Knowledge representation (INFO0049-1)

Exercise session 5

17 Mar 2015

Try to draw search trees wherever possible to see how prolog executes a query

I. Puzzles

1. Three friends ranked first, second and third in a chess tournament. Each has a first name and a nationality different from the two others, practices a sport that the two others don't. Michel plays basket and comes ahead the American. Simon, the French man, ranks better than the tennis player. The rugby player finishes at the first place. Who is the Australian? Which is the sport of Richard? Define a predicate to solve this enigma.

II. Meta predicate - findall

Some things cannot be expressed in 'plain' Prolog because you need *meta predicates*. The most common case is when you want to find all solutions for a query at once (instead of retrieving them manually by typing ';'). We can solve this by using the built-in predicate findall/3. The first argument indicates which part of the query you want to store, the second argument is the query for which we need all results and the third argument is the name of the variable (a list) in which we place the results. A few examples:

```
?- findall(Elem,member(Elem,[a,b,c]),List).
List = [a,b,c]

?- findall((Elem,Fib), (member(Elem,[5,10,15,20]),fibo(Elem,Fib)), Res).

Res = [(5, 5), (10, 55), (15, 610), (20, 6765)]
```

Notice this last example: a query can be a conjunction, but make sure you use extra parentheses to group the conjuncts into a single argument. The same holds for the result variables we want to see. In this case we want pairs (N,fib(N)), so we also use extra parentheses.

Try to solve the next problems using findall/3.

2. During a student initiation ritual some years ago, several students got injured when they came in contact with a mix of 'daily use' products such as vinegar, oil,... . To prevent such accidents from happening in the future, you are asked to develop a system so that you can test whether a certain mixture is dangerous or not. To help you, a chemical expert gives you a database of mixtures of 'daily products' and numbers indicating how dangerous the mixtures are. The database is written as prolog facts:

   ```
   reacts(vinegar,salt,25).
   reacts(salt,water,3).
   reacts('brown soap',water,10).
   reacts('pili pili', milk,7).
   reacts(tonic,bailey,8).
   ```

   Higher numbers are more dangerous. The order of the elements is not important: e.g. salt and vinegar react the same as vinegar and salt do. Now suppose that you can add the numbers: e.g. the number for a mixture of vinegar, salt and water is 25 (vinegar & salt) + 3 (salt & water) = 28. Also, suppose that a number less than 5 results in no irritation, between 6 and 12 results in minor irritation, between 13 and 20 results in minor burning wounds, between 21 and 30 results in severe burning wounds and higher values can be lethal. Write a predicate advice(+Mixture) that given a mixture writes to the screen what the result of using that mixture would be, e.g.

   ```
   ?- advice([vinegar,salt,water]).
   Warning: this mixture causes severe burning wounds, never use this!
   Yes

   ?- advice([jam, salt, pepper, water, 'red onions', 'brown soap']).
   Warning: this mixture could result in minor burning wounds! Yes
   ```

III. Accumulators

An often-used technique in Prolog is that of using an accumulating parameter: we gradually build up ('accumulate') a result in a parameter, and we return this result when we arrive in the base case. An advantage of accumulating parameters is that they can be used to make a program 'tail-recursive' (roughly speaking, a program is tail-recursive if the recursive call is at the very end in the body of the recursive clause).

3. Write a predicate reverse_list(+List, -List_Reversed) that computes the reverse of List using an accumulator and that is tail-recursive.

   ```
   ?- reverse_list([1, 2, 3],List).

   List = [3, 2, 1].
   ```

The program without accumulator has quadratic time-complexity (i.e. the time needed to reverse a list with N elements using this program is proportional to N^2)! Using an accumulating parameter, we can get a program with linear time-complexity (time proportional to N).

4. Catalan numbers come from combinatorial problems like determining the number C_N of strict binary trees with N internal nodes. The Catalan numbers satisfy the following recurrence relation:

$$C(0) = 1$$

$$For\ N > 0, \qquad C(N) = \sum_{i=0}^{N-1} C(i)C(N-1-i)$$

Write a predicate catalan(+Number, -Result) that calculates the catalan number of N. Compare the naïve version with that of the version using accumulators.

5. Write a predicate get_doubles(+List, -Doubles) to extract from a list all elements that occur at least twice. Try to do this as efficiently as possible using accumulators.

?- get_doubles([1, 2, 3, 4, 1, 4, 5], List).

List = [4, 1].