Programming in Prolog
Lists

Romain Barnoud

Thanks to:
Dr Fariba Sadri
Claudia Schulz

Introduction

Lists are useful to represent sequences or collections of things.

```
dept(eng, aero).
dept(eng, bio_eng).
dept(eng, computing).
dept(eng, eee).
dept(eng, mech_eng).
dept(nat_sci, chemistry).
dept(nat_sci, maths).
dept(nat_sci, physics).
dept(business, finance).
dept(business, management).
```

```
dept(eng,
     [aero, bio_eng, computing,
      eee, mech_eng]).
dept(nat_sci, [chemistry,
      maths, physics]).
dept(business,
     [finance, management]).
```

```
temp(171113, 0000, 16).
temp(171113, 0300, 14).
temp(171113, 0600, 10).
temp(171113, 0900, 11).
temp(171113, 1200, 16).
temp(171113, 1500, 17).
temp(171113, 1800, 14).
temp(171113, 2100, 12).
```

```
temp(171113, [16, 14, 10, 11,
              16, 17, 14, 12]).
```

N.B.

The elements in a list can be any Prolog term (including a list), *e.g.*
`'[a, 1, f(X,Y), [4,Z,6], 2.0]'`.

Definition

A list is a data structure that represent a sequence of any number of terms.

A Prolog list is one of the following:

- `'[]'` called the ***empty list***.
- `'[H|T]'` where `H` is a term and `T` is a list (recursive definition).
  `H`, the first item of the list, is called the ***head***.
  `T`, the remaining of the list, is called the ***tail***.

Abbreviated notation:

```
[a|[b|[c|[d|[]]]]]  ≡ [a,b,c,d]
                    ≡ [a|[b,c,d]]
                    ≡ [a,b|[c,d]]
                    ≡ [a,b,c|[d]]
                    ≡ [a,b,c,d|[]]
```

Head and Tail – Examples

List	Head	Tail
`[]`		
`[1, 2, 3]`		
`[a]`		
`[[b]]`		
`[[]]`		
`[[2.0, c], 42, [], [X,y,f(Z)]]`		

Head and Tail – Examples

List	Head	Tail
`[]`	undef.	undef.
`[1, 2, 3]`	`1`	`[2, 3]`
`[a]`	`a`	`[]`
`[[b]]`	`[b]`	`[]`
`[[]]`	`[]`	`[]`
`[[2.0, c], 42, [], [X,y,f(Z)]]`	`[2.0, c]`	`[42, [], [X,y,f(Z)]]`

Lists and Unification

List 1	List 2	Substitution (if existing)
`[]`	`[X]`	
`[Y]`	`[A|B]`	
`[a,b,c]`	`[Y1,Y2]`	
`[a,b,c]`	`[Y1|Y2]`	
`[a,b,c]`	`[Z1,Z2,Z3]`	
`[a,b,c]`	`[Z1,Z2|Z3]`	
`[[1,2],[3,4]]`	`[H|T]`	
`[[1,2],3]`	`[[X|Y]|Z]`	

Lists and Unification

List 1	List 2	Substitution (if existing)
`[]`	`[X]`	✗
`[Y]`	`[A|B]`	$\{\texttt{A} \mapsto \texttt{Y}, \texttt{B} \mapsto \texttt{[]}\}$
`[a,b,c]`	`[Y1,Y2]`	✗
`[a,b,c]`	`[Y1|Y2]`	$\{\texttt{Y1} \mapsto \texttt{a}, \texttt{Y2} \mapsto \texttt{[b,c]}\}$
`[a,b,c]`	`[Z1,Z2,Z3]`	$\{\texttt{Z1} \mapsto \texttt{a}, \texttt{Z2} \mapsto \texttt{b}, \texttt{Z3} \mapsto \texttt{c}\}$
`[a,b,c]`	`[Z1,Z2|Z3]`	$\{\texttt{Z1} \mapsto \texttt{a}, \texttt{Z2} \mapsto \texttt{b}, \texttt{Z3} \mapsto \texttt{[c]}\}$
`[[1,2],[3,4]]`	`[H|T]`	$\{\texttt{H} \mapsto \texttt{[1,2]}, \texttt{T} \mapsto \texttt{[[3,4]]}\}$
`[[1,2],3]`	`[[X|Y]|Z]`	$\{\texttt{X} \mapsto \texttt{1}, \texttt{Y} \mapsto \texttt{[2]}, \texttt{Z} \mapsto \texttt{[3]}\}$

Membership

`belongs_to(X,L)`: `X` belongs to the list `L`.

`belongs_to/2` – Implementation

- Base case: `X` is the first element of `L`

```
belongs_to(X, L) :-
    L = [X|_].
```

```
belongs_to(X, [X|_]).
```


- Recursive case: Search for `X` in the tail of `L`

```
belongs_to(X, L) :-
    L = [H|T],
    belongs_to(X, T).
```

```
belongs_to(X, [H|T]) :-
    belongs_to(X,T).
```

Membership

`belongs_to(X,L)`: `X` belongs to the list `L`.

belongs_to/2 – Examples of queries

```
?- belongs_to(3, [1,2,3,4]).
yes

?- belongs_to(2, [1,3,5]).
no

?- belongs_to(X, [2,4,6]).
X = 2 ;
X = 4 ;
X = 6 ;
no
```

```
?- belongs_to(1, L).
L = [1|_A] ;
L = [_A, 1|_B] ;
L = [_A, _B, 1|_C] ;
L = [_A, _B, _C, 1|_D] ;
yes
```

Concatenation

`concat(L1, L2, L3)`: `L3` is the list formed by all elements of `L1`, followed by all elements of `L2`.

concat/3 - Implementation

- Base case: the first list is empty

```
concat(L1, L2, L3) :-
    L1 = [],
    L3 = L2.
```

```
concat([], L2, L2).
```


- Recursive case:


```
concat(L1, L2, L3) :-
    L1 = [H1|T1],
    concat(T1, L2, T3),
    L3 = [H1|T3].
```

```
concat([H1|T1], L2, [H1|T3]) :-
    concat(T1, L2, T3).
```

Concatenation

`concat(L1, L2, L3)`: `L3` is the list formed by all elements of `L1`, followed by all elements of `L2`.

concat/3 - Examples of queries

```
?- concat([5,1,8], [4,2], [5,1,8,4,2]).
yes

?- concat([a,b], [d,e], [a,b,c,d,e]).
no

?- concat([0,2,4], [1,3,5], L).
L = [0,2,4,1,3,5] ;
no

?- concat(L1, [y,z], [y,z,x,y,z]).
L1 = [y,z,x] ;
no
```

Concatenation

`concat(L1, L2, L3)`: `L3` is the list formed by all elements of `L1`, followed by all elements of `L2`.

concat/3 - Examples of queries (cont.)

```
?- concat([1,2,4], L2, [1,2,3,4,5]).
no

?- concat(L1, L2, [a,b,c]).
L1 = [], L2 = [a,b,c] ;
L1 = [a], L2 = [b,c] ;
L1 = [a,b], L2 = [c] ;
L1 = [a,b,c], L2 = [] ;
no

?- concat(L1, [1|T2], [1,2,4,1,3,9,1,4,16]).
L1 = [], T2 = [2,4,1,3,9,1,4,16] ;
L1 = [1,2,4], T2 = [3,9,1,4,16] ;
L1 = [1,2,4,1,3,9], T2 = [4,16] ;
no
```

Partitioning

`even_odd(All, Even, Odd)`: `Even` is the sequence of even elements in `All` and `Odd` is the sequence of odd elements in `All`.

`even_odd/3` – Examples of queries

```
?- even_odd([1,2,3,4,5,6], [2,4,6], [1,3,5]).
yes
```



```
?- even_odd([3,7,1,10,3,5,8], Even, Odd).
Even = [10,8], Odd = [3,7,1,3,5] ;
no
```



```
?- even_odd(_, [4,4], [5,3,2]).
no
```

Partitioning

`even_odd(All, Even, Odd)`: `Even` is the sequence of even elements in `All` and `Odd` is the sequence of odd elements in `All`.

`even_odd/3` – Examples of queries (cont.)

```
?- even_odd(L, [8,2], [1,3,5]).
L = [8,2,1,3,5] ;
L = [8,1,2,3,5] ;
L = [8,1,3,2,5] ;
L = [8,1,3,5,2] ;
L = [1,8,2,3,5] ;
L = [1,8,3,2,5] ;
L = [1,8,3,5,2] ;
L = [1,3,8,2,5] ;
L = [1,3,8,5,2] ;
L = [1,3,5,8,2] ;
no
```

Partitioning

`even_odd(All, Even, Odd)`: `Even` is the sequence of even elements in `All` and `Odd` is the sequence of odd elements in `All`.

`even_odd/3` – Proposed solution

```
even_odd([], [], []).

even_odd([N|TAll], [N|TEven], Odd) :-
    N mod 2 =:= 0,
    even_odd(TAll, TEven, Odd).

even_odd([N|TAll], Even, [N|TOdd]) :-
    N mod 2 =:= 1,
    even_odd(TAll, Even, TOdd).
```

Try it at home!

Can you implement the following procedures?

- `len(L,N)`: `N` is the number of elements in list `L`.
- `list_double(L1,L2)`: every element in `L2` is the double of its corresponding element in `L1`.
- `list_avg(L,A)`: `A` is the average of all elements in `L`.

- `access_element(N, L, X)`: `X` is the N^{th} element in `L`.
- `remove(X,L,Rest)`: `Rest` is the list `L` from which every element equal to `X` has been removed.

- `a2b(L1,L2)`: every occurrence of `a` in `L1` occurs as `b` in `L2`, everything else is identical/unifiable.
- `permutation(L1, L2)`: `L2` is a permutation of `L1` (harder!).

Built-in predicates

Prolog has numerous built-in predicates to manipulate lists, some of them available via the 'lists' library[1]

To load the library, either use the query


```
?- use_module(library(lists)).
```

or add the rule

```
:- use_module(library(lists)).
```


to your program.

[1] documentation available at https://sicstus.sics.se/sicstus/docs/4.3.0/html/sicstus/lib_002dlists.html

Built-in predicates

A few useful built-in predicates[2]

- `member(?Element, ?List)`:
  true if `Element` occurs in `List`.
- `nonmember(?Element, ?List)`:
  true if `Element` does not occur in `List`.
- `append(?List1, ?List2, ?List3)`:
  true if `List3` is the list consisting of `List1` followed by `List2`.
- `length(?List, ?Integer)`:
  true if `Integer` is the length of `List`.

[2] How to read the predicate signature:
`+Term`: `Term` is expected to ***not*** be a variable (but may contain variables).
`-Var`: `Var` is expected to be a variable.
`?Arg`: no assumptions is made whether `Arg` is a variable or not.

Built-in predicates

A few useful built-in predicates[2]

- `rev(+List, ?Reversed)`:
  true if `List` and `Reversed` contain the same elements, but in reverse order.
- `sort(+List, -Sorted)`:
  elements from `List` are sorted in ascending order and duplicates are removed. The result is unified with `Sorted`.

- `perm(+List, -Perm)`:
  true if `List` and `Perm` are permutations of each other.

- `subseq0(+Seq, -SubSeq)`:
  true if `SubSeq` is a sub-sequence of `Seq`.
- And many others...

---

[2] How to read the predicate signature:
`+Term`: `Term` is expected to ***not*** be a variable (but may contain variables).
`-Var`: `Var` is expected to be a variable.
`?Arg`: no assumptions is made whether `Arg` is a variable or not.

What have we learned today?

- What are lists in Prolog and how they are represented
- How to design predicates to manipulate lists
- Which built-in predicates are available to use