1

List Notation

To form lists use the special constant symbol `[]` (nil), the special functor `cons` and the dot notation introduced for Lisp lists. A bracket notation also available

infix	dot functor	cons functor
`(a.[])`	`.(a, [] )`	`cons(a, [])`
`(a.(b.[]))`	`.(a,.(b,[]))`	`cons(a,cons(b,[]))`

In list notation, list items are separated by commas and delimited by square brackets. A vertical bar indicates that the next item is the `rest` of the list:

infix	list1	list2
`(a.[])`	`[a]`	`[a|[]]`
`(a.(b.[]))`	`[a,b]`	`[a|[b]]`
`(a.(b.(c.[])))`	`[a,b,c]`	`[a|[b,c]]`
		= `[a,b|[c]]`
		= `[a,b,c|[]]`
		= `[a,b,c]`

©1993 John Mylopoulos

2

List Manipulation - `consP`

How do we construct a list in a fashion similar to the `cons` function in Lisp? We could define the predicate **`consP`**:

`consP(H,T,[H|T])` or `consP(H, T, H.T)`

For example:

```
?- consP(a, [b], X).
X = [a,b]  creates a list from a and [b]
?-  consP(X, Y, [a,b,c]).
X = a      determines head and tail
Y = [b, c]
?- consP(X, Y, Z).
X = H
Y = T         creates a list with unknown
Z = [H | T]   head and tail
?- consP(_, _, a * b).
              checks whether a*b is a
              non-empty list
no            (_ is an anonymous variable)
?- consP(a, [b], [a, b]).
              verifies that a is the head,
              and [b] is the tail of [a,b]
yes
```


©1993 John Mylopoulos

3

List Manipulation

The predicate `append`

`append(X,Y,Z)` - `Z` is the result of appending `Y` to `X`

```
append([], Y, Y).
append(cons(E,X),Y,cons(E,Z))
              :- append(X, Y, Z).
```

[very much in the spirit of the Lisp definitions given earlier!]

```
?-append([],cons(a,[]),V1)
   succeeds with V1 = cons(a, [])

 ?-append(cons(a,cons(b,[])),
          cons(c,cons(d,[])),V2)
   succeeds with
      V2 = cons(a,
           cons(b,cons(c,cons(d,[]))))
```

How???

4

The Predicate `append`

```
append([], Y, Y).
append(cons(E,X),Y,cons(E, Z))
                  :- append(X, Y, Z).
?-append(cons(a,cons(b,[])),
         cons(c,cons(d,[])),V)
```

```
E1 = a
X1 = cons(b,[])
Y1 = cons(c,cons(d,[]))
V = cons(a,Z1)
```

```
?-append(cons(b,[])),
     cons(c,cons(d,[])), Z1)
```

```
E2 = b
X2 = []
Y2 = cons(c,cons(d,[]))
Z1 = cons(b,Z2)
```

```
?-append([],cons(c,cons(d,[])),Z2)
```

```
Y2 = cons(c,cons(d,[]))
Z2 = cons(c,cons(d,[]))
```

■

```
V = cons(a,Z1) = cons(a,cons(b,Z2))
   = cons(a,cons(b,cons(c,cons(d,[]))))
```

5

The Predicate append

```
append([], Y, Y).
append([E|X],Y,[E|Z]) :- append(X,Y,Z).
```

```
?- append([a,b], [c,d], V)
     E1 = a
     X1 = [b]
     Y1 = [c, d]
     V  = [a | Z1]

?- append([b],[c,d], Z1)
     E2 = b
     X2 = []
     Y2 = [c, d]
     Z1 = [b | Z2]

?- append( [], [c, d], Z2)
     Y2 = [c, d]
     Z2 = [c, d]
```

```
V = [a|Z1] = [a|[b|Z2]] =
      [a|[b|[c,d]]] = [a, b, c, d]
```

©1993 John Mylopoulos

6

The Predicate append

```
|?- [user].
| append([], Y, Y).
| append([E|X],Y,[E|Z]) :- append(X,Y,Z).
| ^D
yes

|?- append([a, b, c], [d, e], V).
V= [a, b, c, d, e]
yes

|?- append([a,X,c],[Y,e], [a,b,Z,d,e]).
X =
Y =
Z =           ;
no

| ?- append(U, V, [a, b, c]).
U = []
V = [a, b, c];
U = [a]
V = [b, c];
U = [a, b]
V = [c];
U = [a, b, c]
V = [];
no
```

©1993 John Mylopoulos

7

Using the Predicate `append`

Other uses of `append`

- Find last element of a list

```
last( L, E) :- append( _, [E], L).
```

- Determine if something is a member of a list

```
member(E,L) :- append(?,??,???)
```

- Prefix of a list, e.g., `prefix([a,b],[a,b,c,d])`

```
prefix(L1,L2) :- append(?,??,???)
```

- Suffix of a list
- Middle of a list

```
middle(L1,L2) :-
   append(A,L1,V), append(V,B,L2).
```

What's happening?

Non-procedural nature of `append` definition allows us to use it many different ways!

©1993 John Mylopoulos

8

Equality of Terms

- `equal(`*term1*`, `*term2*`)` or *term1* `=` *term2*
  - trivial to define, unification does all the work!

```
equal(X, X).
```

```
| ?- equal([a, b, [c, d]], [a, b, [c, d]]).
yes
```

```
| ?- equal([a, X | Z], [a, b, [c, d]]).
X = b
Z = [[c, d]]
```

Note that equality succeeds iff its two arguments are unifiable, not just equal

- `term1 == term2`
  - true if the two terms are identical

```
?- [a, X] == [a, X]     succeeds
?- [a, X] == [a, b]     fails
?- [a, X] == [a, Y]     fails
```

- `term1 \== term2`
  - true if the two terms are not identical

©1993 John Mylopoulos

9

List Manipulation - `member`

A value is a member of a list
if the head of the list is equal to the value
or
if the rest of the list contains the value as a member.

In simple Prolog:

```
member(Element,List) :-
List = [Head | Tail], Head = Element.

member(Element,List) :-
List = [Head|Tail],member(Element, Tail).
```

As it is usually defined:

```
member(Element,[Element | Tail]).
member(Element, [Head | Tail]) :-
         member(Element,Tail).
```

or more compactly still:

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


10

List Manipulation - `member`

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

Tracing **`member:`**

```
| ?- trace, member(b, [a, b, c] ).
   (2) 1 Call: member(b, [a, b, c] )  ?
   (3) 2 Call: member(b, [b, c] ) ?
   (3) 2 Exit: member(b, [b, c] )
   (2) 1 Exit: member(b, [a, b, c] )
yes

| ?- trace,member(x, [a,b,c] ).
   (2) 1 Call: member(x, [a, b, c] )  ?
   (3) 2 Call: member(x, [b, c] ) ?
   (4) 3 Call: member(x, [c] ) ?
   (5) 4 Call: member(x, [] ) ?
   (5) 4 Fail: member(x, [] )
   (4) 3 Fail: member(x, [c] )
   (3) 2 Fail: member(x, [b, c] )
   (2) 1 Fail: member(x, [a, b, c] )
no
```

11

List Manipulation - `member`

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

| ?- trace,member(X,[a, b, c]).
   (2) 1 Call: member(_0,[a, b, c]) ?
   (2) 1 Exit: member(a, [a, b, c])
X = a ;
   (2) 1 Back to: member(_0, [a, b, c]) ?
   (3) 2 Call: member(_0, [b, c]) ?
   (3) 2 Exit: member(b, [b, c])
   (2)  1  Exit: member(b, [a, b, c]).
X= b;
   (3) 2 Back to: member(_0, [b, c]) ?
   (4) 3 Call: member(_0, [c]) ?
   (4) 3 Exit: member(c, [c])
   (3) 2 Exit: member(c, [b, c])
   (2)  1  Exit: member(c,[a, b, c]).
X= c;
   (4) 3 Back to : member(_0, [c]) ?
   (5) 4 Call: member(_0, []) ?
   (5) 4 Fail: member(_0, [])
   (4) 3 Fail: member(_0, [c])
   (3) 2 Fail: member(_0, [b, c])
   (2) 1 Fail: member(_0, [a, b, c])
no
```


©1993 John Mylopoulos

12

List Manipulation - `member`

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

|?- trace,member(a, L).
   (2) 1 Call: member(a, _0) ?
   (2) 1 Exit: member(a, [a | _13])
L = [a | _13] ;

   (2) 1 Back to: member(a, _0) ?
   (3) 2 Call member(a, _13) ?
   (3) 2 Exit: member (a, [a | _17] )
   (2) 1 Exit:  member(a,[_12, a | _17])
L = [_12, a  | _17] ;

   (3) 2 Back to: member(a, _13)
   (4) 3 Call: member(a,_17)
   (4) 3 Exit: member(a,[a | _21])
   (3) 2 Exit: member(a,[_16, a | _21])
   (2) 1 Exit: member(a,[_12,_16,a |_21])
L = [_12, _16, a | _21]

   yes
```


©1993 John Mylopoulos

13

List Manipulation - `member`

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

Does the order of the clauses change the meaning? Consider predicate `otherMember`, derived from `member` by exchanging the two clauses:

```
otherMember(X, [ _ |  Y] ) :-
                    otherMember(X,Y).
otherMember(X, [X | _ ] ).

|?- otherMember(c, [a, b, c] ).
yes                          same results

|?- otherMember(x, [a, b, c] ).
no.                          same results

|?- otherMember(X, [a, b, c] ).
X = c;                 same results
X = b;                 but in reverse order
X = a;
no

|?- otherMember(a, X).
! Out of local stack during execution
[ execution aborted]     infinite loop!
```


©1993 John Mylopoulos

14

Permuting

`permute(X, Y)` holds iff `Y` is a permutation of `X`

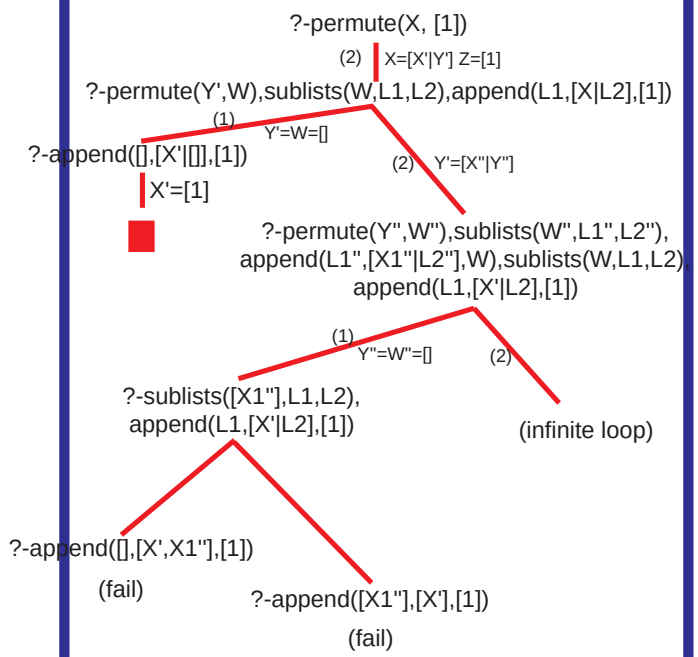
```
permute([], []).
permute([X | Y], Z) :-
     permute(Y, W), sublists(W, L1, L2),
     append( L1, [X | L2], Z).
sublists(X, L1, L2) :- append(L1, L2, X).

?-permute([1,2],X)    %  returns perms of
                      %  [1, 2], then
                      %  returns 'no' if
                      %  prompted for more
?- permute([2], W),
    sublists(W,L1,L2),append(L1,[1|L2],X)
?- permute([],W1),sublists(W1,L1',L2'),
append(L1',[2|L2'],W),sublists(W,L1,L2),
   append(L1, [1 | L2], X)
?- sublists([],L1',L2'),
             append(L1',[2| L2'], W),
   sublists(W,L1,L2),append(L1,[1|L2],X)
?- append([],[2|[]],W),sublists(W,L1,L2),
             append(L1,[1|L2],X)
?-sublists([2],L1,L2),append(L1,[1|L2],X)
```

`Sublists([2],...)` will generate two decompositions of `[2]`, leading to two solutions of `permute([1, 2],...)`

©1993 John Mylopoulos

15

But `permute` doesn't always behave as expected:



After generating the correct answer, `permute` will go into an infinite loop when prompted for more answers

16

Sorting

`sort(X, Y)` holds iff `Y` is a permutation of `X` and `Y` is ordered

```
sort(X, Y) :- permute(X, Y), ordered(Y).
permute([], []).
permute([X | Y], Z) :- permute(Y, W),
     sublists(W, L1, L2),
        append( L1, [X | L2], Z).
sublists(X,L1,L2) :- append(L1,L2,X).
ordered([X,Y|Z]) :- lessThan(X,Y),
                    ordered([Y | Z).
```

Very simple, but also very inefficient!

- n! permutations will be generated, for a list of length n

```
?- sort([3, 1, 2], Y)
   ?- permute({3, 1, 2], Y), ordered(Y)
        Y/[3, 1, 2]
        Y/[3, 2, 1]
        Y/[2, 3, 1]
        Y/[2, 1, 3]
        Y/[1, 3, 2]
        y/[1, 2, 3]
```

17

Quick Sort

- **Algorithm:**
  - **Choose item in list as pivot**
  - **Put all items less than pivot in L1**
  - **Put all items greater than pivot in L2**
  - **Sort L1, and L2**
  - **Append the results (L1, pivot, L2)**

```
qsort([], []).
qsort([Pivot | Rest], Sorted) :-
   part(Pivot, Rest, L1, L2),
   qsort(L1,Lessers),qsort(L2,Greaters),
   append(Lessers,
              [Pivot|Greaters],Sorted).

part(Pivot, [], [], []).
part(Pivot,[Item|Rest],[Item|L1],L2) :-
     lessThan(Item,Pivot),
          part(Pivot, Rest, L1, L2).
part(Pivot,[Item|Rest],L1,[Item|L2]) :-
   notLessThan(Item,Pivot),
          part(Pivot, Rest, L1, L2).
```

©1993 John Mylopoulos

18

Arithmetic without Numbers

How can numbers be represented in logic without making use of the built-in arithmetic predicates?

It is possible to represent natural numbers by taking a distinguished symbol to represent zero (say 0), and defining a natural number to be 0, or the successor of a natural number:

```
successor(X,s(X)).

naturalNumber(0).
naturalNumber(Successor) :-
   successor(Pred,Successor),
      naturalNumber(Pred).
```

Or more compactly:

```
naturalNumber(0).
naturalNumber(s(X)):- naturalNumber(X).
```

```
| ?- naturalNumber(Z).
Z = 0;
Z = s(0);
Z = s(s(0));
...
```

©1993 John Mylopoulos

19

Arithmetic in Prolog

How to perform addition? Define a three-place predicate `+(LHS,RHS,Result)` that says:

if `LHS` is 0
then `Result` is `RHS`
otherwise, `LHS` is a successor to some `X`,
and `Result` is the successor to `(+ X RHS)`

```
+(LHS,RHS,Result) :- LHS=0,RHS=Result.
+(LHS,RHS,Result) :- successor(LHS,X)),
   +(X,RHS,RES),successor(RES,Result).
```

or more compactly:

```
+(0, RHS, RHS).
+(s(X), RHS, s(RES) ) :- +(X,RHS,RES).
```

This addition predicate also performs subtraction!

Try it with trace !

20

Multiplication in Prolog

```
times(0,X,0).
times(s(X),Y,Z) :-
        times(X,Y,W),
        plus(W,Y,Z).

plus(0,X,X) :- natural_number(X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).

natural_number(0).
natural_number(s(X)) :- natural_number(X).

?- plus(s(s(s(0))),s(s(0)),X).

X = s(s(s(s(s(0)))))

?- times(s(s(s(0))),s(s(0)),X).

X = s(s(s(s(s(s(0))))))
```

21

Arithmetic without Numbers

We could also do arithmetic with lists, by representing a natural number by a list of the same length:

```
0  []          or       []
1  [[]]                 [[]]
2  [[], [] ]            [[[]]]
3  [[], [], []]         [[[[]]]]
```
etc.

```
el([]).              nl([]).
el([[]|T]):-el(T).   nl([L]):-nl(L).
```

Then addition and subtraction is just the same as the list `append` function.

This approach is useful if we are dealing with the traversal of a structure, where the number of operations corresponds to the number of substructures;

but...

clearly not meant for real arithmentic!!!.

©1993 John Mylopoulos