COMP219: Artificial Intelligence

Lecture 9: Lists in Prolog

Overview

- Last time
  - Recursion in Prolog; infinite loops; structures; declarative vs procedural meaning.

- Today:
  - Lists: syntax of lists and writing procedures using lists;
  - Carrying out simple debugging

- Learning outcome covered today:

  Understand and write Prolog code to solve simple knowledge-based problems.

Last Week

- Recursion is a powerful construct essential to Prolog
- Take care with recursive rules to avoid an infinite sequence of recursive calls
- The order of clauses and goals does matter
- Declarative meaning vs. procedural meaning
- Structures

Recap: Structures

- Structures are a useful data structure in Prolog
- They are objects that have several components (terms) and a name (functor) that associates them together
  - `date(5, february, 2002)`
  - `location(depot1, manchester)`
  - `id_no(rajeev, gore, 02571)`
  - `state(ontable, onblock)`

Structures

- Both `location(depot1, manchester)` and `manchester` are known as *terms*
- Components of structured objects can themselves be structured

  `id_no(name(rajeev,gore),02571)`
- Structures can contain variables

  `location(X, manchester)` could be used in a program to mean any depot in Manchester – this is taken to mean ‘find values for X that are in Manchester’

Using Structured Objects in Procedures

```
% move( State1, Move, State2): making Move in State1 results in State2;
% a state is represented by a structure:
% state( MonkeyHorizontal, MonkeyVertical, BoxPosition, HasBanana)

move( state( middle, onBox, middle, noBanana),   % Before move
      grasp,                                     % Grasp banana
      state( middle, onBox, middle, banana) ).   % After move

move( state( P, onFloor, P, H),
      climb,                                     % Climb box
      state( P, onBox, P, H) ).

move( state( P1, onFloor, P1, H),
      push( P1, P2),                             % Push box from P1 to P2
      state( P2, onFloor, P2, H) ).

move( state( P1, onFloor, B, H),
      walk( P1, P2),                             % Walk from P1 to P2
      state( P2, onFloor, B, H) ).

canGet( state( _, _, _, banana) ).               % canGet 1: Monkey already has it
canGet( State1) :-                               % canGet 2: Do some work to get it
    move( State1, Move, State2),                 % Do something
    canGet( State2),                             % Get it now
```

Lists

- Lists are commonly used in Prolog
  `[clare,sean,richard,paula]`
- The first item in a list is the ***head*** of the list
- The remaining part is the ***tail*** of the list
- The tail is a list and head is an element of a list
  - In the above list `clare` is the head
  - whereas `[sean,richard,paula]` is the tail
- We can also represent the list with a pipe
  - `[clare |[sean,richard,paula]]`
  - which would match with `[Head|Tail]`

Pipe symbol | separates head from rest

Some Queries on List Patterns

Suppose a program exists with:
spectrum([red,orange,yellow,green,blue,indigo,violet]).

The argument is a ***list***

```
?- spectrum(X).
X = [red, orange, yellow, green, blue, indigo, violet].
?- spectrum([X,Y]).
false.
?- spectrum([X | Y]).
X = red,
Y = [orange, yellow, green, blue, indigo, violet].
?- spectrum([ [X] | Y]).
false.
?- spectrum([X,Y,Z | T]).
X = red,
Y = orange,
Z = yellow,
T = [green, blue, indigo, violet].
```

Some Queries on List Patterns

Suppose a program exists with:
spectrum([red,orange,yellow,green,blue,indigo,violet]).

The argument is a ***list***

?- spectrum(X).
X = [red, orange, yellow, green, blue, indigo, violet].
?- spectrum([X,Y]).
false.

Asks for a list with ***exactly two terms***

?- spectrum([X | Y]).
X = red,
Y = [orange, yellow, green, blue, indigo, violet].
?- spectrum([[X] | Y]).
false.
?- spectrum([X,Y,Z | T]).
X = red,
Y = orange,
Z = yellow,
T = [green, blue, indigo, violet].

Some Queries on List Patterns

Suppose a program exists with:
spectrum([red,orange,yellow,green,blue,indigo,violet]).

The argument is a ***list***

```
?- spectrum(X).
X = [red, orange, yellow, green, blue, indigo, violet].
?- spectrum([X,Y]).
false.
```

Asks for a list with ***exactly two terms***

```
?- spectrum([X | Y]).
X = red,
Y = [orange, yellow, green, blue, indigo, violet].
```

The variable ***following pipe*** binds to a list

```
?- spectrum([ [X] | Y]).
false.
?- spectrum([X,Y,Z | T]).
X = red,
Y = orange,
Z = yellow,
T = [green, blue, indigo, violet].
```

Some Queries on List Patterns

Suppose a program exists with:
spectrum([red,orange,yellow,green,blue,indigo,violet]).

The argument is a ***list***

?- spectrum(X).
X = [red, orange, yellow, green, blue, indigo, violet].
?- spectrum([X,Y]).
false.

Asks for a list with ***exactly two terms***

?- spectrum([X | Y]).
X = red,
Y = [orange, yellow, green, blue, indigo, violet].

The variable ***following pipe*** binds to a list

?- spectrum([[X] | Y]).
false.

The first term is ***not a list***

?- spectrum([X,Y,Z | T]).
X = red,
Y = orange,
Z = yellow,
T = [green, blue, indigo, violet].

Some Queries on List Patterns

Suppose a program exists with:
spectrum([red,orange,yellow,green,blue,indigo,violet]).

The argument is a ***list***

?- spectrum(X).
X = [red, orange, yellow, green, blue, indigo, violet].
?- spectrum([X,Y]).
false.

Asks for a list with ***exactly two terms***

?- spectrum([X | Y]).
X = red,
Y = [orange, yellow, green, blue, indigo, violet].

The variable ***following pipe*** binds to a list

?- spectrum([[X] | Y]).
false.

The first term is ***not a list***

?- spectrum([X,Y,Z | T]).
X = red,
Y = orange,
Z = yellow,
T = [green, blue, indigo, violet].

Asks for the first three terms in a list and then the rest of the tail.
Succeeds if the list contains ***at least three*** terms.

Member

```
tmember(H,[H|Tail]).
tmember(X,[H|Tail]):-
  tmember(X,Tail).
```

Base: H is a member of a list if it is the 1st term

Recursive: X is a member of a list if it is on the tail

The query

```
?- tmember(richard,[clare,sean,richard,paula]).
```

doesn't match with the first clause but does with the second, where:

```
X=richard, H=clare, Tail=[sean,richard,paula].
```

Creates the new subgoal:

```
tmember(richard,[sean,richard,paula]).
```

Matches the second clause again:

```
X = richard, H = sean, Tail = [richard,paula].
```

Creates the new subgoal :

```
tmember(richard,[richard,paula]).
```

Matches the base case and succeeds - richard is a member of the list – returns 'true'

Member

```
tmember(H,[H|Tail]).
tmember(X,[H|Tail]):-
  tmember(X,Tail).
```

Base: H is a member of a list if it is the 1st term

Recursive: X is a member of a list if it is on the tail

The query

```
?- tmember(richard,[clare,sean,richard,paula]).
```

doesn't match with the first clause but does with the second, where:

```
X=richard, H=clare, Tail=[sean,richard,paula].
```

Creates the new subgoal:

```
tmember(richard,[sean,richard,paula]).
```

Matches the second clause again:

```
X = richard, H = sean, Tail = [richard,paula].
```

Creates the new subgoal :

```
tmember(richard,[richard,paula]).
```

Matches the base case and succeeds - richard is a member of the list – returns 'true'

NB: Prolog has a built in 'member' functor; we write our own for study by including the prefix 't'

Member Failing

- The query

  `?- tmember(john,[clare,sean,richard,paula]).`

  keeps recursively calling `tmember` as follows

  ```
  tmember(john,[sean,richard,paula]).
  tmember(john,[richard,paula]).
  tmember(john,[paula]).
  tmember(john,[]).
  ```

- There is no rule to apply to the empty list so `tmember(john,[])` fails

- `tmember(X,[X|T])` needs ***at least one*** element to match the X

Exercise

- Suppose we have a program comprised of the following:

```
list1([[boy,girl],cat,[]]).
```

- What answers will we get for the following queries?

1) `list1(X,Y,Z).`

2) `list1([X,Y,Z]).`

3) `list1([X | Y]).`

Debugging Prolog Programs

- The debugging tool called tracing is invoked by typing trace at the prompt
  ?- trace.

- You can also enter trace mode by typing
  ?- trace, whatever-your-query-is.

- This allows you to follow step-by-step how Prolog is evaluating the query in trying to satisfy the goal.
  - Pressing return will give the new goal or whether the current goal has succeeded or failed.
- To turn trace off you can type nodebug.

- Advice for debugging:
  - Test smaller units e.g. individual procedures
  - Look out for infinite recursions

- This is using trace in Unix. In Windows, there are some variations.

Tracing a Successful Goal

```
?- trace, tmember(richard,[clare,sean,richard,paula]).
Call: (7) tmember(richard, [clare, sean, richard, paula]) ? creep
Call: (8) tmember(richard, [sean, richard, paula]) ? creep
Call: (9) tmember(richard, [richard, paula]) ? creep
Exit: (9) tmember(richard, [richard, paula]) ? creep
Exit: (8) tmember(richard, [sean, richard, paula]) ? creep
Exit: (7) tmember(richard, [clare, sean, richard, paula]) ? creep
true .
```

Does not tell you which clause is called; goes down to what succeeds (if any); passes success back up to the topmost goal.

Note: following trace instructions for Unix/Linux.

Tracing a Failing Goal

```
?- trace, tmember(john,[clare,sean,richard,paula]).
   Call: (7) tmember(john, [clare, sean, richard, paula]) ? creep
   Call: (8) tmember(john, [sean, richard, paula]) ? creep
   Call: (9) tmember(john, [richard, paula]) ? creep
   Call: (10) tmember(john, [paula]) ? creep
   Call: (11) tmember(john, []) ? creep
   Fail: (11) tmember(john, []) ? creep
   Fail: (10) tmember(john, [paula]) ? creep
   Fail: (9) tmember(john, [richard, paula]) ? creep
   Fail: (8) tmember(john, [sean, richard, paula]) ? creep
   Fail: (7) tmember(john, [clare, sean, richard, paula]) ? Creep
   false.
```

No variables in initial query, so no re-doing values.

Goes down to where it fails; passes failure back up to topmost goal (unless there are other branches to explore).

Append

- tappend is a useful example of list processing

```
/*******************/
% tappend(L1,L2,L3)
% takes two lists L1 and L2 and returns a
% list L3 which is the result of appending
% L2 to L1
/******************/
tappend([],L2,L2).

tappend([H1|L1],L2,[H1|L3]) :-
   tappend(L1,L2,L3).
```

The third argument must be a variable (or equal to L2) so it matches L2.

Append

- tappend is a useful example of list processing

```
/******************/
% tappend(L1,L2,L3)
% takes two lists L1 and L2 and returns a
% list L3 which is the result of appending
% L2 to L1
/*****************/
tappend([],L2,L2).

tappend([H1|L1],L2,[H1|L3]) :-
   tappend(L1,L2,L3).
```

Base: appending an empty list to a list gives that list.

The third argument must be a variable (or equal to L2) so it matches L2.

Append

- tappend is a useful example of list processing

```
/*******************/
% tappend(L1,L2,L3)
% takes two lists L1 and L2 and returns a
% list L3 which is the result of appending
% L2 to L1
/******************/
tappend([],L2,L2).

tappend([H1|L1],L2,[H1|L3]) :-
   tappend(L1,L2,L3).
```

Base: appending an empty list to a list gives that list.

Recursive: where the first list is not empty, make the head of the first list the head of the third list. The second list applies where the base is satisfied.

The third argument must be a variable (or equal to L2) so it matches L2.

Append in Action

```
?- trace, tappend([a,b],[c,d],E).
   Call: (7) tappend([a, b], [c, d], _G2168) ? creep
   Call: (8) tappend([b], [c, d], _G2285) ? creep
   Call: (9) tappend([], [c, d], _G2288) ? creep
   Exit: (9) tappend([], [c, d], [c, d]) ? creep
   Exit: (8) tappend([b], [c, d], [b, c, d]) ? creep
   Exit: (7) tappend([a, b], [c, d], [a, b, c, d]) ? creep
E = [a, b, c, d].
```

Find a value for E that makes the query true.

Append in Action

```
?- trace, tappend([a,b],[c,d],E).
   Call: (7) tappend([a, b], [c, d], _G2168) ? creep
   Call: (8) tappend([b], [c, d], _G2285) ? creep
   Call: (9) tappend([], [c, d], _G2288) ? creep
   Exit: (9) tappend([], [c, d], [c, d]) ? creep
   Exit: (8) tappend([b], [c, d], [b, c, d]) ? creep
   Exit: (7) tappend([a, b], [c, d], [a, b, c, d]) ? creep
E = [a, b, c, d].
```

Find a value for E that makes the query true.

Base clause instantiates L3 to L2. Then they differ.

Every recursive call to
tappend(L1,L2,L3)
takes off one more element from L1 until the base clause can be satisfied, making L3 the same list as L2. Then it backtracks through the calls, instantiates the variables, so recursively adds the head of L1 to the head of L3.

Exercise

1. What will be the answer to the following query?

```
append([A],B,[c]).
```

2. What will be the answer to the following query?

```
append([a],B,[C]).
```

Append Again (decomposing a list)

```
?- trace,tappend(A,B,[a,b,c]).
   Call:   (7) tappend(_G2163, _G2164, [a, b, c]) ? creep
   Exit:   (7) tappend([], [a, b, c], [a, b, c]) ? creep
A = [],
B = [a, b, c] ;

   Redo: (7) tappend(_G2163, _G2164, [a, b, c]) ? creep
   Call:   (8) tappend(_G2291, _G2164, [b, c]) ? creep
   Exit:   (8) tappend([], [b, c], [b, c]) ? creep
   Exit:   (7) tappend([a], [b, c], [a, b, c]) ? creep
A = [a],
B = [b, c] ;

   Redo: (8) tappend(_G2291, _G2164, [b, c]) ? creep
   Call:   (9) tappend(_G2294, _G2164, [c]) ? creep
   Exit:   (9) tappend([], [c], [c]) ? creep
   Exit:   (8) tappend([b], [c], [b, c]) ? creep
   Exit:   (7) tappend([a, b], [c], [a, b, c]) ? creep
A = [a, b],
B = [c] ;

   Redo: (9) tappend(_G2294, _G2164, [c]) ? creep
   Call:   (10) tappend(_G2297, _G2164, []) ? creep
   Exit:   (10) tappend([], [], []) ? creep
   Exit:   (9) tappend([c], [], [c]) ? creep
   Exit:   (8) tappend([b, c], [], [b, c]) ? creep
   Exit:   (7) tappend([a, b, c], [], [a, b, c]) ? creep
A = [a, b, c],
B = [] ;

   Redo: (10) tappend(_G2297, _G2164, []) ? creep
   Fail:   (10) tappend(_G2297, _G2164, []) ? creep
   Fail:   (9) tappend(_G2294, _G2164, [c]) ? creep
   Fail:   (8) tappend(_G2291, _G2164, [b, c]) ? creep
   Fail:   (7) tappend(_G2163, _G2164, [a, b, c]) ? creep

false.
```

In the redos, different possible solutions are considered though the goal is the same.

Summary

- Lists are common data structures in Prolog
- Procedures related to lists are often recursive
  - Do it to the head, then do it to the rest
- Tracing a procedure can help you see what your program is doing

- Next time
  - Search in complex environments