Processing lists in Prolog

This lecture begins by showing the iteration can be expressed as recursion. Lists are introduced as a versatile data structure then the essentials of list processing in Prolog are demonstrated.
Previously...

Two programs:
- outputting the nodes of binary tree (stored as a recursive object);
- route finding round a very simple network.

A standard pattern:
- terminating (base) clause
- recursive (continuing) clause
Why use recursion?

Allows variable number of operations:
– you don’t have to know in advance how many operations you have to perform before the program stops.

Allows processing of data structures of arbitrary depth or length:
– e.g. a **binary tree** of any depth from 1 ... ∞
– e.g. a **list** of any length from 1 ... ∞
Prolog doesn’t have iteration

But all iteration can be rewritten using recursion…

For instance, the *for loop*

The terminating clause:

```prolog
for_loop(Counter, End, _Step) :-
  Counter > End,
  write_message(message('Counter is: ',
                        Counter)).
```
Prolog doesn’t have iteration

But all iteration can be rewritten using recursion…

For instance, the *for loop*

The recursive clause:

```prolog
for_loop(Counter, End, Step) :-
    Counter =< End,
    write_message(message('Counter is: ', Counter)),
    Counter1 is Counter + Step,
    for_loop(Counter1, End, Step).
```

*Demo 1*
Lists: the versatile data structure

Lists are so useful that whole programming languages are written around them:

– LISP – the first functional language.

Lists allow us to write programs about unpredictable situations:

– a differing number of “things” for a program to begin with;
– a differing number of intermediate “things” to store while the program is running.
Variable number of outputs

If the problem is to get a list of all the paths from Start to Finish:

<table>
<thead>
<tr>
<th>Start</th>
<th>Finish</th>
<th>Routes</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
Lists in Prolog

List processing is (fundamentally) the same in all languages.

There are superficial differences of notation between languages.

Lists in Prolog use: ‘[‘, ‘]’, ‘,’.  
e.g.:  
  [a, 2, c(3, d)].
Some lists in Prolog

List of mixed terms:
\[ [a, 2, c(3, d), \text{Variable}] \]

List of one kind of term:
\[ [\text{atom1}, \text{atom2}, \text{atom3}, \text{atom4}] \]

List of lists:
\[ [[a], [b, c, [d, [e], f]], [g]] \]

Empty list:
\[ [] \]
Consolidation moment

Lists are used to store data/information that **varies in quantity**:

1. that varies from run to run of the program;
2. that varies whilst the program is running.

Prolog’s lists:
- start with ‘[’
- end with ‘]’
- separate elements by ‘,’
Unifying lists - 1

It is essential to understand how to unify lists:

A list will unify with a variable:

\[ [a_1, b_2, c_3] = \text{Var1}. \]
\[ \text{Var2} = []. \]

Contents of a list can be accessed through the head:

\[ | \text{?- \,[Head}\mid\text{Tail}] = [a_1, b_2, c_3]. \]
\[ \text{Head} = a_1, \]
\[ \text{Tail} = [b_2, c_3] ? \]

Note: the tail is always a list
Two special cases:

1. **Accessing more than one element at a time:**
   
   \[ [a_1, b_2, c_3] = [\text{Hd}_1, \text{Hd}_2 | \text{Tail}] \].

2. **What happens when there is nothing in the tail?**

   \[ | ?- [\text{Head}|\text{Tail}] = [a_1]. \]
   
   Head = \text{a}_1,
   
   Tail = [] ?

   **Note:** the tail is *always* a list
Principles of list processing - 1

When we start running a program, we know very little about any list we’re using:

*either:*

- a list is empty

*or:*

- the list has at least one element

  (but we don’t know how many elements because lists vary in length).
Principles of list processing - 2

This converts into a familiar program pattern:

either:

terminate because the list is empty

or:

do something with the first element and recurse.
Consolidation moment

Elements in a list can only be accessed from the front of the list.

It is possible to extract more than one head element at a time.

List processing is a recursive because either:

– you want to process the head of the list

or

– you want to process something in the tail.
First list processing example - 1

Count the elements of a list:

*terminate because the list is empty:*
\[
\text{count\_elem}([], \text{Total}, \text{Total}).
\]

*count the first element and recurse:*
\[
\text{count\_elem}([\text{Hd}|\text{Tail}], \text{Sum}, \text{Total}) :- \\
\text{Sum1} \text{ is } \text{Sum} + 1, \\
\text{count\_elem}(\text{Tail}, \text{Sum1}, \text{Total}).
\]
First list processing example - 2

Finishing the code:

Add an interface so the user doesn’t have to remember a literal:

```prolog
count_elem(List, Total) :-
    count_elem(List, 0, Total).
```

This is the accumulator pair
Second list processing example - 1

Description of the problem:

*Input* – a list of atomics

*Output* – a list of numbers and a list of atoms in the order of the “input list”

Example:

classify([atom, 1, 3.5], Numbers, Atoms).

Numbers = [1, 3.5]
Atoms = [atom]
Second list processing example - 2

Design thoughts:

– A **recursive problem** – because it uses lists of varying length.
– Needs a **terminating clause**.
– All elements in the list need to be scanned – so terminate when we’ve got to the end of the list.
– We can classify the **head** of the list and then classify the contents of the **tail**.
– The head is a number **OR** an atom – so there will be two recursive rules.
Second list processing example - 3

First program the terminating clause

When the “input” list is empty, then the “output” lists are also empty.

classify([], [], []).

Now test it!

Demo3
Second list processing example - 4

When the terminating clause is OK, develop the recursive clauses:

When the “input” head is number, add it to the list of numbers.
Second list processing example - 5

A Java programmer’s version of the recursive clause:

classify(Input, Numbers_Out, Atoms_Out) :-
    Input = [Head|Tail],
    number(Head),
    classify(Tail, Numbers_in_Tail, Atoms_in_Tail),
    Numbers_Out = [Head|Numbers_in_Tail],
    Atoms_Out = Atoms_in_Tail.
Second list processing example - 6

A Prolog programmer’s version of the recursive clause:

classify([Head|Tail], [Head|Numbers], Atoms) :-
  number(Head),
  classify(Tail, Numbers, Atoms).

The Prolog programmer knows that Prolog matches clauses and goals by unification – so unifications can be written into the head of clauses.
Second list processing example - 7

A Prolog programmer’s version of the other recursive clause:

classify([Head|Tail], Numbers, [Head|Atoms]) :-
    atom(Head),
    classify(Tail, Numbers, Atoms).
Running classify/3- 1

Like many Prolog procedures, it can be run “forwards” and “backwards”.

Why does this happen?
– declarative languages don’t describe a process but a “situation”;
– the number recursive rule is true when the head of the first argument unifies with the head of the second argument;
– this isn’t a process/procedure but truth statement.

Demo4
Terminating when given element is found

For instance finding a given element:

% 1 - terminating
elem(Elem, [Elem|_]).

% 2 - recursive
elem(Elem, [_|Tail]) :-
    elem(Elem, Tail).

Notice, this can be run “backwards” to enumerate the individual elements of a list.

Demo5
Terminating when given number of elements have been scanned

% 1 - recursive
nth(Count, Item, [_|Tail]) :-
    Count > 1,
    Count0 is Count - 1,
    nth(Count0, Item, Tail).

% 2 - terminating
nth(1, Head, [Head|_]).

The code counts down from the given position to 1.
Consolidation moment

Three ways to halt recursion in list processing:

1. at the end of the list (\[\]);
2. when a specific element is found;
3. when a specific position in a list is reached.