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:
  \[
  \text{tree(tree(} \text{empty, jack, empty)\text{, fred, tree(} \text{empty, jill, empty)\text{))}
  \]
- "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:
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

- 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

```
tree_size(empty, 0).
\[ \text{tree_size(tree(Left, _, Right), N)) :-} \]
  tree_size(Left, LeftSize),
  tree_size(Right, RightSize),
  N is LeftSize + RightSize + 1.
```
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:
  
  ```prolog
  list(1, list(2, list(3, nil)))
  ```

- Since lists are used so often, Prolog has a special notation:
  
  ```prolog
  [1, 2, 3] = list(1, list(2, list(3, nil)))
  ```
Examples of Lists

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

Unify the two terms on either side of the equals sign

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

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

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

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

The empty list is written as ‘[]’

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

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

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

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

The right hand list has two elements:

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

\[ [a, f(e)] [n, b, [2]] \]
\[ Y is the tail of the list, \]
\[ [n, b, [2]] is just one element \]
List Membership

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

- 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:

append([1], [2], [1, 2]).

Since append([], [2], [2]):

append([H|T1], L, [H|T2]) :- append(T1, L, T2).

Entire program is:

append([], L, L).
append([H|T1], L, [H|T2]) :-
    append(T1, L, T2).
Reversing Lists

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

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

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

Entire program is:

\[
\text{rev}([], []) .
\text{rev}([A|B], C) :-
\text{rev}(B, D),
\text{append}(D, [A], C).
\]
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.
```