COMP4418: Knowledge Representation and Reasoning

Introduction to Prolog II

Maurice Pagnucco
School of Computer Science and Engineering
University of New South Wales
NSW 2052, AUSTRALIA
morri@cse.unsw.edu.au

Prolog

- Compound terms can contain other compound terms
- A compound term can contain the same kind of term, i.e., it can be recursive:
  \[
  \text{tree}(\text{tree}(\text{empty, jack, empty}), \text{fred, tree}(\text{empty, jill, empty}))
  \]
- “empty” is an arbitrary symbol used to represent the empty tree
- A structure like this could be used to represent a binary tree that looks like:

```
          fred
         /   \   
   jack         jill
  /   \       /   \ 
empty  empty empty  empty
```
Binary Trees

- A binary tree is either empty or it is a structure that contains data and left and right subtrees which are also binary trees.

- To test if some datum is in the tree:
  
  ```prolog
  in_tree(X, tree(_, X, _)).
  in_tree(X, tree(Left, Y, _) :-
    X \= Y, in_tree(X, Left).
  in_tree(X, tree(_, Y, Right) :-
    X \= Y, in_tree(X, Right).
  ```
The Size of a Tree

- \text{tree\_size}(\text{empty}, 0).
- \text{tree\_size}(\text{tree}(\text{Left}, _, \text{Right}), \text{N}) \leftarrow
  \text{tree\_size}(\text{Left}, \text{LeftSize}),
  \text{tree\_size}(\text{Right}, \text{RightSize}),
  \text{N} \text{ is } \text{LeftSize} + \text{RightSize} + 1.

- The size of the empty tree is 0
- The size of a non-empty tree is the size of the left subtree plus the size of the right subtree plus one for the current node
Lists

- A list may be nil or it may be a term that has a head and a tail. The tail is another list.
- A list of numbers, [1, 2, 3] can be represented as:
  \[
  \text{list}(1, \text{list}(2, \text{list}(3, \text{nil})))
  \]
- Since lists are used so often, Prolog has a special notation:
  \[
  [1, 2, 3] = \text{list}(1, \text{list}(2, \text{list}(3, \text{nil})))
  \]
### Examples of Lists

\[ [X, Y, Z] = [1, 2, 3] \]

Unify the two terms on either side of the equals sign

<table>
<thead>
<tr>
<th>( X )</th>
<th>( Y )</th>
<th>( Z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Variables match terms in corresponding positions

\[ [X|Y] = [1, 2, 3] \]

The head and tail of a list are separated by using ‘|’ to indicate that the term following the bar should unify with the tail of the list

<table>
<thead>
<tr>
<th>( X )</th>
<th>( Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[2, 3]</td>
</tr>
</tbody>
</table>

\[ [X|Y] = [1] \]

The empty list is written as ‘[]’

<table>
<thead>
<tr>
<th>( X )</th>
<th>( Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[]</td>
</tr>
</tbody>
</table>

The end of a list is usually ‘[]’
More list examples

\[ X, Y|Z \] = \[fred, jim, jill, mary\]?

There must be at least two elements in the list on the right

\[ X = \text{fred} \]
\[ Y = \text{jim} \]
\[ Z = \text{[jill, mary]} \]

\[ X|Y \] = \[[a, f(e)], [n, b, [2]]\]?

The right hand list has two elements:

\[ X = \text{[a, f(e)]} \]
\[ Y = \text{[[n, b, [2]]]} \]

\[ a, f(e) \]
\[ n, b, [2] \]

\[ Y \] is the tail of the list,
\[ [n, b, [2]] \] is just one element
List Membership

\[
\text{member}(X, [X|\_]).
\]
\[
\text{member}(X, [\_|Y]) :-
\]
\[
\quad \text{member}(X, Y).
\]

- Rules about writing recursive programs:
  - Only deal with one element at a time
  - Believe that the recursive program you are writing has already been written and works
  - Write definitions, not programs
Appending Lists

- A commonly performed operation on lists is to append one list to the end of another (or, concatenate two lists), e.g.,
  \[
  \text{append}([1, 2, 3], [4, 5], [1, 2, 3, 4, 5]).
  \]

- Start planning by considering the simplest case:
  \[
  \text{append}([], [1, 2, 3], [1, 2, 3]).
  \]

- Clause for this case:
  \[
  \text{append}([], L, L).
  \]
Appending Lists

- Next case:
  
  ```prolog
  append([1], [2], [1, 2]).
  ```

- Since `append([], [2], [2])`:
  
  ```prolog
  append([H|T1], L, [H|T2]) :- append(T1, L, T2).
  ```

- Entire program is:
  
  ```prolog
  append([], L, L).
  append([H|T1], L, [H|T2]) :-
      append(T1, L, T2).
  ```
Reversing Lists

- \texttt{rev([1, 2, 3], [3, 2, 1]).}

- Start planning by considering the simplest case:
  \texttt{rev([], []).}

- Note:
  \texttt{rev([2, 3], [3, 2]).}
  and
  \texttt{append([3, 2], [1], [3, 2, 1]).}
Reversing Lists

Entire program is:

\[
\text{rev([], []).} \\
\text{rev([A|B], C) :-} \\
\hspace{1em} \text{rev(B, D),} \\
\hspace{1em} \text{append(D, [A], C).}
\]
An Application of Lists

- Find the total cost of a list of items:
  ```prolog
cost(flange, 3).
cost(nut, 1).
cost(widget, 2).
cost(splice, 2).
```

- We want to know the total cost of `[flange, nut, widget, splice]`
  ```prolog
total_cost([], 0).
total_cost([A|B], C) :-
  total_cost(B, B_cost),
  cost(A, A_cost),
  C is A_cost + B_cost.
```