CS206 Lecture 10

Prolog- Lists and Recursion

G. Sivakumar
Computer Science Department
IIT Bombay
siva@iitb.ac.in
http://www.cse.iitb.ac.in/∼siva

Tue, Jan 21, 2003

Plan for Lecture 10

- Lists in Prolog
- Recursive Definitions

Representing Structured Data

In the simple database-like example, constants like

```
jaya, kerala, ganga, hindi, ...
```

sufficed to denote data-objects.

For numbers, we used

```
0, s(X)
```

a unary function symbol "s" for successor.

Lists are built using a binary operator and are very useful in representing structured data.

Lists in Prolog

Examples:

```
[]        Empty List
[a]       Singleton List
[a,b]     Two element List
[a,b,a]   Three element List
```

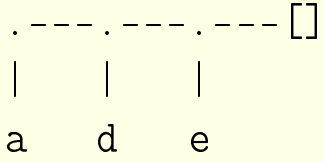
That is, Lists are represented in Prolog simply by sqaure brackets with elements of the list separated by commas.

Note: Lists can be nested: `[a, [a,b], c]` has 3 elements.

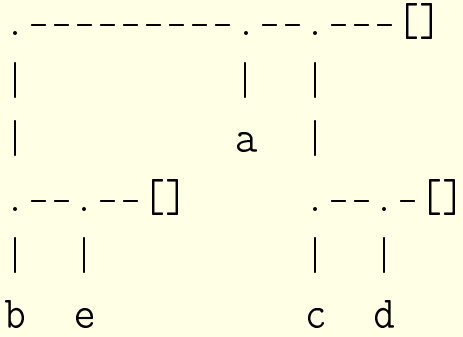
Diagrams for Lists

Similar to **tree** notation for terms.

Example: `[a,d,e]`



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



How to Add elements to Lists

Elements can be added only to the front (Head) of a list.

Prolog uses the symbol "|" with the meaning:

```
[ New-First | Old-List ]
```

Examples:

```
- If L = [b,c,d] then
   [a | L]  =  [a,b,c,d]
- If L = [b,c,d] then
   [[a] | L]  = [[a], b, c, d]
 - If L = [[b],c,d] then
   [a | L] = [a, [b], c, d]
```

Exercise: Draw diagrams for these.

More about List Constructor

Really "|" is like a binary function symbol. Instead of writing

```
|(arg1, arg2)
```

as Prolog syntax insists, we use for convenience

```
[ arg1 | arg2 ]
```

"|" need not be used only when for adding elements to lists.
It is very useful in describing **patterns** when stating facts and rules in a Prolog Program.

Double-Headers!

Consider lists like

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

How can we define a predicate `dbh(X)` in a Prolog Program to mean that: "`X` is a list with the first two elements same"

- Cannot write down (infinitely) many facts one for each such list.
- Can be done with a single fact in the Program.

Double-Header (Answer)

Let the Program have only 1 fact

```
dbh([ X | [X | Rest]]).
```

What happens when we ask the query:

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

The query can be unified with the fact in the program by changing

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

Exercise: Try out the diagrams.

What about a query like:

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

There is no way to unify this with the fact in the program.

So Prolog answers **no**.

What about the following queries?

```
dbh([]).
dbh([a]).
dbh([[c],[c], d, e]).
```

Identifying Lists that are Not Doubletons

Suppose we wish to write a predicate

```
ndb(X)
```

that checks that "X is not a list with exactly two elements."
Can you explain, why the following program works?

```
ndb([]).
ndb([X]).
ndb([X | [Y | [Z | Rest]]]).
```

Why can we not write the third fact as:

```
ndb([X | [ Y | Rest ]]).
```

Lists of Same Length

How can we write a predicate to check if two lists have same number of elements.

That is, we want the following facts to be true.

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

One solution:

```
slength([],[]).
slenght([X | Restof1], [Y | Restof2]) :-
     slength(Restof1, Restof2).
```

Question: How will you read second rule in English?

Structural Recursion

It is very important that the previous example (`slength`) is understood thoroughly by the student.

This is a typical pattern of how Prolog programs are written:

- Base case
- Inductive case

and the structural recursion (on the data-type) needs explaining.

Why are these two rules enough? Will it answer correctly for all queries now? Tracing execution on a few sample goals will be very illustrative now.

Students should also be constantly reminded of how they would solve such problems in *C* or *Pascal*.

Execution trace

```
slength([a,b,c], [d,e,f])
   | Rule can be used with
   | X = a,  Restof1 = [b,c]
   | Y = d,  Restof2 = [e,f]
slength([b,c], [e,f])
   | * New copy of rule is used *
   | Rule can be used with
   | X = b,  Restof1 = [c]
   | Y = e,  Restof2 = [f]
slength([c], [f])
   | * New copy of rule is used *
   | Rule can be used with
   | X = c,  Restof1 = []
   | Y = f,  Restof2 = []
slength([],[])
   *** Given as fact ***
```

Another Execution trace

```
slength([a,b,c], [d,e,f,g])
  |  Rule can be used  with
  |  X = a,  Restof1 = [b,c]
  |  Y = d,  Restof2 = [e,f,g]
slength([b,c], [e,f,g])
  | * New copy of rule is used *
  | Rule can be used with
  | X = b,  Restof1 = [c]
  | Y = e,  Restof2 = [f,g]
slength([c], [f,g])
  | * New copy of rule is used *
  | Rule can be used with
  | X = c,  Restof1 = []
  | Y = f,  Restof2 = [g]
slength([],[g])
  * No rule applies! Fail. *
```

Length of a list

Let the predicate `len(List, N)` mean that "There are `N` elements in List." That is the following facts are true.

```
len([a,b,c], s(s(s(0)))).
len([e],s(0)).
```

How will you code this as a:

- Prolog program
- C or Pascal program

Solution to Length of List

```
/* base case */
len([],0).

/* inductive case */
len([X|Rest], s(N)) :-
     len(Rest, N).
```

The inductive rule can be read in English as:

> "If the length of Rest is `N`, then the length adding `X` to Rest is the successor of `N`."

We can actually compute with this.
The query `len([a,b],Ans)` should give:
`Ans = s(s(0))` as an answer.

Trace of Length Query execution

```
len([a,b], Ans).
  |  Rule applies making
  |  X1 = a, Rest1 = [b]
  |  Ans = s(N1)
len([b],N1).
  |  * New copy of rule.
  |    Variable names changed. *
  |  Rule applies  with
  |  X2 = b, Rest = []
  |  N1 = s(N2)
len([],N2).
  | Given fact makes N2 = 0
Goal solved.
  Composing the subsitutions
  for variables we get
  Ans = s(N1) = s(s(N2)) = s(s(0))
```

Food for Thought

Now that we have written a way to compute the length of a list, compare this solution to check if two list have the same length with the previous one.

```
len([],0).
len([X|Rest], s(N)):-
   len(Rest, N).

slength(List1, List2):-
   len(List1,N),
   len(List2,N).
```

Which do you prefer?

More Problems to Try

`member(X, List)` means "`X` is present in List"

`sublist(L1, L2)` means "`L1` is a sublist of `L2`"

`flatform(L1, L2)` means "`L2` is a flat version of `L1`"

Examples:

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