Lists

Lists

- A list is written as a sequence of values, known as *list elements*, separated by commas and enclosed in square brackets, e.g. **[dog,cat,fish,man]**.
- A list element does not have to be an atom. It can be any Prolog term, including a bound or unbound variable or another list, so **[X,Y,mypred(a,b,c),[p,q,r],z]** is a valid list
- A list element that is itself a list is known as a *sublist*.
- Lists can have any number of elements, including zero
- The list with no elements is known as the *empty list* and is written as **[]**.

Lists...

- For non-empty lists, the first element is known as the *head*.
- The list remaining after the first element is removed is called the *tail*
- The head of the list **[dog,cat,fish,man]** is the atom *dog* and the tail is the list **[cat,fish,man]**.
- The head of list **[x,y,mypred(a,b,c),[p,q,r],z]** is the atom *x*. The tail is the list **[y,mypred(a,b,c),[p,q,r],z]**.

Notation for Lists

- **Standard Bracketed Notation -** lists are written as a sequence of list elements written in order, separated by commas and enclosed in square brackets.
- Lists are most valuable when the number of elements needed cannot be known in advance and would probably vary from one use of the program to another.

Notation for Lists…

- **The 'cons' (list constructor) notation -** A list is written with two parts joined together by the vertical bar character | which is known as the ***cons character*** or simply as ***cons***.
- Thus a list is represented by the notation **[** ***elements*** **|** ***list*** **]**.
- ***Elements*** is a sequence of one or more list elements, which may be any Prolog terms
- ***list*** represents a list.
- For example, **[one|[two,three]]** represents **[one,two,three]**.

Notation for Lists…

- Below are equivalent ways of writing the same list of four elements:
  - **[alpha,beta,gamma,delta]**
  - **[alpha|[beta,gamma,delta]]**
  - **[alpha,beta|[gamma,delta]]**
  - **[alpha,beta,gamma|[delta]]**
  - **[alpha,beta,gamma,delta|[]]**
  - **[alpha,beta|[gamma|[delta|[]]]]**

Example 1

- If variable *L* is bound to a list, say **[red,blue,green,yellow]**, we can represent a new list with the atom **brown** inserted before the elements already there by **[brown|L]**.

?- L=[red,blue,green,yellow],write([brown|L]),nl.

Output:

[brown,red,blue,green,yellow]

Example 2

- If variable *A* is bound to the list **[brown,pink]** and variable *L* is bound to the list **[red,blue,green,yellow]**, the list **[A,A,black|L]** represents:

[[brown,pink],[brown,pink],black,red,blue,green,yellow].

Example 3

- A new list *L1* created from a list *L* input by the user.

?- write('Type a list'),read(L),L1=[start|L],write('New list is '),write(L1),nl.

Type a list: [[london,paris],[x,y,z],27].

New list is [start,[london,paris],[x,y,z],27]

Decomposing a List

- The predicate **writeall** writes out the elements of a list, one per line.

```
/* write out the elements of a list, one per line */
writeall([]).
writeall([A|L]):- write(A),nl,writeall(L).
```

- The second clause of **writeall** separates a list into its head *A* and tail *L*, writes out *A* and then a newline, then calls itself again recursively
- The first clause of **writeall** ensures that evaluation terminates when no further elements of the list remain to be output.

Decomposing a List…

?- writeall([alpha,'this is a string',20,[a,b,c]]).

Output:

alpha

this is a string

20

[a,b,c]

yes

Decomposing a List…

- Note that although **writeall** takes a list as its argument, its definition does not include a statement beginning: **writeall(L):-**
- Instead, the main part of the definition begins: **writeall([A|L]):-**

Decomposing a List…

```
write_english([]).
write_english([[City,england]|L]):-
   write(City),nl,
   write_english(L).
write_english([A|L]):-write_english(L).
go:- write_english([[london,england],[paris,france],
   [berlin,germany],[portsmouth,england],
   [bristol,england],
   [edinburgh,scotland]]).
```


Decomposing a List…

- The predicate **write_english** defined above takes as its argument a list such as

  [[london,england],[paris,france],[berlin,germany],[portsmouth,england],[bristol,england],[edinburgh,scotland]]

- Each element is a *sublist* containing the name of a city and the name of the country in which it is located.
- Calling **write_english** causes the names of all the cities that are located in England to be output.

Decomposing a List…

- The second clause of **write_english** deals with those sublists that have the atom *england* as their second element
- In this case the first element is output, followed by a new line and a recursive call to **write_english**, with the tail of the original list as the argument
- Sublists that do not have *england* as their second element are dealt with by the final clause of **write_english**, which does nothing with the sublist but makes a recursive call to **write_english**, with the tail of the original list as the argument.

Decomposing a List…

?- go.

london

portsmouth

bristol

yes

Decomposing a List…

- The predicate **replace** defined as:

  replace([A | L],[first | L]).

  takes as its first argument a list of at least one element
- If the second argument is an unbound variable, it is bound to the same list with the first element replaced by the atom **first**
- Using the 'cons' notation, the definition takes only one clause

Decomposing a List…

?- replace([1,2,3,4,5],L).

L = [first,2,3,4,5]

?- replace([[a,b,c],[d,e,f],[g,h,i]],L).

L = [first,[d,e,f],[g,h,i]]

Built-in Predicate: member

- The **member** built-in predicate takes two arguments
- If the first argument is any term except a variable and the second argument is a list
- **member** succeeds if the first argument is a member of the list denoted by the second argument (i.e. one of its list elements).

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

Yes

Built-in Predicate: member…

?- member(mypred(a,b,c),[q,r,s,mypred(a,b,c),w]).

Yes

?- member(x,[]).

No

?- member([1,2,3],[a,b,[1,2,3],c]).

yes

Built-in Predicate: member…

- If the first argument is an unbound variable, it is bound to an element of the list working from left to right
- This can be used in conjunction with backtracking to find all the elements of a list in turn from left to right, as follows:

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

X = a ;

X = b ;

X = c ;

no

Example 4

- Predicate **get_answer2** defined as:

```
get_answer2(Ans):-repeat,
   write('answer yes, no or maybe'),read(Ans),
   member(Ans,[yes,no,maybe]),
   write('Answer is '),write(Ans),nl,!.
```

reads a term entered by the user. It loops using **repeat**, until one of a list of permitted answers (*yes*, *no* or *maybe*) is entered and the **member** goal is satisfied.

Example 4…

?- get_answer2(X).

answer yes, no or maybe: possibly.

answer yes, no or maybe: unsure.

answer yes, no or maybe: maybe.

answer is maybe

X = maybe

Built-in Predicate: length

- The **length** built-in predicate takes two arguments. The first is a list. If the second is an unbound variable it is bound to the length of the list, i.e. the number of elements it contains.

?- length([a,b,c,d],X).

X = 4

?- length([[a,b,c],[d,e,f],[g,h,i]],L).

L = 3

?- length([],L).

L = 0

Built-in Predicate: length…

- If the second argument is a number, or a variable bound to a number, its value is compared with the length of the list.

  ?- length([a,b,c],3).

  yes

  ?- length([a,b,c],4).

  no

  ?- N is 3,length([a,b,c],N).

  N = 3

Built-in Predicate: reverse

- The **reverse** built-in predicate takes two arguments. If the first is a list and the second is an unbound variable (or vice versa), the variable will be bound to the value of the list with the elements written in reverse order, e.g.

  ?- reverse([1,2,3,4],L).

  L = [4,3,2,1]

  ?- reverse(L,[1,2,3,4]).

  L = [4,3,2,1]

Built-in Predicate: reverse…

?- reverse([[dog,cat],[1,2],[bird,mouse],[3,4,5,6]],L).

L = [[3,4,5,6],[bird,mouse],[1,2],[dog,cat]]

- Note that the order of the elements of the sublists **[dog,cat]** etc. is not reversed.
- If both arguments are lists, **reverse** succeeds if one is the reverse of the other.

  ?- reverse([1,2,3,4],[4,3,2,1]).

  yes

Built-in Predicate: reverse…

- **?- reverse([1,2,3,4],[3,2,1]).**
- **no**

Example 5

- The predicate **front/2** defined as:

```
front(L1,L2):-
   reverse(L1,L3),remove_head(L3,L4),reverse(L4,L2).
remove_head([A|L],L).
```

takes a list as its first argument

- If the second argument is an unbound variable it is bound to a list which is the same as the first list with the last element removed

Example 5…

- For example if the first list is **[a,b,c]**,the second will be **[a,b]**
- In the body of the rule the first list *L1* is reversed to give *L3*. Its head is then removed to give *L4* and *L4* is then reversed back again to give *L2*.

?- front([a,b,c],L).

L = [a,b]

?- front([[a,b,c],[d,e,f],[g,h,i]],L).

L = [[a,b,c],[d,e,f]]

Example 5…

- The **front** predicate can also be used with two lists as arguments. In this case it tests whether the second list is the same as the first list with the last element removed.

?- front([a,b,c],[a,b]).

yes

?- front([[a,b,c],[d,e,f],[g,h,i]],[[a,b,c],[d,e,f]]).

yes

?- front([a,b,c,d],[a,b,d]).

no

Built-in Predicate: append

- The term *concatenating* two lists means creating a new list, the elements of which are those of the first list followed by those of the second list
- Concatenating **[a,b,c]** with **[p,q,r,s]** gives the list **[a,b,c,p,q,r,s]**.
- Concatenating **[]** with **[x,y]** gives **[x,y]**.
- The **append** built-in predicate takes three arguments. If the first two arguments are lists and the third argument is an unbound variable, the third argument is bound to a list comprising the first two lists concatenated.

Built-in Predicate: append…

?- append([1,2,3,4],[5,6,7,8,9],L).

L = [1,2,3,4,5,6,7,8,9]

?- append([],[1,2,3],L).

L = [1,2,3]

?- append([[a,b,c],d,e,f],[g,h,[i,j,k]],L).

L = [[a,b,c],d,e,f,g,h,[i,j,k]]

Built-in Predicate: append…

- The **append** predicate can also be used in other ways. When the first two arguments are variables and the third is a list it can be used with backtracking to find all possible pairs of lists which when concatenated give the third argument, as follows:

?- append(L1,L2,[1,2,3,4,5]).

L1 = [] ,

L2 = [1,2,3,4,5] ;

L1 = [1] ,

L2 = [2,3,4,5] ;

Built-in Predicate: append…

L1 = [1,2] ,

L2 = [3,4,5] ;

L1 = [1,2,3] ,

L2 = [4,5] ;

- **L1 = [1,2,3,4] ,**
- **L2 = [5] ;**
- **L1 = [1,2,3,4,5] ,**
- **L2 = [] ;**
- **no**

Built-in Predicate: append…

- This example shows a list broken up in a more complex way.

?- append(X,[Y|Z],[1,2,3,4,5,6]).

X = [] ,

Y = 1 ,

Z = [2,3,4,5,6] ;

X = [1] ,

Y = 2 ,

Z = [3,4,5,6] ;

X = [1,2] ,

Y = 3 ,

Built-in Predicate: append…

```
Z = [4,5,6] ;
X = [1,2,3] ,
Y = 4 ,
Z = [5,6] ;
X = [1,2,3,4] ,
Y = 5 ,
Z = [6] ;
X = [1,2,3,4,5] ,
Y = 6 ,
Z = [] ;
no
```


END