PROLOG

Combining Lists & Built-in Predicates

Artificial Intelligence Programming in Prolog

Lecturer: Tim Smith

Lecture 6

11/10/04

Collecting Results

- Last time we discussed three methods of collecting 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.

- Today we will look how you can use an accumulator during recursion and backtracking to combine lists.

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

Constructing a list during Recursion

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

Desired behaviour

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

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

Accumulator

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

`pred([],L2,L2).` ← 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)

Always the same variable

```
| ?- 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

```
| ?- 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
```

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

Constructing a list in backtracking

- 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.

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

← Head is not added until backtracking.

- 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.

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

← base case: when L1 is empty make the new list equal to the Output list.

append/3

Variable changes at every Call.

```
| ?- 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.

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

```
|?-append(X,[c,d],[a,b,c,d]).
  X = [a,b] ?
  yes
```

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.

```
| ?- reverse(X,Y,[a,b,c,d]).
  X = [], Y = [a,b,c,d] ? ;
  X = [a], Y = [b,c,d] ? ;
  X = [b,a], Y = [c,d] ? ;
  X = [c,b,a], Y = [d] ? ;
  X = [d,c,b,a], Y = [] ? ;
  *loop*
```

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

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

Built-in Predicates

```
var/1
nonvar/1
atom/1
atomic/1
number/1
integer/1
float/1
compound/1
ground/1
```

Identifying terms

```
=../2
functor/3
arg/3
```

Decomposing structures

```
findall/3
setof/3
bagof/3
```

Collecting all solutions

Identifying Terms

- These built-in predicates allow the type of terms to be tested.

`var(X)`	succeeds if X is currently an uninstantiated variable.
`nonvar(X)`	succeeds if X is not a variable, or already instantiated
`atom(X)`	is true if X currently stands for an atom
`number(X)`	is true if X currently stands for a number
`integer(X)`	is true if X currently stands for an integer
`float(X)`	is true if X currently stands for a real number.
`atomic(X)`	is true if X currently stands for a number or an atom.
`compound(X)`	is true if X currently stands for a structure.
`ground(X)`	succeeds if X does not contain any uninstantiated variables.

var/1, nonvar/1, atom/1

var(X) succeeds if X is currently an uninstantiated variable.

```
| ?- var(X).
true ?
yes
```

```
| ?- X = 5, var(X).
no
```

```
| ?- var([X]).
no
```

nonvar(X) Conversely, `nonvar/1` succeeds if X is not a variable, or already instantiated.

```
| ?- nonvar(X).
no
```

```
| ?- X = 5,nonvar(X).
X = 5 ?
yes
```

```
| ?-nonvar([X]).
true ?
yes
```

atom(X) is true if X currently stands for an atom: a non-variable term with 0 arguments, and not a number

```
| ?- atom(paul).
yes
```

```
| ?- X = paul,atom(X).
X = paul ?
```

```
| ?- atom([]).
yes
| ?-atom([a,b]).
no
```

number/1, integer/1, float/1

number(X) is true if X currently stands for any number

```
| ?- number(X).
no
```

```
| ?- X=5,number(X).
X = 5 ?
yes
```

```
| ?- number(5.46).
yes
```

To identify what type of number it is use:

integer(X) is true if X currently stands for an integer (a whole positive or negative number or zero).

float(X) is true if X currently stands for a real number.

```
| ?- integer(5).
yes
```

```
| ?- integer(5.46).
no
```

```
| ?- float(5).
no
```

```
| ?- float(5.46).
yes
```

atomic/1, compound/1, ground/1

- If `atom/1` is too specific then you can use **`atomic/1`** which accepts numbers and atoms.

```
| ?- atom(5).        | ?- atomic(5).
no                   yes
```

- If `atomic/1` fails then the term is either an uninstantiated variable (which you can test with `var/1`) or a **`compound`** term:

```
|?-compound([]).  |?-compound([a]).  |?-compound(b(a)).
no                yes                yes
```

- **`ground(X)`** succeeds if X does not contain any uninstantiated variables. Also checks inside compound terms.

```
|?-ground(X).             |?-ground(a(b,X)).
no                        no
|?-ground(a).             |?-ground([a,b,c]).
yes                       yes
```

Decomposing Structures

- When using compound structures you can't use a variable to check or make a functor.

```
|?- X=tree, Y = X(maple).
Syntax error Y=X<<here>>(maple)
```

functor(T,F,N) is true if F is the principal functor of T and N is the arity of F.

arg(N,Term,A) is true if A is the Nth argument in Term.

```
|?-functor(t(f(X),a,T),Func,N).
N = 3, Func = t ?
yes
```

```
|?-arg(2,t(t(X),[]),A).
A = [] ?
yes
```

```
| ?- functor(D,date,3), arg(1,D,11), arg(2,D,oct),
   arg(3,D,2004).
D = date(11,oct,2004) ? yes
```

Decomposing Structures (2)

- We can also decompose a structure into a list of its components using `=../2`.

`Term =.. L` is true if L is a list that contains the principal functor of Term, followed by its arguments.

```
| ?- f(a,b) =.. L.
L = [f,a,b] ?
yes
```

```
|?- T =.. [is_blue,sam,today].
T = is_blue(sam,today) ?
yes
```

- By representing the components of a structure as a list they can be recursively processed without knowing the functor name.

```
| ?- f(2,3)=..[F,N|Y], N1 is N*3, L=..[F,N1|Y].
L = f(6,3)?
yes
```

Collecting all solutions

- You've seen how to generate all of the solutions to a given goal, at the SICStus prompt (;):

```
| ?- member(X, [1,2,3,4]).
   X = 1 ? ;
   X = 2 ? ;
   X = 3 ? ;
   X = 4 ? ;
   no
```

- It would be nice if we could generate all of the solutions to some goal within a program.
- There are three similar built-in predicates for doing this:

findall/3

setof/3

bagof/3

Meta-predicates

- `findall/3`, `setof/3`, and `bagof/3` are all ***meta-predicates***
  - they manipulate Prolog's proof strategy.

```
findall(X,P,L)
setof(X,P,L)
bagof(X,P,L)
```

All produce a list L of all the objects X such that goal P is satisfied (e.g. `age(X,Age)`).

- They all repeatedly call the goal P, instantiating the variable X within P and adding it to the list L.
- They succeed when there are no more solutions.
- Exactly simulate the repeated use of ';' at the SICStus prompt to find all of the solutions.

findall/3

- **`findall/3`** is the most straightforward of the three, and the most commonly used:

```
| ?- findall(X, member(X, [1,2,3,4]), Results).
     Results = [1,2,3,4]
     yes
```

- This reads: `find all of the Xs, such that X is a member of the list `[1,2,3,4]` and put the list of results in `Results`'.
- Solutions are listed in the result in the same order in which Prolog finds them.
- If there are duplicated solutions, all are included. If there are infinitely-many solutions, it will never terminate!

findall/3 (2)

- We can use `findall/3` in more sophisticated ways.
- The second argument, which is the goal, might be a compound goal:

```
| ?- findall(X, (member(X, [1,2,3,4]), X > 2), Results).
     Results = [3,4]?
     yes
```

- The first argument can be a term of any complexity:

```
|?- findall(X/Y, (member(X,[1,2,3,4]), Y is X * X),
    Results).
    Results = [1/1, 2/4, 3/9, 4/16]?
    yes
```

setof/3

- **`setof/3`** works very much like `findall/3`, except that:
  - It produces the ***set*** of all results, with any duplicates removed, and the results ***sorted***.
  - If any variables are used in the goal, which do not appear in the first argument, `setof/3` will return a separate result for each possible instantiation of that variable:

```
|?-setof(Child, age(Child,Age),Results).
    Age = 5,
    Results = [ann,tom] ? ;
    Age = 7,
    Results = [peter] ? ;
    Age = 8,
    Results = [pat] ? ;
    no
```

```
age(peter, 7).
age(ann, 5).
age(pat, 8).
age(tom, 5).
age(ann, 5).
```

Knowledge base

setof/3 (2)

- We can use a ***nested*** call to `setof/3` to collect together the individual results:

```
| ?- setof(Age/Children, setof(Child,age(Child,Age),
     Children), AllResults).
     AllResults = [5/[ann,tom],7/[peter],8/[pat]] ?
     yes
```

- If we don't care about a variable that doesn't appear in the first argument, we use the following form:

```
| ?- setof(Child, Age^age(Child,Age), Results).
     Results = [ann,pat,peter,tom] ? ;
     no
```

- This reads: `Find the set of all children, such that the `Child` has an `Age` (whatever it might be), and put the results in `Results`.'

bagof/3

- **bagof/3** is very much like `setof/3` except:
  - that the list of results might contain duplicates,
  - and isn't sorted.

```
| ?- bagof(Child, age(Child,Age),Results).
     Age = 5, Results = [tom,ann,ann] ? ;
     Age = 7, Results = [peter] ? ;
     Age = 8, Results = [pat] ? ;
     no
```

- `bagof/3` is different to `findall/3` as it will generate separate results for all the variables in the goal that do not appear in the first argument.

```
| ?- findall(Child, age(Child,Age),Results).
     Results = [peter,pat,tom,ann,ann] ? ;
     no
```

PROLOG

Summary

- Showed two techniques for combining lists:
  - Use an accumulator to build up result during recursion: reverse/3
  - Build result in the head of the clause during backtracking: append/3
- Built-in predicates
  - Identifying Terms
    - **`var/1, nonvar/1, atom/1, atomic/1,`**
    - **`number/1, integer/1, float/1,`**
    - **`compound/1, ground/1`**
  - Decomposing Structures
    - **`=../2, functor/3, arg/3`**
  - Collecting all solutions
    - **`findall/3, setof/3, bagof/3`**