Lecture notes 9.1

Lists in Prolog

COMP 2411, session 1, 2004

Representation of lists

Lists are a fundamental data structure in Prolog.

Lists are built from a binary functor, denoted "." (dot), and a nullary constant, denoted "`[]`".

If "`T`" is a list and "`H`" is a structure, then "`.(H,T)`" is a new list whose first element is "`H`" and whose next elements are the members of "`T`".

We call "`H`" the head of the new list, and "`T`" the tail of the new list.

An alternative syntax for the list "`.(H,T)`" is "`[H|T]`".

Representation variants (1)

There are many syntactic variants to represent a list:

```
?- [a,b,c,d] = [a|[b,c,d]].

yes

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

yes

?- [a,b,c,d] = [a,b,[c,d]].

No
```


Representation variants (2)

```
?- [a,b,c,d] = [a,b|[c,d]].

Yes

?- [a,[b,c,d]] = [a|[[b,c,d]]].

yes

?- X = .([],[]).

X = [[]]

?- X = [[a,b]|[c,d]].

X = [[a, b], c, d]
```

Printing lists

Note how the built-ins "`write`" and "`display`" operate:

```
?- write([a,b,c]).

[a, b, c]

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

[a, b, c]

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

.(a, .(b, .(c, [])))

?- display([a|[b,c]]).

.(a, .(b, .(c, [])))
```


A few procedures for lists

Following are a few basic programs that deal with lists.

Most of them are already defined in SWI-Prolog, but it is important to understand how they work.

For all examples below, you should experiment and understand what happens (tracing the execution):

- when you change the ordering of the clauses;
- when you change which argument is a variable and which argument is an instantiated list or element;
- when you generate all possible solutions.

Being a list

"`list(L)`" succeeds iff "`L`" is a list.

Can be used to generate uninstantiated lists.

```
list([]).

list([_|T]) :- list(T).
```


Being a member of a list

"`member(X,L)`" succeeds iff "`X`" is a member of list "`L`".

Can be used to generate all members of a list, or generate all lists that contain some structure.

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

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

Being a concatenation of two lists

"`append(L1,L2,L)`" succeeds iff "`L`" is the result of appending "`L1`" to "`L2`".

Can be used to find all splittings of a list.

```
append([],L,L).

append([H|T1],L,[H|T2]) :- append(T1,L,T2).
```


Being the prefix of a list

"`prefix(P,L)`" succeeds iff "`P`" is a prefix (initial segment) of list "`L`".

```
prefix([],_).

prefix([H|P],[H|T]) :- prefix(P,T).
```


Being the suffix of a list

"`suffix(S,L)`" succeeds iff "`S`" is a suffix (final segment) of list "`L`".

```
suffix(L,L).

suffix(S,[_|T]) :- suffix(S,T).
```


Being a sublist of a list

"`sublist(S,L)`" succeeds iff "`S`" is a sublist of list "`L`".

```
sublist(S,L) :- prefix(P,L), suffix(S,P).
```

...or...

```
sublist(S,L) :- suffix(S,L), prefix(S,S).
```

...or...

```
sublist(S,L) :- prefix(S,L).

sublist(S,[_|T]) :- sublist(S,T).
```

...or other simple solutions, using append.

Being the last element of a list

"`last(E,L)`" succeeds iff "`E`" is the last element of list "`L`".

```
last(E,L) :- append(L1,[E],L).
```


Selecting an element in a list

"`select(E,L,R)`" succeeds iff "`E`" is a member of list "`L`" and "`R`" is the list of remaining elements in "`L`".

```
select(X,[X|T],T).

select(X,[Y|T],[Y|R]) :- select(X,T,R).
```