Logic Programming and Prolog

Keep reading: Scott, Chapter 12
Lecture Outline

- Prolog
  - Lists
  - Programming with lists
  - Arithmetic
Lists

<table>
<thead>
<tr>
<th>list</th>
<th>head</th>
<th>tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a, b, c]</td>
<td>a</td>
<td>[b, c]</td>
</tr>
<tr>
<td>[X, [cat], Y]</td>
<td>X</td>
<td>[[cat], Y]</td>
</tr>
<tr>
<td>[a, [b, c], d]</td>
<td>a</td>
<td>[[b, c], d]</td>
</tr>
<tr>
<td>[X</td>
<td>Y]</td>
<td>X</td>
</tr>
</tbody>
</table>

Programming Languages CSCI 4430, A. Milanova
Lists: Unification

- \([ H_1 \mid T_1 ] = [ H_2 \mid T_2 ]\)
  - Head \(H_1\) unifies with \(H_2\), possibly recursively
  - Tail \(T_1\) unifies with \(T_2\), possibly recursively

- E.g., \([ a \mid [b, c] ] = [ X \mid Y ]\)
  - \(X = a\)
  - \(Y = [b, c]\)

- NOTE: In Prolog, = denotes unification, not assignment!
Question

- \([X,Y,Z] = [\text{john, likes, fish}]\)
  - \(X = \text{john, } Y = \text{likes, } Z = \text{fish}\)

- \([\text{cat}] = [X | Y]\)
  - \(X = \text{cat, } Y = [ ]\)

- \([[[\text{the, } Y]|Z] = [[[X, hare]|[is,here]]]\)
  - \(X = \text{the, } Y = \text{hare, } Z = [\text{is, here}]\)
Lists: Unification

- Sequence of comma separated terms, or
- `[ first term | rest_of_list ]`

```
[ [the | Y] | Z ]    = [ [X, hare] | [is, here] ]
```
Lists Unification

- Look at the trees to see how this works!

\[
\begin{align*}
[a, b, c] &= [X \mid Y] \\
X &= a, \ Y &= [b, c].
\end{align*}
\]

\[
\begin{align*}
[a \mid Z] &=? [X \mid Y] \\
X &= a, \ Y &= Z.
\end{align*}
\]
Improper and Proper Lists

\[[1 \mid 2]\] versus \[[1, 2]\]

1 \quad 2

1

2 [ ]
Question. Can we unify these lists?

\[ [abc, Y] \quad =? \quad [abc \mid Y] \]

Answer: No. There is no value binding for \( Y \) that makes these two trees isomorphic.
Aside: The Occurs check
Lecture Outline

- Prolog
  - Lists
  - Programming with lists
  - Arithmetic
Member_of

?- member(a,[a,b]).
   true.

?- member(a,[b,c]).
   false.

?- member(X,[a,b,c]).
   X = a ;
   X = b ;
   X = c ;
   false.

1. member(A, [A | B]).
2. member(A, [B | C]) :- member(A, C).
Member_of

?- member(a,[a,b]).
  true.
?- member(a,[b,c]).
  false.
?- member(X,[a,b,c]).
  X = a ;
  X = b ;
  X = c.
?- member(a,[b,c,X]).
  X = a ;
  false.

1. member(A, [A | B]).
2. member(A, [B | C]) :- member (A, C).
1. member(A, [A | B]).
2. member(A, [B | C]) :- member(A, C).
Member_of

\[
\text{member}(A, [A|B]). \\
\text{member}(A, [B|C]) :- \text{member}(A,C).
\]

**logical semantics:** For every \(A, B\) and \(C\)

\[
\text{member}(A, [B|C]) \text{ if } \text{member}(A,C);
\]

**procedural semantics:** Head of clause is procedure entry. Tail of clause is procedure body; subgoals correspond to calls.
“Procedural” Interpretation

member(A, [A|B]).
member(A, [B|C]) :- member(A,C).

member is a recursive “procedure”

member(A, [A|B]). is the base case. “Procedure” exits with true if the element we are looking for, A, is the first element in the list. It exits with false if we have reached the end of the list.

member(A, [B|C]) :- member(A,C). is the recursive case. If element A is not the first element in the list, call member recursively with arguments A and tail C.
Give all answers to the following query:
?- member(a,[b, a, X]).

Answer:
true ;
X = a ;
false.
Question

Give all answers to the following query:
?- member(a, [b | a]).

Answer:
false.
Append

\begin{align*}
\text{append}([\ ], A, A). \\
\text{append}([A|B], C, [A|D]) & : - \text{append}(B,C,D).
\end{align*}

\begin{itemize}
  \item Build a list:
  \begin{align*}
  \text{? - append}([a,b,c],[d,e],Y). \\
  Y & = [a,b,c,d,e]
  \end{align*}
  \end{itemize}

\begin{itemize}
  \item Break a list into constituent parts:
  \begin{align*}
  \text{? - append} (X,Y,[a,b]). \\
  X & = [], Y = [a,b]; X = [a], Y = [b]; \\
  X & = [a,b], Y = []; \text{false.}
  \end{align*}
  \end{itemize}
More Append

\begin{verbatim}
append([], A, A).
append([A|B], C, [A|D]) :- append(B,C,D).
\end{verbatim}

- Break a list into constituent parts

\begin{verbatim}
?- append(X,[b],[a,b]).
X = [ a ]
?- append([a],Y,[a,b]).
Y = [ b ]
\end{verbatim}
More Append

?- append(X,Y,[a,b]).
X = [ ]
Y = [a,b] ;
X = [a]
Y = [b] ;
X = [a,b]
Y = [ ] ;
false.
Unbounded Arguments

- Generating an unbounded number of lists
  
  \[
  \text{?- append}(X,[b],Y).
  \]
  
  \[
  X = [ ]
  \]
  
  \[
  Y = [b] ;
  \]
  
  \[
  X = [ _G604 ]
  \]
  
  \[
  Y = [ _G604, b] ;
  \]
  
  \[
  X = [ _G604, _G610 ]
  \]
  
  \[
  Y = [ _G604, _G610, b] ;
  \]
  
  Etc.

- Be careful when using append with 2 unbounded arguments!

An underscore, “don’t care” variable. Unifies with anything. E.g., \text{bad}(\text{Dog}) :- \text{bites}(\text{Dog},_).
Question

What does this “procedure” do:

\[ p([],[]). \]
\[ p([A|B],[[A]|Rest]) :- p(B,Rest). \]

?\ - p([a,b,c],Y).
\[ Y = [ [a],[b],[c] ] \]

Can also “flatten” a list:

?\ - p(X,[[a],[b],[c]]).
\[ X = [ a,b,c ] \]
Common Structure

- “Processing” a list:

  \[
  \text{proc}([],[]).
  \text{proc}([H|T],[H1|T1]) : - \ f(H,H1), \text{proc}(T,T1).
  \]

- Base case: we have reached the end of list. In our case, the result for \([\ ]\) is \([\ ]\).

- Recursive case: result is \([H1|T1]\). \(H1\) was obtained by calling \(f(H,H1)\) --- processes element \(H\) into result \(H1\). \(T1\) is the result of recursive call of \text{proc} on \(T\).
Lecture Outline

- Prolog
  - Lists
  - Programming with lists
  - Arithmetic
Arithmetic

- Prolog has all arithmetic operators
- Built-in predicate **is**
  - `is(X, 1+3)` or more commonly we write
  - `X is 1+3`
  - `is` forces evaluation of `1+3`:
  ```
  ?- X is 1+3
  X = 4
  ```
  - `is` is unification not assignment!
  ```
  ?- X = 4-1.
  X = 4-1 % unifies X with 4-1!!!
  ```
Arithmetic: Pitfalls

- **is** is not invertible! That is, arguments on the right cannot be unbound!
  - 3 **is** 3 – X.
  
  **ERROR:** is/2: Arguments are not sufficiently instantiated

- This doesn’t work either:
  
  `- X is 4, X = X+1.
   false.

  Why? What’s going on here?
Exercise

- Write `sum`, which takes a list of integers and computes the sum of the integers. E.g.,
  
  `sum([1,2,3],R).
  \(?- R = 6.`

- How about if the integers are arbitrarily nested? E.g.,
  
  `sum([[1],[[[2]],3]],R).
  \(?- R = 6.`
Exercise
Exercise

- Write `plus10`, which takes a list of integers and computes another list, where all integers are shifted +10. E.g.,
  `plus10([1,2,3],R).` 
  `- R = [11,12,13].`

- Write `len`, which takes a list and computes the length of the list. E.g.,
  `len([1,[2],3],R).` 
  `- R = 3.`
Exercise

- Write **atoms**, which takes a list and computes the number of atoms in the list. E.g.,
  
  \[\text{atoms}([a, [b, [[[c]]]]], R).\]
  
  ?- R = 3.

- Hint: built-in predicate **atom(X)** yields true if \(X\) is an atom (i.e., symbolic constant such as \(x, \text{abc}, \text{tom}\)).
The End