Meta-interpreters in Prolog

`A program fully developed in Prolog`.

1. Introduction

The main difference between **meta-predicates** and **meta-interpreters** is: **meta-predicates** are built-in Prolog tools that influence the standard execution of the program. In contrast, **meta-interpreters** are programs, designed and tested by a developer, that take another program in input and define their own interpretation rules.

Simply, we can develop a new interpreter that follows our **interpretation rules**.

The meta-interpreters use two main meta-predicates:

1. **clause(Head, Body)**: used to detect if it exists a given clause inside the database program.
2. **call(T)**: evaluation of the subject *T* of the **call** meta-predicate.

2. Meta-interpreter

In `Prolog`, as we already know, there is no **difference** between programs and data. From this first assumption and by the set of **built-in predicates**, we could design any kind of **meta-interpreter**.

A meta-interpreter is a specific type of meta-program, that takes in input a program and returns another program. In other words, we can define any given meta-interpreter as: `a program written in a specific language L that is able to execute other programs written in that language L`.

In `Prolog`, we shall say that a meta-interpreter for a language **L** is defined as an interpreter for **L**, but fully written in **Prolog**.

Given these premises, would it be possible to make a **Prolog machine** written in Prolog for interpreting another Prolog program? This query is answered by the **vanilla meta-interpreter**.

3. Vanilla meta-interpreter

The **vanilla meta-interpreter** is a Prolog machine designed in Prolog for executing Prolog programs.

Assuming a predicate `solve(Goal)` (outside of the set of built-in predicates) that answers true if the variable `Goal` can be proved using the clauses of the current program taken in account, a possible vanilla meta-interpreter would be:

```
solve(true) :- !.
solve((A, B)) :- !, solve(A), solve(B).
solve(A) :- clause(A, B), solve(B).
```

This snippet of code has three main passages:

- 1st step. If we pass a **true fact** to the predicate, then all the other possibilities are thrown away from the search tree and the computation stops.
- 2nd step. `(A, B)` is the **conjuction** of different literals. The goal of the predicate is to prove the truth of the conjuction. This clause answers to the question: *which is the resolution order of the conjuction?*. By this example, we have choosen the `left-most` rule.
- 3rd step. Given a rule `A`, the predicate will check if, inside the database program, `A` is already defined as a rule with a body `B`, and then we solve `B` in order to find the truth of the rule.

The last instance provides us the real meaning of the vanilla meta-interpreter: given a `goal`, it will use our Prolog program to prove that `goal`.

But, what is the main advantage of using a `home-made` interpreter? We have already a Prolog interpreter that does the same and also better than us. Vanilla interpreter mimics the same behavior of the Prolog interpreter, with a main difference: vanilla is developed in Prolog, indeed the standard one is written in C, so its compilation time is very fast.

However, we can use the vanilla meta-interpreter **to change** the general behavior of Prolog program language. Let's see some snippets for better understanding.

Right-most rule example

Define a Prolog interpreter that adopts the rule `right-most`.

From the previous snippet, we can achieve the `right-most` rule by changing the order of the resolution process.

```
solve(true) :- !.
solve((A, B)) :- !, solve(B), solve(A).
solve(A) :- clause(A, B), solve(B).
```

Note: the predicate `solve(...)` does not belong to the set of built-in ones.

Number of steps example

Define a prolog interpreter `solve(Goal, Step)` that:

- It is true if Goal can be proved.
- In case Goal is proved, Step is the number of resolution steps used to prove the Goal.

```
solve(true, 0) :- !.
solve((A, B), S) :- !, solve(A, SA), solve(B, SB), S is SA + SB.
solve(A, S) :- clause(A, B), solve(B, SB), S is 1 + SB.
```

As before, we focus on the main passages:

- 1st step. Solving a **true fact** requires 0 steps.

- 2nd step. The total number of steps to solve the conjuction `(A, B)` is equal to the sum of the steps for solving `A` and `B`.
- 3rd step. The amount of steps necessary to solve the rule `A` is equal to the steps to solve the **body** of the rule.

```
a :- b, c.
b :- d.
c.
d.

?- solve(a, Step)
   yes Step = 4
```

For solving the rule `a` are necessary 4 steps. The number is simply computed by all the unifications done: `a :- b, c.` → `a :- d, c.` → `a :- true, c.` → `a :- true, true.`.

4. Expert System

An **expert system** is a construction over the knowledge base that allows us to make queries about the same knowledge base.

Let's suppose we have a knowledge base, and we want to query it looking to prove something.

```
good_pet(X) :- bird(X), small(X).
good_pet(X) :- cuddly(X), yellow(X).
bird(X) :- has_feathers(X), tweets(X).
yellow(tweety).
```

We want to know if `tweety` is a good pet:

```
?- good_pet(tweety).
ERROR. Undefined procedure: has_feathers/1
```

`Does it mean that we do not know if tweety has feathers?`
No, since Prolog assumes the **Close World Assumption** (every unstaged information is considered false), `tweety` does not have feathers. However, when we look to the knowledge base we are more to the side of the **Open World Assumption**: the informations not stated are not false but simply unknown. We have to define a way to retrieve them.

The basic idea is to extend our **knowledge base**. If we try to prove a `Goal` using the given knowldge base and we fail, then we could ask more informations to the user. By the way, this approach has two main problems:

- Following this method, our Prolog program will end up asking everything to the user.
- At every new start we lose all the informations already asked to the user.

We need a method that allows us to store the retrieved informations inside a permanent storage.

5. Modifying dynamically the program

Any time a Prolog program is consulted, its representation in terms of data structure is loaded into a database. This database is the **program database** loaded in memory, handled by the common Database Management System techniques.

We can use this representation to update the knowledge base. The main methods to update the knowledge base are:

- 1st `assert(T)`
  The clause `T` is added to the database program in a non-specified position of the data structure.
- 2nd `asserta(T)`
  The clause `T` is added at the beginning of the database.
- 3rd `assertz(T)`
  The clause `T` is added at the end of the database.
- 4th `retract(T)`
  Removes the first clause that matches `T`.
- 5th `abolish(T)`/`retract_all(T)`
  Removes all the occurencies of that clause from the database.

All the clauses `T` must be already instantiated before the executions of the described predicates. We have to pay attention when we use these methods, they can change the **declarative semantic** of the Prolog program.