List Processing

Artificial Intelligence Programming in Prolog
Lecturer: Tim Smith
Lecture 5
07/10/04
Identifying a list

- Last lecture we introduced lists: \([a, [], \text{green(bob)}]\)
- We said that lists are recursively defined structures:

  "An empty list, \([\,]\), is a list.

  A structure of the form \([X, \ldots]\) is a list if \(X\) is a term and \(\ldots\) is a list, possibly empty."

- This can be tested using the Head and Tail notation, \([H|T]\), in a recursive rule.

  \[
  \begin{align*}
  \text{is\_a\_list}([[]]). & \quad \leftarrow \text{A term is a list if it is an empty list.} \\
  \text{is\_a\_list}([\_|T]):- & \quad \text{A term is a list if it has two} \\
  & \quad \text{elements and the second is a list.}
  \end{align*}
  \]
Base and Recursive Cases

- A recursive definition, whether in prolog or some other language (including English!) needs two things.
  - A definition of when the recursion terminates. Without this the recursion would never stop!
    - This is called the base case: `is_a_list([]).
    - Almost always comes before recursive clause

- A definition of how we can define the problem in terms of a similar, smaller problem.
  - This is called the recursive case: `is_a_list([_|T]):- is_a_list(T).

- There might be more than one base or recursive case.
Focussed Recursion

• To ensure that the predicate terminates, the recursive case must move the problem closer to a solution.
  – If it doesn’t it will loop infinitely.

• With list processing this means stripping away the Head of a list and recursing on the Tail.

    is_a_list([_|T]):-
    is_a_list(T).

• The same focussing has to occur when recursing to find a property or fact.

    is_older(Ancestor,Person):-
    is_older(Someone,Person),
    is_older(Ancestor,Someone).

Head is replaced with an underscore as we don’t want to use it.
Focussed Recursion (2)

Given this program:

parent(tom,jim).
parent(mary,tom).

is_older(Old,Young):-
    parent(Old,Young).

is_older(Ancestor,Young):-
    is_older(Someone,Young),
    is_older(Ancestor,Someone).

A query looking for all solutions will loop.

?-is_older(X,Y).
X = tom,
Y = jim ? ;
X = mary,
Y = tom ? ;
X = mary,
Y = jim ? ;
*loop*

It loops because the recursive clause does not focus the search it just splits it. If the recursive is_older/2 doesn’t find a parent it just keeps recursing on itself.
Focussed Recursion (3)

The correct program:

\begin{verbatim}
parent(tom,jim).
parent(mary,tom).

is_older(Old,Young):-
    parent(Old,Young).

is_older(Ancestor,Young):-
    parent(Someone,Young),
    is_older(Ancestor,Someone).
\end{verbatim}

• Can generate all valid matches without looping.

| ?- is_older(X,Y).
| X = tom,  Y = jim ? ;
| X = mary,  Y = tom ? ;
| X = mary,  Y = jim ? ;
| no

To make the problem space smaller we need to check that \texttt{Young} has a parent before recursion. This way we are not looking for something that isn’t there.
List Processing Predicates: Member/2

- **Member/2** is possibly the most used user-defined predicate (i.e. you have to define it every time you want to use it!)
- It checks to see if a term is an element of a list.
  - it returns `yes` if it is
  - and `fails` if it isn’t.

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

- It 1st checks if the Head of the list unifies with the first argument.
  - If yes then succeed.
  - If no then fail first clause.

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

- The 2nd clause ignores the head of the list (which we know doesn’t match) and recurses on the Tail.
|?- member(ringo, [john, paul, ringo, george]).

Fail(1): member(ringo, [john|_]).
  (2): member(ringo, [_|paul, ringo, george]):-
    Call: member(ringo, [paul, ringo, george]).
      Fail(1): member(ringo, [paul|_]).
        (2): member(ringo, [<_|ringo, george]):-
          Call: member(ringo, [ringo, george]).
            Succeed(1): member(ringo, [ringo|_])).

1) member(H, [H|_]).
2) member(H, [_|T]):-
   member(H, T).

List Processing Predicates: Member/2
Quick Aside: Tracing Prolog

- To make Prolog show you its execution of a goal type \texttt{trace}. at the command line.
  - Prolog will show you:
    - which goal is \texttt{Called} with which arguments,
    - whether the goal succeeds (\texttt{Exit}),
    - has to be \texttt{Redone}, or \texttt{Fails}.
  - The tracer also indicates the level in the search tree from which a goal is being called.
    - The number next to the goal indicates the level in the tree (top level being 0).
    - The leftmost number is the number assigned to the goal (every new goal is given a new number).
- To turn off the tracer type \texttt{notrace}. 

Tracing Member/2

?- trace.
?- member(ringo, [john, paul, ringo, george]).
   1 1 Call: member(ringo, [john, paul, ringo, george]) ?
   2 2 Call: member(ringo, [paul, ringo, george]) ?
   3 3 Call: member(ringo, [ringo, george]) ?
   3 3 Exit: member(ringo, [ringo, george]) ?
   2 2 Exit: member(ringo, [paul, ringo, george]) ?
   1 1 Exit: member(ringo, [john, paul, ringo, george]) ?
yes

?- member(stuart, [john, paul, ringo, george]).
   1 1 Call: member(ringo, [john, paul, ringo, george]) ?
   2 2 Call: member(ringo, [paul, ringo, george]) ?
   3 3 Call: member(ringo, [ringo, george]) ?
   4 4 Call: member(stuart, [george]) ?
   5 5 Call: member(stuart, []) ? ← [] does not match [H|T]
   5 5 Fail: member(stuart, []) ?
   4 4 Fail: member(stuart, [george]) ?
   3 3 Fail: member(ringo, [ringo, george]) ?
   2 2 Fail: member(ringo, [paul, ringo, george]) ?
   1 1 Fail: member(ringo, [john, paul, ringo, george]) ?
no
Collecting Results

• When processing data in Prolog there are three ways to collect the results:

1. Compute result at base case first, then use this result as you backtrack through the program.
2. Accumulate a result as you recurse into the program and finalise it at the base case.
3. Recurse on an uninstantiated variable and accumulate results on backtracking.

• These all have different uses, effect the order of the accumulated data differently, and require different degrees of processing.
Compute lower result first.

- We want to define a predicate, `length/2`, which takes a list as its first argument and returns a number as the second argument that is equal to the length of the list.
- We can use recursion to move through the list element by element and `is/2` to count as it goes.

```
listlength([_|T],N1):-
    listlength(T,N),
    N1 is N+1.
```

- To make a counter we need to initialise it at a value i.e. zero.
- As the counter increases during backtracking it needs to be initialised in the base case.

```
listlength([],0).
```
Compute lower result first: trace.

```prolog
listlength([],0).
listlength([_|T],N1):-
    listlength(T,N),
    N1 is N+1.
```

?- listlength([a,b,c],N).
1 1 Call: listlength([a,b,c],_489) ?
2 2 Call: listlength([b,c],_1079) ?
3 3 Call: listlength([c],_1668) ?
4 4 Call: listlength([],_2257) ?
4 4 Exit: listlength([],0) ?
5 4 Call: _1668 is 0+1 ?
5 4 Exit: 1 is 0+1 ?
3 3 Exit: listlength([c],1) ?
6 3 Call: _1079 is 1+1 ?
6 3 Exit: 2 is 1+1 ?
2 2 Exit: listlength([b,c],2) ?
7 2 Call: _489 is 2+1 ?
7 2 Exit: 3 is 2+1 ?
1 1 Exit: listlength([a,b,c],3) ?
N = 3 ? yes
Why compute lower result first?

```prolog
listlength([],0).
listlength([_|T],N1):-
    N1 is N+1,
    listlength(T,N).
```

```prolog
listlength([],_).
listlength([_|T],N):-
    N1 is N+1,
    listlength(T,N1).
```

|?- listlength([a,b,c],N). |?- listlength([a,b,c],0).
Instantiation error in is/2 fail
1 Call: listlength([a,b,c],0) ?
2   2 Call: _1055 is 0+1 ?
2   2 Exit: 1 is 0+1 ?
3   2 Call: listlength([b,c],1) ?
4   3 Call: _2759 is 1+1 ?
4   3 Exit: 2 is 1+1 ?
5   3 Call: listlength([c],2) ?
6   4 Call: _4463 is 2+1 ?
6   4 Exit: 3 is 2+1 ?
7   4 Call: listlength([],3) ?
7   4 Exit: listlength([],3) ?
5   3 Exit: listlength([c],2) ?
3   2 Exit: listlength([b,c],1) ?
1   1 Exit: listlength([a,b,c],0) ?
yes
Using an Accumulator

- You can also accumulate results as you recurse into the program, finalising the result at the base.
- Once the result is finalised we need someway of getting it back out of the program.

```prolog
listlength([],Acc,Acc).  % Finalise result.

listlength([_|T],Acc,Out):-
    Acc1 is Acc+1,
    listlength(T,Acc1,Out).  % Instantiate result to Output variable in base case and pass back.
```

Increase Accumulator as we recurse
Using an Accumulator (2)

listlength([a,b,c],0,N).
1  1 Call: listlength([a,b,c],0,_501) ?
2  2 Call: _1096 is 0+1 ?
2  2 Exit: 1 is 0+1 ?
3  2 Call: listlength([b,c],1,_501) ?
4  3 Call: _2817 is 1+1 ?
4  3 Exit: 2 is 1+1 ?
5  3 Call: listlength([c],2,_501) ?
6  4 Call: _4538 is 2+1 ?
6  4 Exit: 3 is 2+1 ?
7  4 Call: listlength([],3,_501) ?
7  4 Exit: listlength([],3,3) ?
5  3 Exit: listlength([c],2,3) ?
3  2 Exit: listlength([b,c],1,3) ?
1  1 Exit: listlength([a,b,c],0,3) ?

N = 3 ?
yes

call/1 :-
    listlength([],A,A).
    listlength([_|T],A,O):-
        A1 is A+1,
        listlength(T,A1,O).
Using an auxiliary predicate

- When using an accumulator it needs to be initialised at the right value (e.g. [] or 0).
- Make the predicate with the accumulator an auxiliary to the predicate that the user will call.

```prolog
listlength(List,Length):-
    listlength2(List,0,Length).
```

```prolog
listlength2([],A,A).
listlength2([_|T],A,O):-
    A1 is A+1,
    listlength2(T,A1,O).
```

- This ensures that the accumulator is initialise correctly and that the user doesn’t have to understand the workings of your code to use it.
Combining lists

- A common use of an accumulator is to construct lists.
- If we want to make a new list out of the combined elements of two lists we can’t just make one list the Head of a new list and the other the tail as:

  ```prolog
  | ?- L1=[a,b], L2=[c,d], Z=[L1|L2].
  L1 = [a,b], L2 = [c,d], Z = [[a,b],c,d] ?
  ```

- We need to take each element from L1 and add them to L2 one at a time.

- There are two ways we can do this
  - during recursion, or
  - backtracking.
Constructing a list during Recursion

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

To add L1 to L2 during recursion we can use the bar notation to decompose L1 and add the Head to L2.

\[\text{pred([H|T],L2,Out):=}\]
\[\text{pred(T,[H|L2],Out).}\]

We need to have an extra variable (Out) which can be used to pass back the new list once L1 is empty.

\[\text{pred([],L2,L2).} \leftarrow \text{base case: when L1 is empty make the new list equal to the Output list.}\]

The base case must go before the recursive case.
Constructing a list during Recursion (2)

?- pred([a,b],[c,d],Out).
1  1 Call: pred([a,b],[c,d],_515) ?
2  2 Call: pred([b],[a,c,d],_515) ?
3  3 Call: pred([], [b,a,c,d],_515) ?
3  3 Exit: pred([], [b,a,c,d], [b,a,c,d]) ?
2  2 Exit: pred([b], [a,c,d], [b,a,c,d]) ?
1  1 Exit: pred([a,b], [c,d], [b,a,c,d]) ?
Out = [b,a,c,d] ?
yes

pred([],L2,L2).
pred([H|T],L2,Out):-
  pred(T, [H|L2], Out).

If you construct a list through recursion (on the way down) and then pass the answer back the elements will be in reverse order.
reverse/3

\[
\begin{align*}
&\text{?- pred([a,b],[c,d],\text{Out}).} \\
&1\quad 1 \text{ Call: } \text{pred([a,b],[c,d],_515) ?} \\
&2\quad 2 \text{ Call: } \text{pred([b],[a,c,d],_515) ?} \\
&3\quad 3 \text{ Call: } \text{pred([], [b,a,c,d],_515) ?} \\
&3\quad 3 \text{ Exit: } \text{pred([], [b,a,c,d], [b,a,c,d]) ?} \\
&2\quad 2 \text{ Exit: } \text{pred([b], [a,c,d], [b,a,c,d]) ?} \\
&1\quad 1 \text{ Exit: } \text{pred([a,b], [c,d], [b,a,c,d]) ?} \\
\text{Out} = [\text{b,a,c,d}] ? \\
\text{yes}
\end{align*}
\]

\begin{verbatim}
reverse([],L2,L2).
reverse([H|T],L2,Out):-
    reverse(T,[H|L2],Out).
\end{verbatim}

If you construct a list through recursion (on the way down) and then pass the answer back the elements will be in reverse order.
To maintain the order of list elements we need to construct the list on the way out of the program, i.e. through backtracking.

Use the same bar deconstruction as before but add the head element of L1 to Out in the Head of the clause.

\[
\text{pred}([H|T], L2, [H|Out]):- \quad \leftarrow \quad \text{Head is not added until backtracking.}
\]

\[
\text{pred}(T, L2, Out).
\]

Now when we reach the base case we make L2 the foundation for the new Out list and add our L1 elements to it during backtracking.

\[
\text{pred}([], L2, L2). \quad \leftarrow \quad \text{base case: when L1 is empty make the new list equal to the Output list.}
\]
append/3

| ?- pred2([a,b],[c,d],Out).
1 1 Call: pred2([a,b],[c,d],_515) ?
2 2 Call: pred2([b],[c,d],_1131) ?
3 3 Call: pred2([], [c,d],_1702) ?
3 3 Exit: pred2([], [c,d],[c,d]) ?
2 2 Exit: pred2([b],[c,d],[b,c,d]) ?
1 1 Exit: pred2([a,b],[c,d],[a,b,c,d]) ?
Out = [a,b,c,d] ?

yes

append([],L2,L2).
append([H|T],L2,[H|Rest]):-
    append(T,L2,Rest).

* append/3 is another very common user-defined list processing predicate.
Computing in reverse

- Both `reverse/3` and `append/3` can be used backwards to make two lists out of one.
- This can be a useful way to strip lists apart and check their contents.

```prolog
?- append(X, Y, [a, b, c, d]).
X = [], Y = [a, b, c, d] ;
X = [a], Y = [b, c, d] ;
X = [a, b], Y = [c, d] ;
X = [a, b, c], Y = [d] ;
X = [a, b, c, d], Y = [] ;
no

?- append(X, [c, d], [a, b, c, d]).
X = [a, b] ;
no
```

```prolog
append([], L2, L2).
append([H|T], L2, [H|Rest]) :-
    append(T, L2, Rest).
```
Computing in reverse

- Both `reverse/3` and `append/3` can be used backwards to make two lists out of one.
- This can be a useful way to strip lists apart and check their contents.

```prolog
| ?- reverse(X,Y,[a,b,c,d]).
X = [], Y = [a,b,c,d] ? ;
X = [a], Y = [b,c,d] ? ;  reverse([],L2,L2).
X = [b,a], Y = [c,d] ? ;  reverse([H|T],L2,Out):- reverse(T,[H|L2],Out).
X = [c,b,a], Y = [d] ? ;
X = [d,c,b,a], Y = [] ? ;
*loop*

| ?- reverse([d,c,b,a],Y,[a,b,c,d]).
Y = [] ?
yes
```
Summary

- Base and recursive cases
- Using focused recursion to stop infinite loops.
- List processing through recursion: member/2
- Introduced the Prolog tracer.
- Showed three techniques for collecting results:
  - Recursively find a result, then revise it at each level.
    - listlength/3
  - Use an accumulator to build up result during recursion.
    - reverse/3
  - Build result in the head of the clause during backtracking.
    - append/3