Prolog

Wykład p.t.

Prolog

Lists in Prolog

Antoni Ligęza

ligeza@agh.edu.pl
http://galaxy.uci.agh.edu.pl/~ligeza

Wykorzystano materiały:
http://www.swi-prolog.org/

Lists: basic concepts

Lists are one of the most important structures in symbolic languages.

In most of the implementations of PROLOG lists are standard structures and there are numerous operations on them provided as built-in procedures.

Lists can be used to represent sets, sequences, and more complex structures, such as trees, records, etc.

A list in PROLOG is a structure of the form

$$[t_1, t_2, \ldots, t_n].$$

The order of elements of a list is important; the direct access is only to the first element called the *head*, while the rest forms the list called the *tail*.

Lists vs terms

Lists in fact are also terms.

A list

$$[t_1, t_2, \ldots, t_n]$$

is equivalent to a term defined as follows:

$$l(t_1, l(t_2, \ldots l(t_n, nil)\ldots)),$$

where $l/2$ is the list constructor symbol and nil is symbolic denotation of an empty list.

In practical programming it is more convenient to use the bracket notation. In order to distinguish the head and the tail of a list the following notation is used

$$[H|T].$$

If $L = [a, b, c, d]$ then after unifying L with $[H|T]$ we obtain $H = a$ (a single element), and $T = [b, c, d]$ (a list).

A list can have as many elements as necessary.

An empty list is denoted as $[\,]$.

A list can have arguments of complex structures, i.e. terms, lists, etc.

Interpretation of lists

There are three basic possibilities of interpretation of lists:

- as **sets**,
- as **sets with repeated elements**,
- as **sequences**

When thinking of lists as sets, the order of elements is unimportant.
Examples of sets:

```
[a,b,c,d,e]
```

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

```
[1,a,2,b,f(a),g(b,c)]
```

Repeated elements can occur if desired (multi-sets, bugs, collections). Examples of sets with repeated elements:

```
[a,b,c,d,e,a,c,e]
```

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

```
[1,a,2,b,f(a),g(b,c),b,1,f(a)]
```

When thinking of lists as sequences, the order of elements is important.
Examples of sequences:

```
[a,b,c,d,e]
```

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

```
[1,a,2,b,3,c,d,e,7,f(a),a,a,a]
```

Repeated elements can occur.

member/2 and select/3

Considering a list as a set one can define the following two important basic operations.

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

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

The operation of the above predicates is simple and similar; the basic idea is common.

The *member/2* predicate checks if the first argument belongs to the set represented by the second argument. This may be so if it is the first element of the list or if it is a member of the tail of the list; the definition is recursive.

The *select/3* predicate operates in a similar way, however, its primary use consists in selecting an element of a set and returning the set without the currently selected element. In fact, the *select/3* predicate can be considered as indeterministic choice — after backtracking it returns different element of the set each time.

Lists a s sets: basic operations

```
subset([],_).
subset([X|L],Set):-
      member(X,Set),
      subset(L,Set).

intersect([],_,[]).
intersect([X|L],Set,[X|Z]):-
      member(X,Set),!,
      intersect(L,Set,Z).
intersect([X|L],Set,Z):-
      not(member(X,Set)),
      intersect(L,Set,Z).

union([],Set,Set).
union([X|L],Set,Z):-
      member(X,Set),!,
      union(L,Set,Z).
union([X|L],Set,[X|Z]):-
      not(member(X,Set)),!,
      union(L,Set,Z).

difference([],_,[]).
difference([X|L],Set,[X|Z]):-
      not(member(X,Set)),!,
      difference(L,Set,Z).
difference([_|L],Set,Z):-
      difference(L,Set,Z).
```

Lists as sequences

```
sublist(S,L,Front,End,Rest) :-
     append(Front,End,L), % Rozbiór L na Front i End
     append(S,Rest,End).

append([],L,L).
append([H|T],L,[H|TL]) :- append(T,L,TL).


len([],0).
len([_|T],N) :-
     len(T,NT),
     N is NT + 1.


reverse1([],[]).
reverse1([H|T],RH) :-
     reverse1(T,R),
     append(R,[H],RH).

% Iteration vs. recurence
inverse(L,R) :- do(L,[],R).

do([],R,R).
do([H|T],A,R) :- do(T,[H|A],R).
```

List ordering

```
perm([],[]).
perm([G|Ogon],Wynik):-
     perm(Ogon,W1),
     wstaw(G,W1,Wynik).
wstaw(X,[],[X]):- !.
wstaw(X,L,[X|L]).
wstaw(X,[P|L],[P|LX]):-
     wstaw(X,L,LX).

permutation([],[]).
permutation([G|Ogon],Wynik):-
     permutation(Ogon,W1),
     insert(G,W1,Wynik).

insert(Y,XZ,XYZ):- append(X,Z,XZ),append(X,[Y|Z],XYZ).



sorted([]).
sorted([_]).
sorted([X,Y|T]) :- X =< Y, sorted([Y|T]).

nsort(L,S):-
     perm(L,S),
     sorted(S).

slowsort(L,S):-
     permutation(L,S),
     sorted(S).
```

List ordering

Sort-by-insert:

```
order([],[]).
order([H|T],R):-
     order(T,TR),
     put(H,TR,R).

put(H,[],[H]):-!.
put(H,[X|Y],[H,X|Y]):-
     H < X,!.
put(H,[X|Y],[X|Z]):-
     put(H,Y,Z),!.
```

Quick-sort in PROLOG

```
sort([],[]).
qsort([H|T],S):-
     split(H,T,L,R),
     qsort(L,LS),
     qsort(R,RS),
     append(LS,[H|RS],S).

split(_,[],[],[]).
split(H,[A|X],[A|Y],Z):-
     A =< H,
     !,
     split(H,X,Y,Z).
split(H,[A|X],Y,[A|Z]):-
     A > H,
     !,
     split(H,X,Y,Z).
```

List ordering

Bubble-sort

```
busort(L,S):-
     swap(L,LS),
     !,
     busort(LS,S).
busort(S,S).


swap([X,Y|T],[Y,X|T]):-
     X > Y.
swap([Z|T],[Z|TT]):-
     swap(T,TT).
```

Example: append-fail-sort in loop

Append-fail-sort:

```
loop:- repeat,
     write('loop'),nl,
     fail.

asort(L):-
     retractall(list(_)),
     assert(list(L)),
     go.

go:-
     repeat,
     list(L),
     write(L),nl,
     noimprove(L),
     !,
     nl,write('*****'),write(L),nl.

noimprove(L):- sorted(L).
noimprove(L):-
     append(P,[X,Y|K],L),
     X > Y,
     retract(list(L)),
     append(P,[Y,X|K],N),
     assert(list(N)),
     fail.
sorted([_]).
sorted([X,Y|T]):- X=<Y,sorted([Y|T]).
```