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

- tree_size(empty, 0).
- tree_size(tree(Left, _, Right), N) :-
  tree_size(Left, LeftSize),
  tree_size(Right, RightSize),
  N is LeftSize + 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:
  
  ```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

\[
\begin{align*}
[X, Y|Z] & = [\text{fred, jim, jill, mary}]? \\
X & = \text{fred} \\
Y & = \text{jim} \\
Z & = [\text{jill, mary}]
\end{align*}
\]

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

\[
\begin{align*}
[X|Y] & = [[\text{a, f(e)}], [\text{n, b, [2]}]]? \\
X & = [\text{a, f(e)}] \\
Y & = [[\text{n, b, [2]}]]
\end{align*}
\]

The right hand list has two elements:

\[
\begin{align*}
[a, f(e)] & \quad [n, b, [2]] \\
Y & \quad \text{is the tail of the list,} \\
[n, b, [2]] & \quad \text{is just one element}
\end{align*}
\]
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
Applying 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:
  \[
  \text{append}([1], [2], [1, 2]).
  \]

- Since \(\text{append}([], [2], [2])\):
  \[
  \text{append}([\text{H}|\text{T1}], \text{L}, [\text{H}|\text{T2}]) :\text{ append}(\text{T1}, \text{L}, \text{T2}).
  \]

- Entire program is:
  \[
  \text{append}([], \text{L}, \text{L}).
  \]
  \[
  \text{append}([\text{H}|\text{T1}], \text{L}, [\text{H}|\text{T2}]) :\text{ append}(\text{T1}, \text{L}, \text{T2}).
  \]
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:

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

Find the total cost of a list of items:
\[
\begin{align*}
\text{cost(flange, 3).} \\
\text{cost(nut, 1).} \\
\text{cost(widget, 2).} \\
\text{cost(splice, 2).}
\end{align*}
\]

We want to know the total cost of \([\text{flange, nut, widget, splice}]\)
\[
\begin{align*}
\text{total\_cost([], 0).} \\
\text{total\_cost([A|B], C) :-} \\
\quad \text{total\_cost(B, B\_cost),} \\
\quad \text{cost(A, A\_cost),} \\
\quad C \text{ is A\_cost + B\_cost.}
\end{align*}
\]