

see: pt2_s6.pl

List processing: Membership

Some uses of member/2.

Find elements on a given list:

```
?-  member(X, [tom, dick, harry]).
    X = tom;
    X = dick;
    X = harry;
    no
```

Check if a given element is on a given list:

```
?-  member(1, [9,1,3]).
    yes
```

see: pt2_s6.pl

List processing: Membership

Some uses of member/2.

Insert element on a partially given list:

```
?-  member(1, [9,1,U]).
        true;  (equivalent to yes for all U)
        U=1;
        no
```

Generate templates for all lists on which a given element occurs.

```
?-  member(1, X).
    X = [1|Xs];
    X = [X1, 1|Xs];
    X = [X1, X2, 1|Xs];     (infinitely many answers)
```

see: pt2_s6.pl

A Combinatorial Puzzle

There are five houses in a line, each with an owner, a pet, a cigarette brand, a drink, and a colour.

The Englishman lives in the red house.
The Spaniard owns the dog.
Coffee is drunk in the green house.
The Ukrainian drinks tea.
The green house is immediately to the right of the ivory house.
The Winston smoker owns snails.
Kools are smoked in the yellow house.
Milk is drunk in the middle house.
The Norwegian lives in the first house on the left.
The man who smokes Chesterfields lives next to the man with the fox.
Kools are smoked in the house next to the house with the horse.
The Lucky Strike smoker drinks orange juice.
The Japanese smokes Parliaments.
The Norwegian lives next to the blue house.

A Combinatorial Puzzle

Who drinks water?

Who owns the zebra?

(Find the complete description of the row of houses.)

A Combinatorial Puzzle: One Solution

Not very efficient!!
But it works. (Try it)

```
query_water_zebra(H,W,Z) :-
  H = [house(norwegian,_,_,_,_),_,house(_,_,_,milk,_), _, _],
  member(house(englishman,_,_,_,red), H),
  member(house(spaniard,dog,_,_,_), H),
  member(house(_,_,_,coffee ,green), H),
  member(house(ukrainian,_,_,tea,_), H),
  followedBy(house(_,_,_,_,ivory), house(_,_,_,_,green), H),
  member(house(_,snails,winston,_,_), H),
  member(house(_,_,kools,_,yellow), H),
  nextTo(house(_,_,chesterfield,_,_), house(_,fox,_,_,_), H),
  nextTo(house(_,_,kools,_,_), house(_,horse,_,_,_), H),
  member(house(_,_,lucky_strike, orange_juice,_), H),
  member(house(japanese,_,parliaments,_,_), H),
  nextTo(house(norwegian,_,_,_,_), house(_,_,_,_,blue), H),
  member(house(W,_,_,water,_), H),
  member(house(Z,zebra,_,_,_), H).
```

see: pt2_s11.pl

Arithmetic Operations

Arithmetic

Arithmetic expressions use the standard operators such as

+ – * / *(besides others)*

Operands are simple terms or arithmetic expressions.

EXAMPLE

```
( 7 + 89 * sin(Y+1) ) /  ( cos(X) + 2.43 )
```

Arithmetic expressions must be *ground* at the instant Prolog is required to *evaluate* them.

Arithmetic expressions

E1 =:= E2	tests whether the values of E1 and E2 are equal
E1 =\= E2	tests whether the values of E1 and E2 are unequal
E1 < E2	tests whether the value of E1 is less than the value of E2

Likewise we have

E1 > E2	Tests whether E1 is greater than E2
E1 >= E2	Tests whether E1 is greater than or equal to E2
E1 =< E2	Tests whether E1 is less than or equal to E2

Arithmetic operations

The value of an arithmetic expression E may be computed and assigned to a variable X by the call

X is E

EXAMPLES

```
?- X  is  (2+2).        succeeds and binds X / 4
?- 4  is  (2+2).        succeeds
?- 4  is  (2+3).        fails
?- X  is  (Y+2).        gives an error
```

Arithmetic operations

Do not confuse ‘is’ with ‘=’.

X=Y means “X can be unified with Y”

EXAMPLES

?- X = (2+2).	succeeds and binds X / (2+2)
?- 4 = (2+2).	does not give an error, but *fails*
?- X = (Y+2).	succeeds and binds X / (Y+2)

The ‘is’ predicate is used *only* for the very specific purpose.

Template:

variable **is** arithmetic-expression-to-be-evaluated

Arithmetic operations: Example

Summing a list of numbers:

```
sumlist([ ], 0).
sumlist([ N | Ns], Total) :-
    sumlist(Ns, Sumtail),
    Total  is  N+Sumtail.
```

This is not *tail-recursive* - the query length will expand in proportion to the length of the input list.

see: pt2_s17.pl

Arithmetic operations: Example

Typical non-tail-recursive execution:

Initial query: ?- sumlist([2, 5, 8], T).

Derived queries:

```
?- sumlist([ 5, 8 ]), T is 2+T1.
?- sumlist([ 8 ], T2), T1 is 5+T2, T is 2+T1.
?- sumlist([ ], T3), T2 is 8+T3, T1 is 5+T2, T is 2+T1.
?- T2 is 8+0, T1 is 5+T2, T is 2+T1.
?- T1 is 5+8, T is 2+T1.
?- T is 2+13.
?- .
```

see: pt2_s17.pl

succeeds with the output binding T / 15

Arithmetic operations: Example

Doing it tail-recursively:

```
sumlist(Ns, Total) :- tr_sum(Ns, 0, Total).

tr_sum([ ], Total, Total).
tr_sum([ N | Ns ], S, Total) :-
      Sub is N+S,
      tr_sum(Ns, Sub, Total).
```

Here, tr_sum(Ns, S, T) means

$$T = S + \Sigma\ Ns$$

see: pt2_s19.pl

Arithmetic operations: Example

Typical tail-recursive execution:

```
?- sumlist([ 2, 5, 8 ], T).
 ?- tr_sum([ 2, 5, 8 ], 0, T).
 ?- Sub is 2+0, tr_sum([ 5, 8 ], Sub, T).
 ?- tr_sum([ 5, 8 ], 2, T).
 :
 ?- tr_sum([ 8 ], 7, T).
 :
 ?- tr_sum([ ], 15, T).
 ?- .
```

This again succeeds with T / 15.

see: pt2_s19.pl

Here the query length never exceeds two calls and each derived query can overwrite its predecessor in memory.