Logic Programming Alan Smaill

2/10/2015

Tutorial for week 4 (12–16 Oct)

Recursion, trees, lists

1. A notion of binary tree where data is stored at the leave only is given by the following characterisation:

```
btree( leaf(L) ).
btree( node( TL, TR ) ) :- btree( TL ), btree( TR ).
```

   (a) What Prolog term represents this tree labelled by integers:

```
      .
     / \
    4   .
       / \
       6 8
```

   (b) Define a predicate `mirror/2` which relates a tree to its (left/right) mirror image, and check it works on your tree above. Check that applying it twice returns the original tree.

   (c) Define a predicate `fringe/2` such that `fringe(Tree,Fringe)` holds when`Fringe` is the list of values held in the leaves of the tree, in left-right order. For example, if `yourT` is the Prolog representation of the tree above, we have `fringe(yourT, [4,6,8])`.

   For this part, your definition may use the built-in append/3 predicate but no other built-in or helper functions.

   Trace the behaviour of query `?- fringe(yourT,X)`.

   What happens if you pose query `?- fringe(X,[1,2,3,4])` , and ask for multiple solutions?

   What is the complexity of this implementation of `fringe/2` ? You can assume that append/3 is linear in the size of its first argument.

   (d) (*) It is possible to write `fringe/2` without using append, and indeed no helper functions at all, such that it runs in time linear in the size of the tree involved, though this is not so easy to find. Can you find such a definition, using only pattern matching on the tree structure?

2. **Shuffling**

   Given two lists, a shuffle is a list consisting of alternating elements from the two lists, starting with the first. If one of the lists is empty, then shuffling just returns the other list.

   For example:

   ```
   shuffle([],[1,2,3,4],[1,2,3,4]).
   shuffle([1,2],[3],[1,3,2]).
   shuffle([1,2],[3,4],[1,3,2,4]).
   ```

   Here is a simple definition of `shuffle/3`:

   ```
   simple_shuffle([],L,L).
   simple_shuffle(L,[],L).
   simple_shuffle([X|L],[Y|M],[X,Y|N]) :-
                simple_shuffle(L,M,N).
   ```

   What happens if you ask the query:

   ```
       ?- simple_shuffle([1,2],[3,4],X).
   ```

   What about `?- simple_shuffle(X,Y,[1,2,3,4])` ?

   Define an improved `shuffle/3` such that if `L1` and `L2` are ground then `shuffle(L1,L2,L3)` returns exactly one answer.

3. (*) **Bridge dealing**

   In a four-player game of bridge, each player gets 13 cards, dealt in order. Write a predicate `deal(Cards, H1, H2, H3, H4)` that takes a first argument, and succeeds by binding `H1` to the 13 cards received by player 1 in the deal, etc.

   Hint: One strategy is to write four helper predicates `deal1` that deals to player 1, `deal2` that deals to player 2, etc.

4. (**) **Cutting the deck**

   Write a predicate`cut/3` such that if `L` is a list with even length, then `cut(L,M,N)` succeeds by binding `M` to the first half of `L` and `N` to the second half.

   Hint: One can get `M` and `N` by generating possible splits of `L` using `append/3`, and defining a predicate `same_length/2` that holds of two lists whenever they have the same length (ignoring their element values). Another way to do this is to use the built-in `length/2` predicate.