2. tutorial in Prolog

April 12, 2016

Task 1: Load the database of the British royal family from the last tutorial into Prolog. If you've lost it, it is provided at the end of this document.

Task 2: Compare 4 implementations of `ancestor(Anc,Desc)`, which should connect parents with children, grandparents with grandchildren, grand-grandchildren with grand-grand-children, etc.

```
ancestor1(A,D) :- parent(A,D).
ancestor1(A,D) :- parent(A,B), ancestor1(B,D).

ancestor2(A,D) :- parent(A,D).
ancestor2(A,D) :- ancestor2(B,D), parent(A,B).

ancestor3(A,D) :- parent(A,B), ancestor3(B,D).
ancestor3(A,D) :- parent(A,D).

ancestor4(A,D) :- ancestor4(B,D), parent(A,B).
ancestor4(A,D) :- parent(A,D).
```

First without your computer, guess which implementation matches the behavior. Please, please, please, verify your answers on a computer *after* you made your guess.

a. Works exactly as expected.

b. Ends up in an (almost) infinite loop immediately.

c. Works as expected, but finds grand-parents and grand-children before parents and children.

d. Works as expected, but after the last response, Prolog ends up in an infinite loop.

Check your knowledge:

- What causes the infinite looping in the "bad" implementations?
- Would it help if *left-to-right* rule was changed to *right-to-left* rule?
- Or if the *top-to-bottom* rule was *bottom-to-top* rule?

Task 2: Match terms in various forms with their usual meaning.

Syntactic sugar	*Usual meaning*	*Hacker's syntax*
`[]`	list with 1 item	`'.'(X,Y)`
`[X,Y]`	empty list	`'.'(harry,[])`
`[harry]`	list with 2 items	`[]`
`[X|Y]`	list with at least 1 item	`'.'(X,'.'(Y,[]))`

Lesson learned: A list in Prolog has the same structure as in Scheme. Empty list is `[]`, whereas in Scheme there is `nil`. And instead of `cons`, Prolog has the dot `'.'`. In case of doubt, go back to the hacker's syntax (it's more instructive). In Prolog use the syntactic sugar (it's shorter).

Task 3: Define basic predicates which work with lists:

- `empty_list(X)` succeeds whenever `X` is an empty list.
- `list_of_size_one(X)` succeeds iff `X`, whose size is exactly 1.
- `any_list(X)` succeeds iff `X` is any list. Hence `any_list(harry)` must fail, but `any_list([harry])` succeeds.
- `my_member(X, List)` succeeds iff `X` is inside the `List`. `my_member(b, [a,b,c])` must succeed and `my_member(d, [a,b,c])` must fail. If you did your implementation correctly, `my_member(X, [a,b,c])` should give you all 3 correct answers: `X=a; X=b; X=c`.

Task 4: Extend the (correct) implementation of `ancestor` so that in the 3rd argument, you get a list of people that are "between" the `Ancestor` and `Descendant`.

```
?- ancestor(diana, william, X).
X = [].

?- ancestor(elizabeth, william, X).
X = [charles].
```

See the beauty of Prolog! Ask for all grandgrandparents:

```
?- ancestor(GrandGrandParent,Person,[GrandParent,Parent]).
```

Task 6: Define the `my_append(A, B, AB)` predicate. It appends list `B` to the end of list `A` and gives the result in the third argument `AB`.

```
?- my_append([a,b,c], [d,e], L).
L = [a,b,c,d,e].
```

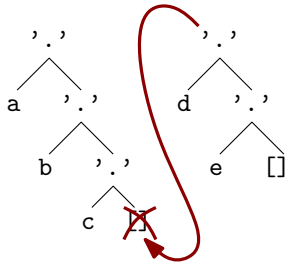
If you did your job correctly, `my_append(L1, L2, [a, b, c])` will reward you with a surprising answer!

Lost? See the hacker's syntax of lists `[a,b,c]` and `[d,e]`:

`'.'(a, '.'(b, '.'(c, []))) and '.'(d, '.'(e, []))`

Look carefully. Now, the secret to `append` is:

> If you remove `[]` in the *first* list and put the second list *instead*, you are done!



The `append` should recurse on the first argument, strip one item after another until it reaches an empty list. While exiting the stack trace, the items are added back, one at a time.

Still lost? What is the result of appending anything to an empty list? Start from `append([],X,?)`...

Task 7: The main procedure behind Prolog is called *unification*. You've used it every time Prolog replaced a variable by a value (try `?- X=a`), but it's much more powerful. For example `[X,Y] = [a,b]` will "extract" values from inside the list.

The somewhat informal definition of unification is:

> Two terms unify if they can be made equal only by substituting variables.

For each of these terms, put a tick ☑ if they will unify or leave ☐ if they don't. If they unify, write down the substitution.

☐ `plus(X,Y) = plus(Z,4)`

☐ `'.'(first,'.'(second,[])) = '.'(A,'.'(B,[]))`

☐ `'.'(first,'.'(second,[])) = '.'(A,B)`

☐ `'.'(first,[]) = '.'(A,'.'(B,[]))`

☐ `'.'(X,'.'(Y,[])) = '.'(Y,'.'(element,[])`

☐ `X = f(X)`

☐ `unify_with_occurs_check(X,f(X))`

Task 8: Define `my_reverse(List, Reversed)` that reverses elements in a list

```
?- my_reverse([1,2,3,4,5], L).
L = [5,4,3,2,1].
```

Don't know how to start? Use `append(List, [X], ListX)` to add an element at the end of a list.

Task 9 (optional): Use an accumulator to reduce the runtime of `my_reverse` from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$.

Task 10 (optional): Extend the `my_member` predicate into the `select` predicate, which gives the "rest" of the list:

```
?- my_select(Elem, [a,b,c], Rest).
Elem = a,
Rest = [b, c] ;
Elem = b,
Rest = [a, c] ;
Elem = c,
Rest = [a, b] ;
false.
```

Task 11 (optional): Define the `descendant` and carefully observe the difference from `ancestor2` and `ancestor3` predicates in task 2.

Task 12 (optional): Rewrite the ancestor predicate so that the `Ancestor` and `Descendant` are included in the list:

```
?- ancestor(X,Y,Z).
X = charles,
Y = harry,
Z = [charles, harry] ;
X = charles,
Y = william,
Z = [charles, william] ;
...
```

Task 13 (optional): Flatten a nested list.

```
?- my_flatten([[a,b],[],[c,[d,e],[f]]],X).
X = [a, b, c, d, e, f] ; ...
```

It's enough to return additional (incorrect) answers. We'll learn how to remove them next week.

```
female(camilla).
female(diana).
female(elizabeth).
female(louise).
female(sophie).

male(charles).
male(edward).
male(george).
male(harry).
male(james).
male(philip).
male(william).

parent(charles,harry).
parent(charles,william).
parent(diana,harry).
parent(diana,william).
parent(edward,james).
parent(edward,louise).
parent(elizabeth,charles).
parent(elizabeth,edward).
parent(george,elizabeth).
parent(philip,charles).
parent(philip,edward).
parent(sophie,james).
parent(sophie,louise).

wife(camilla,charles).
wife(diana,charles).
wife(elizabeth,philip).
wife(sophie,edward).
```