Solved exams

The candidate is invited to present the Prolog “vanilla” meta-interpreter, and to shortly comment (in natural language) the meaning of the clauses.After that, the candidate is invited to write a meta-interpreter `solve(Goal, ListOfSubGoals)` that is evaluated to true if `Goal` can be proved; moreover, in the parameter `ListOfSubGoals` the meta interpreter will return the list of the subgoals used to prove the `Goal`. For example, given the program:

```
p(X) :-q(X), r(X).
p(X) :-s(X).
q(X) :-t(X).
r(1).
r(2).
r(3).
t(1).
t(2).
s(12).
```

and the query

```
?-solve(p(X), Result)
```

the following outcomes are expected

```
X = 1,Result = [p(1), q(1), t(1), r(1)] ;
X = 2,Result = [p(2), q(2), t(2), r(2)] ;
X = 12,Result = [p(12), s(12)].
```

Solution

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

solve(true, []):-!.
solve((A,B), L):-!, solve(A,LA), solve(B,LB), append(LA,LB,L).
solve(A,[A|LB]):- clause(A,B), solve(B,LB).
```

14/06/2021

Question 1

Given the following LPAD program:

```
win_one_match(Team) :-win_over(Team, _).

win_over(scotland, czechRep):0.3.
win_over(scotland, holland):0.2.
```

Which is the probability of the query `win_one_match(scotland)`?The candidate should illustrate the Distribution Semantics by showing the "worlds", the probability of each world, and the formula used to compute the overall probability.

We first want to ground the program, obtaining `win_one_match(scotland) :- win_over(Scotland,_)`.

Then, we want to generate the worlds. Remember that the `none` possibilities are implicit.

This means that we have 4 worlds:

```
win_over(scotland, czechrep):0.3
win_over(scotland, holland):0.2.
```

```
none:0.7
win_over(scotland, holland):0.2.
```

```
win_over(scotland, czechrep):0.3
none:0.8.
```

```
none:0.7.
none:0.8.
```

We'll have to sum the probabilities of the worlds in which `win_one_match(scotland)` is true, namely world 1,2, and 3. These are computed as the product of the single choices' probabilities:

```
(0.3*0.2)+(0.2*0.7)+(0.3*0.8) = 0.06+0,14+0,24 = 0,44 = 44%
```

2) *The candidate is invited to briefly introduce the Knowledge Graph paradigms, and which are the main differences w.r.t. the Semantic Web proposal.*

The Semantic Web was a first proposal that achieved the possibility of using the web as a large Knowledge Base, being able to reason upon it. It was composed of a stack of multiple technologies and standards. Knowledge graphs emerged later, as a response to a major problem in the semantic web: reasoning with it is fairly slow, and cannot be used to answer specific queries in almost-real time. Knowledge graphs are a more *relaxed* way of facing the problem, in which the data are represented in **graphs** which we can traverse using graph algorithms, which are way faster. They represent heterogeneous information, which can be encoded as triples representing two vertices and a relation.

28/06/2021

1) The candidate is invited to write a prolog predicate `count/2` that, given as input a term representing a predicate, it will return the number of solutions for the goal of proving that input predicate. For simplicity, it will be assumed that the proof procedure of the predicate passed as parameter will always terminate in a finite time.For example:

```
p(a).
p(b).
p(c).
```

and the query:`count(p(X), Result)`. The expected answer is:`Yes, Result = 3`.This is because there are three different ways for proving the goal `p(X)`.

```
caunt(A,N):-findall(_,A,L), length(L,N).
```

2) The candidate is invited to introduce the RETE algorithm, and to show with a short example in natural language the differences between inter-elements and intra-elements patterns/features.

The RETE algorithm is, as defined by its creators, a *many patterns-many objects* matching algorithm. It is concerned with the first, most important, part of forward reasoning: the matching step. What happens is that we have some facts defined in our KB, and we want to know when rules match them. The production rules paradigm imposes that we'll have to be able to reason on the insertion of new facts without having to recompute everything, every time. RETE solves this: if keeps track of the facts that match (even partially) a LHS using a network of nodes, composed of **alpha** and **beta** nodes that, respectively, test inter-element features and intra-element features. Alpha nodes can test various features of the fact, first of all the **Kind** of the fact. Beta nodes can merge two alpha "streams" to test inter-element features. A beta node takes exactly two inputs, namely two alpha nodes or a beta node and an alpha node or two beta nodes. The results of this matching are kept in the alpha and beta networks. This sacrifices memory for speed.

For example, if we had a rule like *when a House is added, and the owner is named Montali, send an email to the Agenzia delle Entrate*, we'd have two alpha nodes testing the kind of the fact, namely *Owner* and *House*, then we'd have an alpha node checking the name of the Owner, and a beta node (connecting these two preceding alpha nodes) checking that the owner of the house is in fact the one we're looking at. This beta node will then output the conflict set, namely the set of matched RHS, in which we'll found the email operations.

Some questions I expect

Present the Complex Event Processing paradigm and Drools Fusion

For several reasons, and particularly, for the incredible increase in the quantity of available data linked to the spread of IoT devices, the necessity of event processing tools is increasing. To introduce what we mean by event processing, we first have to define what an **event** is: it's simply a *fact*, composed of a description of the event together with some notion of time. This is just a **timestamp** in the case of an instantaneous event, or a timestamp together with a duration for durative events (in fact, an instantaneous event just has duration equal to 0). Complex event processing consists in the computation of **complex events**, namely events that are not generated by the source itself, but by the reasoning upon **simple events**. For example, if we were working on the ECU of a car, a simple event would be the reading from a single wheel's speed sensor, while a complex event would be the *the car is oversteering* computed by checking the difference of speeds of the single wheels.

Drools, in order to be a complete Business Process tool, provides a Complex Event Processing framework named **Drools Fusion**, in which the facts we already know have been completed with the notion of time. We can distinguish two operational modes: the stream mode, in which the system has a clock and is able to keep track of the sync of events, and cloud mode, which doesn't. Operations like sliding windows are not available in the latter. Drools supports the autonomous discard of events, which helps avoiding the memory cluttering that is expected when dealing with such huge amount of data. It can either be implicit or explicit, but the best choice might probably be using both the strategies mixed up. All the Allen operators (which are used to reason upon time of events) and their negations are implemented. For example, using these we could check if an event happened less than 2 minutes after another one, if it happened during the other one, just before...

The downside to CEP is that it misses an explicit declaration of **state**, which is the basis of **Event Calculus**. We can still implement a kind of state in Drools by defining UpEvents and DownEvents for each fluent.

Event Calculus

Event Calculus is a paradigm proposed by Sergot and Kowalski (in fact, it is also known as Kowalski notation) which is mainly based on the notion of a **state** composed by fluents, which is modified by events. Basically, we can completely define event calculus with its ontology (basically, the meanings of the predicates we use), some domain-independent axioms (for example, the notion that a fluent is *clipped* if something happens after it was made true, making it false) and some domain-dependent axioms (that we use to actually describe the problem).

Basically, we define which fluents are true at the beginning with *Initially*, then describe what events cause with *Initiates*, meaning that an event makes a fluent true, and *Terminates*, meaning an event makes a fluent false. Then, we have the *Happens* predicate telling that an event happens at a given time T, and finally the *HoldsAt* predicate which indicates that a fluent is true at a given time T.

As afore-mentioned, domain-independent axioms are crucial to the language. For example, the `HoldsAt` predicate, is defined if an event `Initiates` a fluent at a time which has already passed, and if nothing clipped the fluent between t and now:

$$HoldsAt(f,t) \leftarrow Initiates(e,f,t) \land (t < t_{now}) \land Happens(e,t) \land \neg clipped(f,t,t_{now})$$

or, more simply, if it was true from the beginning:

$$HoldsAt(f,t) \leftarrow Initially(f) \land \neg clipped(f,0,t_{now})$$

Event calculus has the downside that it cannot safely be implemented in Prolog, as there's a negation in the *Clipped* domain-independent axioms which is definitely not safe in SLDNAF.

It then allows deductive reasoning only, which is not ideal for our use cases.

Present Drools Expert

Drools Expert is a forward-reasoning engine that was born with Business Process Management in mind. This area researches how businesses take **decisions**, which we can often describe as simple **rules** having some premises and some consequences. For example, in a pizza shop, some rules might be *if we're low on mozzarella go shopping*, or *if a customer wants a sausage pizza, upsell a cold beer*.

Drools expert is based on Java, sharing lots of syntax with it, and follows the **Production Rules** approach, which is concerned with the possibility of dinamically adding facts to a knowledge base without having to recompute everything as we'd do in Prolog. It is based on a descendant of RETE, Phreak, and presents rules defined as

```
rule "name"
when
//something
then
//something
end
```

We can test the kind of fact simply using the *object* notation, like `Customer()` and its attributes in the parenthesis. We can even introduce variables in the rule with the $ notation. For example, the beer rule we mentioned before would be encoded as follows:

```
rule "Beer upsell"
when
$c1 : Customer()
$p : Pizza(additions=="sausage")
Order(orderedBy == $c1, object == $p)
then
SpamText toSend = new SpamText("Want a beer too?");
send(toSend);
end
```

There are some useful quantifiers too like `exists, not, forall` and possibility of side effects in the KB through insertion, deletion and update of facts. Drools manages the conflict resolution aspects too by permitting the insertion of priorities (salience) and more complex operators.

Talk about LPAD

While logic programming is an extraordinary tool for "pure predicate reasoning" we'd sometimes like to be able to handle **uncertainty**, in the form of probabilities. Probabilistic Logic Programming is the *science* concerned with this: logic programming enhanced with probabilities.

As with lots of other tasks, we can use Prolog and a specific library: LPAD. Simply, what happens is that we enhance the heads of the clauses with probabilities stating how possible each disjunct is. We could use it with rules (having body and head) or just atomic formulas. For example, if we wanted to state that motorcyclists have 30% chance of falling if they have a flat tire, and wheelies are definitely dangerous, we could do the following:

```
fall(X):0.4 :- flat_tire(X)
fall(X):0.7 :- wheelies(X)
```

Note that the *doesn't fall* predicate is implicit, having probability `1-0.4` and `1-0.7`.

Then, when we query the KB, the formulas are first grounded: if I queried `fall(Simone)`, the `X` is unified with `Simone`, then the **atomic choices** (i.e. choose a single possibility for a head) are done until we have generated all the **possible worlds**. Then, each world has probability equal to the product of the choices' probabilities, and we sum the probabilities of the worlds in which the query is true.

In other words, if Simone likes to do wheelies and has a flat tire, we'll have 4 worlds:

```
fall(simone) :- flat_tire(simone)
fall(simone) :- wheelies(simone)

null :- flat_tire(simone)
fall(simone) :- wheelies(simone)

fall(simone) :- flat_tire(simone)
null :- wheelies(simone)

null :- flat_tire(simone)
null :- wheelies(simone)
```

Respectively having probabilities $0.4 * 0.7$, $0.6 * 0.7$, $0.4 * 0.3$, $0.6 * 0.3$.

The probability of me falling is then $0.28 + 0.42 + 0.12 = 0.82$, definitely not worth the motorcycle ride.