Project 3: Implementing Prolog in Clojure

For this project, we will ask you to implement (a subset of) Prolog in Clojure.

Your implementation should be able to run queries like those we saw in the first Prolog lab (built-in predicates like `forall` etc excepted).

Here is what we expect of you:

- Define a macro named `<-` which can be used to define new rules and facts. For instance:

  ```
  (<- (male george))
  (<- (parent Parent Child) (father Parent Child))
  (<- (parent Parent Child) (mother Parent Child))
  (<- (append (list) T T))
  (<- (append (list H T) L2 (list H TR))
      (append T L2 TR))
  ```

  `<-` should behave like `assertz`: it should add the rule *after* other rules for the same predicate.

  The first clause after `<-` is *the head* (what appears left of `:-` in Prolog), the rest is *the body* (nothing for simple facts).

  Like shown in the `append` example, the head may contain any kind of structure (functors, atoms, ...), not only variables.

- Define a macro named `?-` which runs queries against the knowledge base. It takes a variable number of goal clauses (the effect should be the same as separating the clauses with a comma in Prolog). For instance:

  ```
  (?- (male george))
  (?- (male cecilia))
  (?- (father F anthony))
  (?- (mother cecilia C) (father george C))
  (?- (append (list 1 (list 2 (list))) (list 3 (list 4 (list))) R))
  ```

The answer to those queries should look (respectively) something like this:

```
()            ; worked, no bindings to report
nil           ; unsatisfiable query
([F bob])     ; satisfiable with F = bob
([C howard)]  ; satisfiable with C = bob
([R (list 1 (list 2 (list 3 (list 4 (list))))])
```

This is just an indication, you do not need to respect this exact format, but your implementation needs to clearly report when the query is unsatisfiable, or otherwise the bindings used to satisfy a query. Only top-level bindings must appear: i.e. you must only show the value of variables that actually appear in the query. It is okay to show **unbound** variables as **value** for variables appearing in the query.

If the goal clauses can be satisfied with multiple bindings, you only need to report the first valid binding (you may report more for extra credit).

Nothing else is required (no cut operator, conjunction or disjunction operators or built-in predicates). Of course, we won't complain if these things are present. This assignment is more difficult than the previous ones, so no extensions are necessary to get the maximum grade. Like always, we much prefer a clean solution to the core problem rather than a half-assed frankenstein implementation with lots of features.

Implementation Hints

There are two main approaches to building a small Prolog interpreters. Without giving too much aways, these have to do with how you will deal with scoping. In Prolog, variable names are scoped by rules. You can deal with this either by rewriting your clauses as you search for a solution, or by encoding scopes explicitly.

A good starting point for your implementation would be to implement unification between two values.

Remember that your implementation will need to backtrack, hence your implementations should store data that will enable backtacking later. This is typically done at what is called *choice points*. But what is a choice point? Think for yourself.

Try to build your code without mutation. Clojure encourages this style, and it makes a whole lot of sense to do so when implementing a Prolog interpreter. The only exception to this rule is the database itself. You should store the database in a an `atom` or a `ref` in order to replace it when adding a new rule.

Be mindful of stack space: remember that recursion will consume stack space unless using `recur`, hence the main loop of your implementation should use `recur` or some form of non-recursive looping.

To check if a character is lower/uppercase (to distinguish between atoms and variables), you can use `Character/isUpperCase` and `Character/isLowerCase`.

Tests

To help you, we supply a test file called `tests.clj`. This file requires the `prolog` namespace, so your solution should use that namespace.

Despite its name, the file does not contain any test, but it loads some sample data into the database (namely the family tree from lab 1 and the definition of `append/3`). Feel free to edit this file and use it as you see fit.

Deliverables

Follow these instructions to the letter, or you will LOSE POINTS.

The project deadline is **Thursday the 11th of May, 23:59**.

You must post on Moodle (section: Project 3) the three following files (and **only** those, with exactly these names):

- `prolog.clj` - The implementation of your SQL framework.
- `report.pdf` - A short report, see instructions below.

Warning: We will be intransigeant towards plagiarism. We are well aware of existing solutions, and will cross check student's submissions, both manually and automatically. You may discuss the assignment between yourself, but do not share **any** code. We will find out. We always do.

Report

We ask you to write a short report, which should contain **only** the answers to the following questions (formatted as question then answer, not as a continuous text):

1. How did you deal with scopes, as discussed in the *Implementation Hints* section. Give some details.

2. Explain briefly the core loop of your program. Do not be scared by the appellation *core loop*, which may end up referring to the major part of your program. It's simply the part of the code that is repeatedly called to prove the satisfiability of a goal and supply the appropriate bindings.

   In particular, we are interested in the values that change each iteration. If your core loop is a recursion function, these would the successive values taken by the function's parameters. Explain what these values represent and how they change.

3. Does your program mutate any state, besides the knowledge base? If yes, why? Did you define macros besides `<-` and `?-`? If yes, briefly explain their purpose. If you did neither of these things, leave this section blank.

4. If you implemented any feature besides those we required, explain them here. If you have something nifty to point out about your code, or iff you reached some particularly neat insight while doing this project, you can share it here. It's perfectly fine to leave this section blank.

Interviews

We will see each of you for a small interview on the 17th of May. This will serve as a capstone for the course. Details will follow.

During this interview, you will demonstrate your solution, and we will ask some questions about it.

You may also use this opportunity to ask for feedback about all three projects.