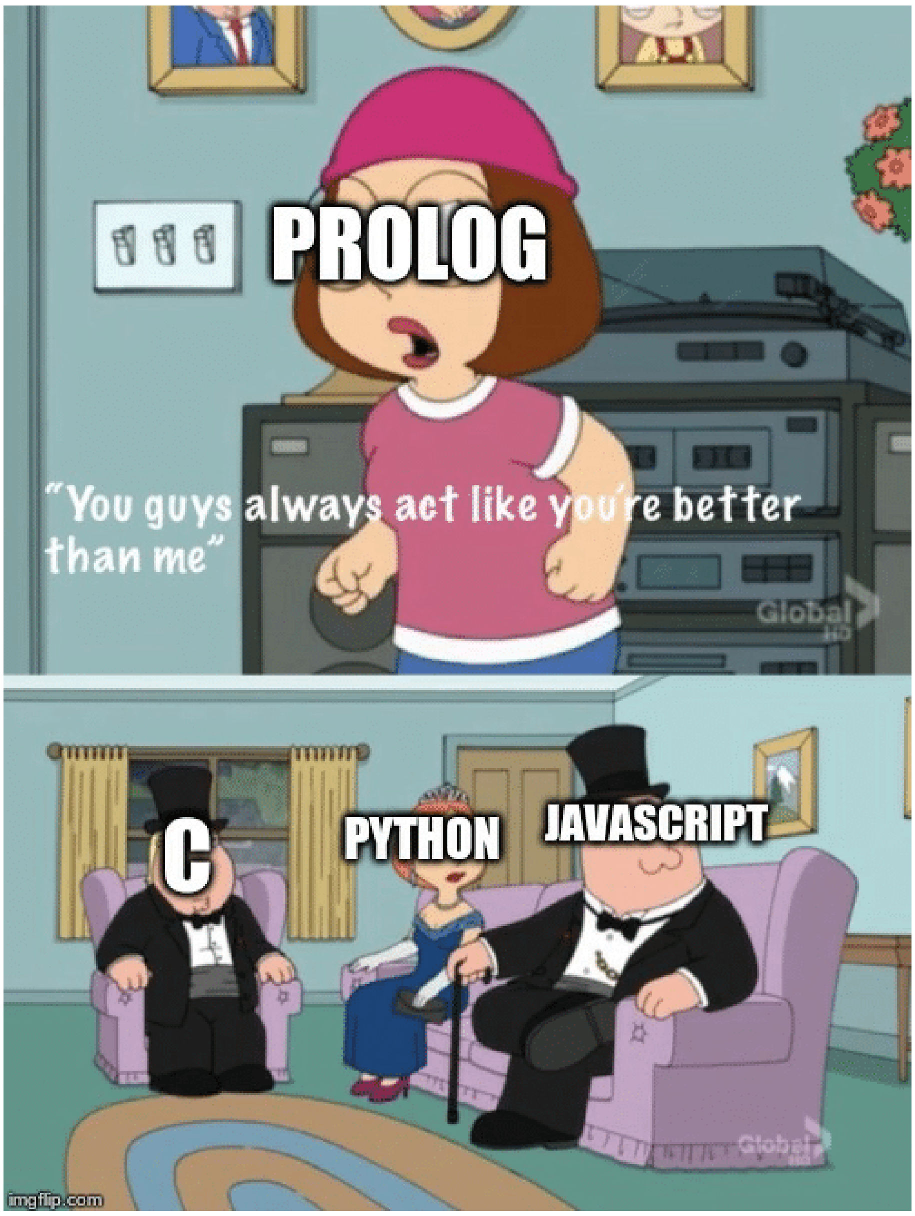
FAIKR Module 2 Q&A



Premise

We wrote this file in order to help study for the Fundamentals of AI and Knowledge Representation Module 2 Exam of Professor Chesani.

Good luck :)

1 Prolog

Question 1

Introduce the vanilla meta-interpreter, and to explain its clauses.

The Prolog vanilla meta-interpreter is defined as follows:

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

The explaination of each clause if the following:

1. A tautology is a success.
2. To prove a conjunction, we have to prove both atoms.
3. To prove an atom `A`, we look for a clause `A :- B` that has `A` as conclusion and prove its premise `B`.

Question 2

Define a meta-interpreter for the Prolog language, where the selection of the subgoal in the current resolvent is right-most rather than left-most (as it is usually).

The standard resolution strategy in Prolog is left-most selection of subgoals. However, if you want to define a meta-interpreter with a right-most selection strategy, you need to modify the order in which subgoals are selected from the resolvent.

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

Question 3

Define a meta-interpreter for the Prolog language, where whenever a special predicate break/0 is encountered, the meta-interpreter directly solve it by asking the user if she wants to continue.
The evaluation of subgoal `break/0` will succeed only if the user types `"yes"`. To this end, the candidate can use the predicate `read(Term)`, that "Read the next Prolog term from the current input stream and unify it with Term."

We are required to define the behavior of a single predicate. The most straightforward approach is to handle this specific case by adding a clause:

```
solve(true) :- !.
solve( (A, B) ) :- !, solve(A), solve(B).
solve(break) :- print("continue? "), read(I), !, I=yes.
solve(A) :- clause(A, B), solve(B).
```

In case the input is `yes`, the program continues. Otherwise, it fails as the cut prevents from backtracking.

Question 4

Present the notion of the cut "!" operator in the Prolog language. Moreover, the candidate should illustrate its usage through a very short example.

In Prolog, the cut operator "!" is a powerful and sometimes controversial feature. It is used to control the backtracking behavior of the Prolog interpreter. The cut is a goal that, once reached during the execution of a query, prevents the system from backtracking beyond that point. It essentially commits to the choices made before the cut.

A cut can be useful to achieve mutual exclusion. In other words, to represent a conditional branching:

```
if a(X) then b else c
```

a cut can be used as follows:

```
p(X) :- a(X), !, b.
p(X) :- c.
```

If `a(X)` succeeds, other choice points for `p` will be dropped and only `b` will be evaluated. If `a(X)` fails, the second clause will be considered, therefore evaluating `c`.

Question 5

Describe how negation is tackled in Prolog, what is NAF, what is SLDNF, and the issues related with NAF over terms containing unbounded variables.

Prolog proves a negated atom through the negation-as-failure inference rule: an atom $\neg A$ is true iff A is false in finite time.

SLDNF resolution corresponds to the SLD resolution with NAF. When proving a literal L, there are two possibilities:

- $L = A$ is a positive literal. In this case, the standard SLD resolution is applied.
- $L = \neg A$ is a negative literal. In this case, the interpreter attempts to prove A. If it fails, then L is true; if it succeeds, then L is false.

A problem of the implementation of NAF in Prolog is that the behavior of a negation on unbound variables it not always what one could expect. Consider the following snippet:

```
capital(rome) .
region_capital(bologna).
city(X) :- capital(X).
city(X) :- region_capital(X).
```

If we query `\+capital(X), city(X).`, the interpreter will return a failure. This happens because when proving `\+capital(X)`, the interpreter proves `capital(X)` and binds `X` to `rome` which succeeds, therefore failing `\+capital(X)`. Note that `bologna` is not tried as it is not a possible term for `capital/1`.
If we query `city(X), \+capital(X).`, the interpreter will return `X=bologna`. In this case, the variable `X` can be bound to both `rome` and `bologna`. The former results in a failure, the latter succeeds.

Question 6

Introduce the meta-predicate `clause/2`, showing its use by means of a short example program

The `clause(H, B)` meta-predicate is true if it can unify `H` and `B` with an existing clause of the program.
`H` represents the head of a clause and cannot be a variable (but can be a term containing variables).
`B` represents the body of the clause and can be either a term or a variable.

```
p(X) :- q(X)
q(X).

:- clause(q(X), B).
    yes B=true

:- clause(p(X), B).
    yes B=q(X)

:- clause(p(1), q(1)).
    true

:- clause(z(X), B).
    false
```

Question 7

Describe the (operational) semantics of the predicates setof, bagof, and findall. Moreover, the description should be illustrated by some short examples.

Illustrates the meaning of the meta-predicate findall. The candidate is also invited to illustrates the use of the predicate by means of an example.

`bagof/3` `bagof(X, P, L)` unifies `L` with all the instances of `X` that satisfies `P`, with repetitions. If none satisfies `P`, it fails. For instance:

```
p(1).
p(3).

:- bagof(X, p(X), L).
    yes L=[1, 3] X=X
```

Note that variables appearing in `P` but not in `X` are bound at their first occurrence, which might lead to unwanted behaviours.

`setof/3` Same as `bagof/3` but without repetitions.

`findall/3` Same as `bagof/3`, but when none of the instances of `X` satisfy `P`, it succeeds and unifies `L` with an empty list. Moreover, variables appearing in `P` but not in `X` are not bound permanently.

Question 8

Describe the distribution semantics adopted in LPAD, using also a short program to illustrate such semantics.

The distribution semantics of LPAD is based on the concept of worlds. A world is defined when the head of each clause of the program has been selected. More formally, a selection σ is a set of choices (C, θ, i) which denotes that for the clause C the i-th head has been selected (θ is the substitution that will be applied to the clause). The probability of a world is given by the product of the probabilities of selecting each specific head:

$$\mathcal{P}(w) = \mathcal{P}(\sigma) = \prod_{(C,\theta,i)\in\sigma} \mathcal{P}(C, i)$$

For instance, given the following program:

```
burnt(X):0.8 ; null:0.2 :- onfire(X).
burnt(X):0.6 ; null:0.4 :- smell(X).
onfire(pizza).
```

The probability of the world where:

```
burnt(X) :- onfire(X).
null :- smell(X).
```

is $\mathcal{P}(w) = 0.8 \cdot 0.4 = 0.32$.

2 Theory

Question 9

Introduce the notions of close world assumption and open world assumption, and to briefly discuss how Prolog and Description Logics deal with these aspects.

In closed-world assumption, the truth value of a predicate that is not stated in the knowledge base is false. On the other hand, in open-world assumption, what is not stated is considered unknown.

Prolog is based on a subset of the closed-world assumption as first-order logic is undecidable. In fact, Prolog only considers as true the positive atoms and, for the negations, only those that can be proven in finite time (see NAF in question 5).

Description logics is based on the open-world assumption. When reasoning using description logics, a sentence that cannot be inferred is considered unknown and it requires to branch the reasoning process by considering all the possible alternatives of the unknown sentence.

Question 10

Briefly introduce the ALC Description Logics, mentioning the operators that are supported (negation, AND, ALL, EXISTS), and their meaning (possibly with a short example for each operator).

Briefly introduce the Description Logics, and in particular the principal operators of the ALC fragment (AND operator; ALL operator; [EXISTS 1 r] operator; concept complement (negation)).

Description logics are based on concepts (categories), roles (relationships) and constants (objects). Atomic concepts can be used to form complex concepts using concept forming operators such as:

`[ALL r c]` `r` is a role and `c` is a category. Concept of the individuals whose relation `r` (if any) only involves individuals in the category `c`.

For instance, `[ALL :ChildOf Male]` contains all the individuals with only male children and also those without children.

`[EXISTS` n `r]` n is an integer and `r` is a role. Concept of the individuals `r`-related to at least n other individuals.

For instance, `[EXISTS 1 :ChildOf]` contains all the individuals with at least a child.

`[AND` $\texttt{c}_1$ `...` $\texttt{c}_n$`]` $\texttt{c}_1, \dots, \texttt{c}_n$ are concepts. Concept of the individuals that belong to all the concepts $\texttt{c}_1, \dots, \texttt{c}_n$.

Negation The negation of a concept is its complement. (Note that this was not covered in this year's course).

Question 11

Within the terminological approach towards the representation of concepts/-categories and individuals/instances, the candidate is invited to illustrate the notions of:

- **Disjointness over a set S of categories ($S = c_1, c_2, \ldots, c_n$, where $c1, \ldots, c_n$ are categories).**
- **Exhaustive Decomposition of a category c into a set S of categories.**
- **Partition of a category c into a set of categories S.**

The candidate is invited to illustrate these notions through a simple example.

Disjointness A set of categories S is disjoint iff each category in S is different from the others:

$$\forall c_i, c_j \in S, c_i \neq c_j : c_i \cap c_j = \varnothing$$

For instance a set of categories $S = \{\texttt{animals}, \texttt{helicopters}\}$ is disjoint.

Exhaustive decomposition A set of categories S is an exhaustive decomposition of a category c iff each object in c belongs to at least a category in S:

$$\forall o \in c : (\exists c_i \in S : o \in c_i)$$

For instance a set of categories $S = \{$ `pizzas with mozzarella`, `pizzas without mozzarella`, `pizzas with tomato`, `pizzas without tomato` $\}$ is an exhaustive decomposition of the category `pizzas`.

Partition A set of categories S is a partition of a category c iff:

- S is disjoint;
- S is an exhaustive decomposition of c.

For instance a set of categories $S = \{$ `pizzas with mozzarella`, `pizzas without mozzarella` $\}$ is a partition of the category `pizzas`.

Question 12

Briefly introduce the notion of Semantic Networks, and to highlight some of the limits that were present in their original formulation.

Semantic networks are a graphical representation of categories and objects. Nodes of the network represent objects and categories. Arcs connecting nodes can represent:

- Properties of an object.
- Properties of a category.
- Relationship between objects.
- Is-a relations.

Semantic networks are limited to a subset of first-order logic and cannot express negations, quantifiers, disjunctions and functions nesting.

Question 13

Briefly introduce the Knowledge Graph paradigms, and which are the main differences w.r.t. the Semantic Web proposal.

Knowledge graphs are based on a simple standardized vocabulary of types and properties. Knowledge is represented through a graph where the nodes are terms and the edges are relations. Important properties are:

- There are only facts and no axioms (A-box and T-box are the same).
- Data does not have a fixed structure.

- Queries can be solved by exploring the graph using traditional graph algorithms.

Differently from the Semantic Web, knowledge graphs are not based on description logics and a have less formal foundation but gains in computational speed.

Question 14

Briefly introduce the three different approaches (presented in the course), to deal with the reasoning with temporal information.

Situation calculus In situation calculus, a situation is a description of what holds in a given moment. A fluent indicates if a property holds in a given situation. Actions are described by mean of axioms:

Possibility axiom An action `a` is possible if its preconditions $\Phi_{\texttt{a}}$ are met:

$$\Phi_{\texttt{a}}(s) \Rightarrow \texttt{poss(a, } s\texttt{)}$$

Successor state axiom A fluent holds in a state if it was made true by an action in the previous state or it was already true and the action at the previous state did not change it. Note that this is subject to the frame problem.

Event calculus Fluents are reified as terms and a fixed set of predicates allows to describe the evolution of the world without the frame problem. For more details see question 15.

Allen's logic of intervals Based on the concept of intervals. An interval has a begin and end time. Temporal operators allow to express properties over intervals (e.g. intervals overlap).

Question 15

Describe the predicates/terminology used in the definition of the Event Calculus Framework.

Introduce the terminology used in the Event Calculus Framework having care, in particular, of indicating the framework axioms.

Introduce the ontology and the domain-independent axioms of the Event Calculus framework.

Event calculus is based on the concepts of fluents (properties that holds in the worlds) and events (actions that modify the world). It has the following primitive predicates:

`holdsAt(F, T)` The fluent `F` holds at time `T`.

`happens(E, T)` The event `E` is (was) executed at time `T`.

`initiates(E, F, T)` The event `E` makes the fluent `F` to start holding at time `T`.

`terminates(E, F, T)` The event `E` makes the fluent `F` to stop holding at time `T`.

$\texttt{clipped(T}_i\texttt{, F, T}_j\texttt{)}$ The fluent `F` stops holding between times $\texttt{T}_i$ and $\texttt{T}_j$.

`initially(F)` The fluent `F` holds at time $\texttt{T}_0$.

Moreover, the following axioms hold:

$$\texttt{holdsAt(F, T}_j\texttt{)} \Leftarrow \texttt{happens(E, T}_i\texttt{)} \land (\texttt{T}_i < \texttt{T}_j) \land \texttt{initiates(E, F, T}_i\texttt{)} \land \neg\texttt{clipped(T}_i\texttt{, F, T}_j\texttt{)}$$

$$\texttt{holdsAt(F, T}_j\texttt{)} \Leftarrow \texttt{initially(F)} \land \neg\texttt{clipped(T}_0\texttt{, F, T}_j\texttt{)}$$

$$\texttt{clipped(T}_i\texttt{, F, T}_j\texttt{)} \Leftarrow \texttt{happens(E, T}_k\texttt{)} \land (\texttt{T}_i < \texttt{T}_k < \texttt{T}_j) \land \texttt{terminates(E, F, T}_k\texttt{)}$$

Question 16

Explain Forward reasoning in rule-based systems, and highlight the difference w.r.t. backward reasoning.

A rule-based system contains a set of rules in the form of logical implications and facts that can make the rules true. When a new fact is added to the knowledge base, a rule-based system runs the following steps until quiescence:

Matching The rules triggered by the new fact are determined and added to an agenda.

Conflict resolution Determine which rules to execute in the agenda, considering possible conflicts.

Execution The selected rules are executed and the resulting new facts are added to the knowledge base.

Differently from forward reasoning, in backward reasoning the result is obtained starting from the goal by searching for a proof that finds the facts that make the conclusion true. If new facts are added to the knowledge base, the reasoning mechanism need to be restarted as it is unable to dynamically consider changes.

Question 17

Introduce the RETE algorithm, and to show with a short example in natural language the differences between inter-elements and intra-elements patterns/features.

RETE is an efficient algorithm for rule-based systems that allows to associate to each rule the facts that matches its left-hand-side. It is based on the concepts of:

Intra-element features Features that can be checked within the fact. For example, given the rule "when a pizza is ready, then . . . ", the check that the pizza is ready can be done directly on the fact.

Inter-element features Features that involve more facts. For example, given the rule "when the pizza for table X is ready and the tagliatelle for table X is ready, then . . . ", to check the table X it is required to involve two facts.

To match the rules, RETE compiles the left-hand-side of the rules into different networks:

Alpha-networks capture intra-element features and save the results into alpha-memories that can be used by beta-networks.

Beta-networks capture inter-element features and save the results into beta-memories, which corresponds to the conflict set.