Fundamental of Artificial Intelligence and Knowledge Representation, Module 4

Ludovico Granata

2020/2021

Contents

1 Introduction

This part is about a part of the classical artificial intelligence that is not related to machine learning. We will reason using rule, that is a form of reasoning very common between human and so is a very easy way for us to present and exploit knowledge. In the last 10 years we have seen the explosion of machine learning with deep learning but currently in the last 5 years the rule based approach has been re-discovered, not for the reasoning part but for the knowledge representation, because they provide simple and useful model. Indeed today one big problem of deep learning is explaining the results, while with a rule based approach we do not have this problem.

2 Prolog - A (not so) quick recall

Prolog can be both a interpreted or compiled programming language.

2.1 Terminology

A Prolog program is a set of definite clauses of the form:

```
Fact A.
Rule A :- B1,B2,...,Bn
Goal  :-B1,B2,...,Bn
```

Where A and Bi are atomic formulas :

- A is the head of the clause
- B1,B2,...,Bn is the body of the clause
- ”,” is for conjunction
- ”:-” stands for the logical implication, where A is the consequent, and B1,B2,...,Bn is the antecedent

An atomic formula has the form:

```
p(t1,t2,...,tn)
```

where p is a predicate symbol (alphanumeric string starting with low-capital letter). A predicate is everything for which we can say it is true or false (eg. we can't say that ”federico” is true or false but we can say if ”federico is awesome” is true or false).The object of our predicate (t1,t2,...,tn) are terms.
A term is recursively defined as :

- Constants (integer/floating numbers, alphanumeric strings starting with a **small letter**) are terms
- Variables (alphanumeric strings starting with a **capital letter** or starting with the symbol ”_”) are terms
- f(t1,t2,...,tk) is a term if ”f” is a function symbol with k arguments and t1,t2,...,tk are terms. f(t1,t2,...,tk) is also known as structure. Constants can be viewed as functions with zero arguments (arity zero).

NOTICE that there is no distinction between constants, function symbols and predicate symbols. For example in Java there are some keywords that are reserved (Class, etc...), while in Prolog there isn't this distinction. The only distinction introduced is syntactically which is the fact that variable always start with a capital letter while constant with small letter. This has some consequences that are quite huge for artificial intelligence, for example I can interpret the same string in different ways (constant, or maybe a predicate) so it can be interpret as data or as a program. This makes Prolog unique.

2.2 Declarative Interpretation

Variables within a clause are universally quantified

```
grandfather(X,Y) :- father (X,Z) , father (Z,Y)
```

means : ”for each X,Y, X is the grandfather of Y if it exists Z so that X is father of Z and Z is father of Y.

2.3 Execution of a Prolog program

A computation is the attempt to prove, through resolution, that a formula logically follow from the program. The resolution adopted by Logic Programming has two non-determinism:

- The rule of computation
- The search strategy

Prolog adopts SLD (Selective Linear Definite clause resolution) with the following choices:

- Rule of computation: "left-most"
- Search strategy : "depth-first" with chronological backtracking

Depth-first can bring you to loops.

2.4 Multiple solutions, and the disjunction

There may exist multiple answer for a query, we can get them by forcing a failure: backtracking should be started then , looking for the next solution. Practically means to ask the procedure to explore the remaining part of the SLD tree. In prolog standart we can ask for more solutions through the operator ";" :
eg.

```
:- sister (maria, W).
yes W=giovanni;
W=anna;
no
```

Operator ";" can be interpreted as:

- A disjunction operator, that looks for alternative solutions
- Whithin a Prolog program, to express the disjunction

2.5 Procedural interpretation of Prolog

How to understand Prolog when our background is on procedural languages (Python, C, Java...) ?
In Prolog a procedure (The method of a Object Oriented Programming language) is written down as a set of clauses whose:

- Heads have the same predicate symbol
- Predicate symbols have the same arity

The idea is that the arguments appearing in the head of the procedure are the formal parameters

```
:- p(t1,t2,...,tn)
```

so this can be viewed as the call to procedure p. Arguments of p (t1,t2,...,tn) are the actual parameters. So the unification mechanism of Prolog can be viewed as the mechanism for passing the parameters. In Prolog there's no distinction between input parameters and output parameters.
So the body of the clause then can be viewed as a sequence of calls to procedures. Two clauses with the same head are two alternative definitions of the body of a procedure, so we can have overloading of a method.
The only difference with a procedural language is that in Prolog all the variables are "single assignment", meaning that they assume a single value along the proof (except for backtracking), once we assign a value to a variable we can't change it anymore.

2.5.1 Example 1

Write a program that prints the sentence : "Federico is awesome"
In **Java** it would be:

```
void p(){
  System.out.println("Federico is awesome!");
  return;
}
```

In **Prolog** it would be:

```
p :- print ('Federico is awesome!')
```

2.5.2 Example 2

Write a program that takes in input a string and print it out
In **Java** it would be:

```
void p(String message){
  System.out.println(message);
}
```

In **Prolog** it would be:

```
p(Message) :-
    print(Message).
```

Notice that we put the capital letter for Message

2.5.3 Example 3

Write a program that takes in input two integer and a String and prints out the String and the sum of integers In **Java** it would be:

```
int sum (String message, int a , int b){
  int result;
    result = a+b;
    System.out.println( message + ":"+ result);
    return result;
}
```

In **Prolog** it would be:

```
sum(Message, A, B, Result) :-
    Result is A + B,
        print (Message), print (': '), print (Result).
```

Notice that while in Java you can return just one parameter, in Prolog you can return as much parameters as you like

2.6 Iteration and recursion in Prolog

In Prolog there is no iteration (while, for,repeat...) but we can get iterative behaviour through recursion. A function f is tail-recursive if f is the most external call in the recursive definition (valid for every programming language). In other words, if the result of the recursive call is not subject of any other call. Tail-recursion is equivalent to iterations. Every type of iteration can be written in a recursive way and every recursive function can be written as a tail-recursive function. Usually recursion is more computationally expensive then iteration but not in the case of tail recursion, in this case they have the same computational costs (space and time consumption).

example of a tail-recursive function:

```
int doSomethingRecursive (int x){
  if (x<=0){
    return x;
    }
    else {
      return doSomethingRecursive(x-1)
    }
}
```

2.7 Arithmetic in Prolog

In prolog to evaluate an expression we can do with a special pre-defined predicat "is".

```
T is Expr
```

same as

```
is (T, Expr)
```

The expression **Expr** is evaluated and the result is unified with T.
Variables in Expr **must** be completerly instantiated at the moment of evaluation

2.8 Some insight on the execution process

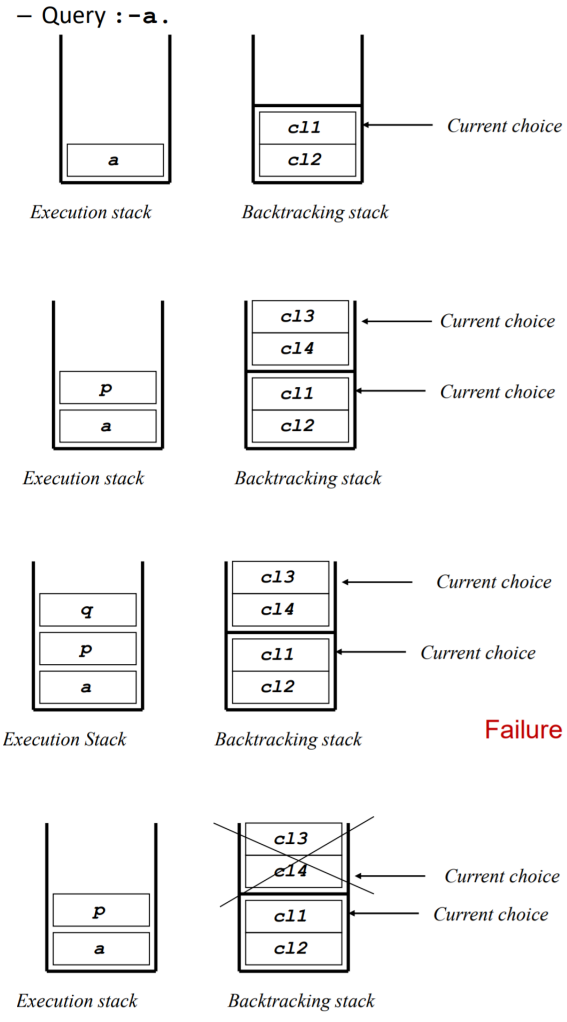
The execution process is build upon (at least) two stacks:

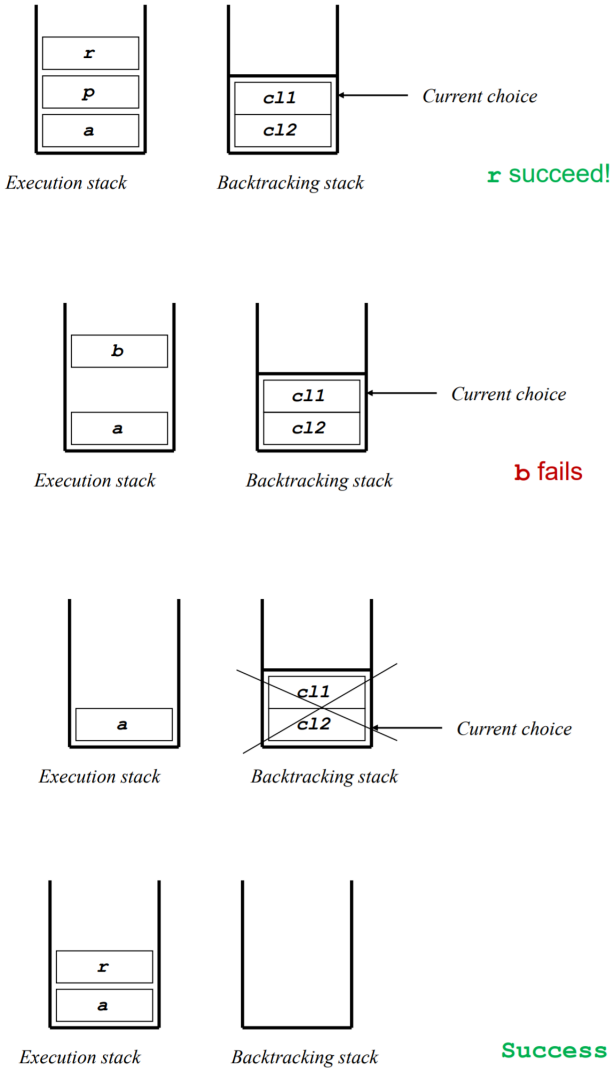
- **Execution stack** : it contains the activation records of the procedures/predicates

• **Backtracking stack** : it contains the set of open choice points

Example

```
(cl1)      a :- p,b.
(cl2)      a :- r.
(cl3)      p :- q.
(cl4)      p :- r.
(cl5)      r.
```





2.9 The CUT

In Prolog there are a number of pre-defined predicates that allows to interfere and control the execution process of a goal and the **cut** ("!") is among them.
The cut **has no logical meaning** and **no declarative semantics**, is very bad and low level. Unfortunately it heavily affects the execution process and it is very useful to write fast Prolog programs.
The evaluation of the CUT is to make some choices as definitive and non-backtrackable, in other words, some block are removed from the backtracking stack. The CUT alters the control of the program and declarativeness is com-

pletely lost.
Consider the clause:

```
p :- q1,q2,...,qi, !, qi1,qi2,...,qn
```

The evaluation of ! always succeeds and it is ignored in backtracking. All the choices made in the evaluation of goals q1,q2,...,qi and goal p are made definitive; in other words, the choice points are removed from the backtracking stack. The choice points of the goals placed after the cut (qi1,qi2,...,qn) are not touched or modified. If the evaluation of qi1,qi2,...,qn fails, then the "whole" p fails, even if there were other alternatives for p, these would have been removed by the cut. **In other words**: all the goals (including the head p) before the cut will not backtrack, while the other goal can backtrack.

Example 1

```
a(X,Y) :- b(X), !, c(Y).
a(0,0).
b(1).
b(2).
c(1).
c(2).

:- a(X,Y).
 yes  X=1 Y=1;
      X=1 Y=2;
 no
```

Example 2 : The first succeeds while the second fails

```
first :
p(X) :- q(X), r(X).
q(1).
q(2).
r(2).

second :
p(X) :- q(X),!,  r(X).
q(1).
q(2).
r(2).

:-p(X)
```

2.9.1 Use of the CUT to achieve mutual exclusion between two clauses

Suppose we have a condition a(X), used to choose between two different program paths:

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

we can achieve this with the CUT in the following way:

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

what happens is that:

- If a(X) is true, then the cut is evaluated and the choice point for p(X) is removed.
- If a(X) fails, backtracking is started before the cut.

Example: *Write a predicate that receives a list of integers, and returns a new list containing only positive numbers*

```
without the CUT
filter ([],[]).
filter([H|T], [H|Rest]) :- H>0, filter (T,Rest).
filter([H|T], Rest) :- H=<0, filter (T,Rest).

with the CUT
filter ([],[]).
filter([H|T], [H|Rest]) :- H>0, !, filter (T,Rest).
filter([_|T], Rest) :-  filter (T,Rest).
```

2.10 Negation

Prolog allows only definite clauses, no negative literals, moreover SLD does not allow to derive negative information

```
person(mary).
person(john).
person(anna).
dog(fuffy).
```

Intuitively, fuffy is not a person, but it cannot be proved by using the facts in our KB. So we use the **Closed World Assumption**(CWA): we have not written that fuffy is a person so it is not.
CWA is a non-monotonic inference rule: adding new axioms to the program might change the set of theorems that previously held, which is not a good property. Due to FOL undecidability, there is no algorithm that establishes in a finite time if A is not logical consequence of P. Operationally, it happens that if A is not logical consequence of P, SLD resolution is not guaranteed to terminate.
Example

```
city(rome) :- city(rome).
city(bologna).
```

SLD cannot prove, in finite time, that city(rome) is not logical consequence. As consequence use of CWA must be restricted to those atoms whose proof terminates in finite time (those atoms for which SLD does not diverge)

So let's drop the CWA, indeed Prolog adopts a less powerful rule, **The Negation as Failure**. Negation as Failure derives only the negation of atoms whose proof terminates with failure in a finite time.
Given a program P, we name FF(P) the set of atoms for which the proof fails in a finite time
NF rule : $NF(P) = \{\sim A | A \in FF(P)\}$

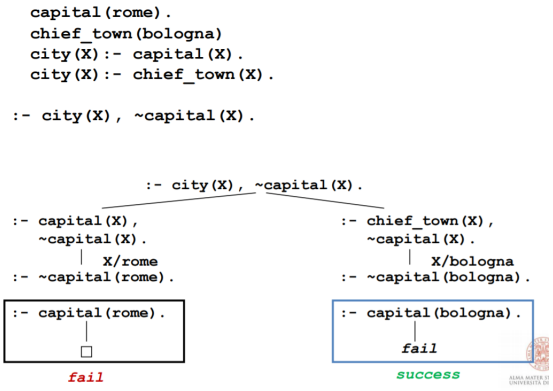
To solve goals containing also negative atoms, SLDNF has been proposed. It extends SLD resolution with Negation as Failure (NF).
A SLDNF step is defined as:

- Do not select any negative literal Li, if it is not "ground";
- If the selected literal Li is positive, then apply a normal SLD step
- If Li is $\sim$ A (with A "ground") and A fails in finite time (it has a finite failure SLD tree), then Li succeeds, and the new resolvent is:

$$:-L1, ..., Li-1, Li+1, ..., Lm$$

A selection rule is safe if it selects a negative literal only when it is ground. The selection of ground negative literals only is needed to ensure correctness and completeness of SLDNF-
SLDNF is used in Prolog to implement the Negation as Failure. To prove $\sim$A, where A is an atom, the Prolog interpreter try to prove A.

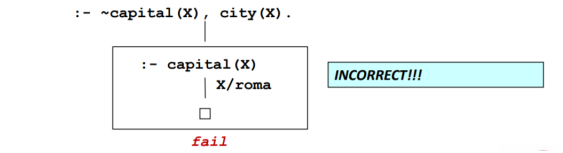
- If the proof for A succeed, then the proof of $\sim$ A fails
- If the proof for A fails in a finite time, then $\sim$A is proved successfully



Prolog does not use a safe selection rule: it selects ALWAYS the left-most literal, without checking if it is ground. Indeed, it is a non-correct implementation of SLDNF.

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

:- ~capital(X), city(X).
```



- The problem lies in the meaning of the quantifiers of variables appearing in negative literals:

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

:- ~capital(X).
```

The intended meaning is: "does exist an X that is not capital?"

F = ∃ **X** **~capital(X).**

Answer: it exists an entity (bologna) that is not a capital.

- Instead, in SLDNF, we are looking a proof for

  :- capital(X).

  with the explicit quantifier:

  F = ∃**X** **capital(X).**

- Then, the result is negated:

  F = ~ (∃**X** **capital(X)).**

  syntactic transformation:

  F = ∀ **X** **(~ capitale(X))**

- Summing up, if there is **X** that is a capital, the proof for F fails.

3 Prolog Meta-Predicates

In Prolog predicates (i.e programs) and terms (i.e data) have the same syntactical structure. As consequence, predicates and terms can be exchanged and exploited in different roles. The same possibility has been achieved by other programming language only in the last years while Prolog do it since the beginning in the '80. For example in Python I can pass as a parameter of a function the code of another function.

3.1 The call predicate

The first predicate we are interested in is the *call* predicate. It has arity one, a term T:

```
call(T)
```

The term T is considered as a atom, and the Prolog interpreter is requested to evaluate it (execute it). Obviously, at the moment of the evaluation, T must be a non-numeric term (in logic a number is not a program).
The Call is considered a meta-predicate because its evaluation directly interferes with the Prolog interpreter underneath, within the same evaluation instance.

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

:- call(q(Y)).

yes Y = a.
```

When executed, Call, asks to the Prolog interpreter of proving q(Y). It can be used also within programs

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

:- p(q(Y)).

yes Y = a.
```

This possibility open up the idea that Prolog was able to support second order logic because you can pass entire proposition in first order logic as a parameter, and having predicate's truth value about the truth value of a logic sentence is something the is linked to second order logic. This path was investigated by many scientists but they didn't bring so many results.
Some Prolog interpreters allow the following notation

```
p(X) :- X.
that means
p(X) :- call (X)
```

X is also said to be a meta-logic variable.

3.1.1 CALL predicate with CUT

Having the Call In conjunction with the CUT we can write the if_then_else construct that behaves in a procedural way:

```
if_then_else (Cond,Goal1,Goal2) :-
      call(Cond), !,
            call(Goal1).
if_then_else (Cond,Goal1,Goal2) :-
      call(Goal2).
```

3.2 Fail predicate

The fail predicate takes no arguments (arity zero), its evaluation **always fails**. So it forces the interpreter to explore other alternatives, in other words, it explicitly activate the backtracking.
Why should we force the failure of a proof?

- To obtain some form of iteration over data

for example: let us consider a KB with facts p/1, apply a predicate q(X) on all X that satisfy p(X)

```
iterate :- call(p(X) ),
           verify (q(X)),
           fail.
           iterate.

verify(q(X)) :- call(q(X)), ! .
```

- To implement the negation as failure

We can implement it as failure through a predicate **not(P)**, that is true if P is not a consequence of the program:

```
not(P) :- call(P), !, fail.
not(P).
```

if call(P) succeed then it will be executed the CUT and then it will fail, if it will not succeed it will backtrack and choose the second not(P) and it will succeed.

```
\+(P)
```

means not(P).
The meaning is not really logical

3.2.1 Fail predicate with CUT

The sequence " *!,fail*" is often used to force a global failure of a predicate p (and not only backtracking). For example, define the fly property, that is true for all the birds except penguins and ostrich, in other words we want to write a KB able to deal with default, but supporting exceptions.

```
fly(X) :- penguin(X), !, fail.
fly(X) :- ostrich(X), !, fail.
...
fly(X) :- bird(X).
```

Checking over all the possible exception might not be feasible, this is one immediate problem of this way of presenting knowledge.

3.3 Predicates setof and bagof

In Prolog the usual query

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

is interpreted with X existentially quantified. As result, it is returned a possible substitution for variables of p such that the query is satisfied. But sometimes it might be interesting to ask queries like **"which is the set S of element X such that p(X) is true?"**
Some Prolog interpreters support this type of second-order queries by providing pre-definite predicates.

- setof(X,P,S).

S is the set of instances X that satisfy the goal P

- bagof(X,P,L).

L is the list of instances X that satisfy the goal P
In both cases, if there are no X satisfying P, the predicates fail.
Which is the difference?
bagof returns a list possibly containing repetitions, while setof should not contain repetitions. But this is not always true, it depends by the implementation.
Example

```
p(1).
p(2).
p(0).
p(1).
q(2).
r(7).

:- setof (X,p(X),S).
  yes   S = [0,1,2]
       X = X

:- bagof (X,p(X),S).
  yes   S=[1,2,0,1]
       X = X
```

Variable X, at the end, has not been unified with any value.
Problem

```
father (giovanni, mario).
father (giovanni, giuseppe).
father (mario, paola).
father (mario, aldo).
father (giuseppe, maria).

:- setof (X, father(X,Y), S).
  yes X=X Y=aldo S=[mario];
          X=X Y=giuseppe S=[giovanni];
            X=X Y=maria S=[giuseppe];
            X=X Y=mario S=[giovanni];
```

```
            X=X Y=paola S=[mario];
     no
```

Here we were expecting all the X for which father(X,Y) is true, but instead it returned those X for which, for the same value of Y, father(X,Y) is true. That is very strange, but indeed the problem is that Y is quantified wrongly, if I want to get the intended meaning I have to quantify it existentially

```

:- setof (X, Y^father(X,Y), S).
  yes [giovanni,mario,giuseppe]
              X=X
              Y=Y

```

or simply I can switch to the predicate **findall**:

```
findall(X,P,S)
```

that is true if S is the list of instance X (without repetitions, but not always true) for which predicate P is true.
so our previous example with findall will be:

```
:- findall (X, father(X,Y), S).

yes S=[giovanni,mario,giuseppe]
               X=X
               Y=Y


  equivalent to
  :- setof (X,Y^father(X,Y), S).
```

3.4 Verifying properties of terms with predicates

- **var(Term)** →true if Term is currently a variable.
- **nonvar(Term)** → true if Term currently is not a free variable.
- **number(Term)** → true if Term is a number.
- **ground(Term)** →true if Term holds no free variables.

3.5 Accessing the structure of a term

Is possible to transform a term into a representation with an operator (not a predicate):

```
Term =.. List
```

List is a list whose head is the functor of Term and the remaining arguments are the arguments of the term. Either side of the predicate may be a variable, but not both.

```
?- foo(hello,X) =.. List.
List = [foo,hello,X]

?- Term =.. [baz,foo(1)].
Term = baz(foo(1))
```

This is useful operator because it allows us to represent programs as lists of facts and so we can simply work with list, we can, for example, change some parameter or whatever transformation we want to do.

3.6 Accessing the clauses of a program

We have said that in Prolog terms and programs have the same syntactic structure, it allow reflection since the beginning (the ability of a language of query, checking the source code during the program execution).
A clause is represented as a term, for example:

```
h.
h :- b1,b2,...,bn

correspond to the terms:
(h,true)
(h,',',(b1,','(b2,','(...',','(bn-1,bn) ...)))
```

So with the predicate **clause**

```
clause(Head, Body)
```

we are able to query the program itself and see indeed there is or not a fact of the type of the program. clause is true if (Head,Body) is unified with a clause stored within the database program. Where Head must be instantiated to a non-numeric term and Body can be a variable or a term describing the body of a clause.
Example

```
p(1).
q(X,a) :- p(X), r(a)
q(2,Y) :- d(Y).

?- clause (p(1), BODY).
    yes BODY=true

?- clause (q(X,Y), BODY).
  yes X=_1 Y=a BODY=p(_1),r(a);
            X=2 Y=_2 BODY=d(_2);
         no

   notice that _1 is a variable
```

3.7 Loading modules and libraries

In Prolog we can use a lot of modules and libraries, and to do so we use a predicate:

```
use_module (library(XXX)).
```

4 Prolog Meta-Interpreters

The starting point is always the same: **In Prolog there is no difference between programs and data** (predicates and terms).
We can ask the Prolog interpreter to provide us the clauses of the program (*clause(Head, Body).*) and we can also ask the interpreter to "execute" a term (*call(T)*).
Meta-interpreters works as meta-programs, that are programs who execute, works, deal with other programs. The idea of meta-interpreters is that input and possibly output, are not simple data but programs.
They are used for **rapid prototyping** of interpreters of symbolic languages. A meta-interpreter for a language L is defined as an interpreter for L, but written in Prolog. For example I can take a piece of code written in Python and make an Interpreter in Prolog for Python, obviously it won't be fast computationally, but it will be useful for prototyping fast.
Fun fact: Also other languages can be used to write an interpreter, like Java, Python and C.
So why is Prolog special? Why is the beautiful prof. Chesani stressing this point so much?
Indeed there is a huge difference because our Prolog program can interfere with the execution of itself at runtime. A program written in C, for example, is not able to interfere with his own execution.
One of the most asked question at the exam is the Vanilla meta-interpreter

4.1 Meta-interpreter for Pure Prolog aka the Vanilla meta-interpreter

Define a predicate **solve(Goal)** that answers true if Goal can be proved using the clauses of the current program:

```
solve(true) :- !.
if we query solve with the Goal = true then Solve(true) is true and
     we do not need to explore any other options.

solve((A,B)):-!,solve(A),solve(B).
(A,B) means A and B, the meaning is that we have to solve A and
    then B, where B possibly can be again a conjunction of other
    things.

solve(A):-clause(A,B),solve(B).
A is not a conjunction but a single atom.
So we look in our program database if there is a clause that
    defines which is the behaviour of head A and body B and then
    trying to solve B.
```

This is called **Vanilla meta-interpreter**. It represent the standard behaviour of the Prolog interpreter.
So why we would be interested in this Vanilla meta-interpreter if it mimics the default behaviour of the Prolog interpreter? Why on the earth we would be interested in redo the same things that the Prolog interpreter already do better and faster?
It's true, the Vanilla meta-interpreter does not provide an advantage, but it is the starting point for constructing many other different meta-interpreter for Prolog with different behaviour.
Notice that the vanilla meta-interpreter explores the current program, searching for the clauses, until it can prove the goal, or it fails. No need to "call" any predicate. The meta-interpreter will never execute any of the clauses defined in my program. So the Vanilla meta-interpreter does not support pre-defined predicate. To deal with pre-defined predicate we could add:

```
solve(A) :- predefined(A),call(A).
```

Now we can define a Prolog interpreter that has different behaviour.

Example
Write a Prolog interpreter that adopts the calculus rule "right most":

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

Example
Write 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 (in case of conjunctions, the number of steps is defined as the sum of the steps needed for each atomic conjunct)

```
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.
```

This can be useful to know how much it cost to solve a problem.

Example
Let us suppose to represent a knowledge base in terms of rules, and for each rule we have also a "certainty" score (between 0 and 100). Define a meta-interpreter **solve(Goal, CF)** , that is true if **Goal** can be proved, with certainty CF. For conjunction, the certainty is the minimum of the certainties of the conjunctions. For rules, the certainty is the product of the certainty of the rule itself times the certainty of the proof of the body

```
solve(true,100):-!.
solve((A,B),CF) :- ! , solve(A,CFA),solve(B,CFB),min(CFA,CFB,CF).
```

```
solve(A,CFA) :- rule(A,B,CF),solve(B,CFB), CFA is ((CFB*CF)/100).

min(A,B,A) :- A<B,!.
min(A,B,B).
```

Used for example to build a program for physicians, because sometime they have a certain degree of confidence about a rule (it's not probability, that would be computed over a sample space). In Python achieving this would be more complicated.

Example

Let us suppose we have a knowledge base, and we want to query it looking to prove/verify 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).
```

In this knowledge base we have the rule that says that tweety has feathers, so folowing the close world assumption, it launch an error:

```
ERROR: Undefined procedure: has_feathers/1
```

Does it means that we do not know if tweety has feather? Does it means that we do not know anything about the concept "having feathers"?
I know for sure that some bird has feathers so I believe that I have to extend my knowledge base.
The idea is that if it fails, the reasoner could also ask help to the user, and this is already pre-configure:

```
prove(true) :- !.
prove((B,Bs)) :- !, prove(B), prove(Bs).
prove(H) :- clause(H,B), prove(B).
prove(H) :- write('Is'), write(H), writeln('true?'),
read(Answer),
Answer = yes.
```

We can add other predicates that limits the type of things that we can ask.

Example

Write a meta-interpreter for the Prolog language that prints out, before and after the execution of a subgoal, the subgoal itself. Example:

```
p(X) :- q(X).
q(1).
q(2).

?- solve(p(X)).
yes X/1
```

```
Solving: p(X_e0)
Selected Rule: p(X_e0) :- q(X_e0)
Solving: q(X_e0)
Selected Rule: q(1) :- true
Solved: true
Solved: q(1)
```

The meta-interpreter will be:

```
solve(true) :-!.
solve((A,B)) :- !,solve(A), solve(B).
solve(A) :- write('Solving: '), write(A), n1, clause(A,B),
write('Selected Rule: '),
write(A), write(':-'),
write(B),n1, solve(B),
write('Solved: '), write(B), n1.
```

In this way I can get an explanation for how the system worked, and nowadays getting an explanation from an AI system is consider quite important.

4.2 Dynamically modifying the program - assert

We can also thing of changing dynamically our program, how? Well the clauses are stored into tables and I can use:

- **assert(T)**

When assert is evaluated, T must be instantiated to a term denoting a clause (either a fact or a rule), T is added to the database program in a non-specified position. In backtracking, assert is ignored.

- **retract(T)**

5 Probabilistic Logic Programs

It would be nice to define rules and attach to this rules some concept of uncertainty. It is very difficult but it is also the most promising way for mixing different approaches in AI. For example probabilistic knowledge can be extracted from observation of the word and data collection and big data querying or machine learning, but we as humans are used to reason using certain mechanism that use a rule base approach, so mixing the two would be very interesting.
There are two major systems available:

- Logic Programs with Annotated Disjunctions - LPAD (we will focus on this in our example)

- ProbLog

The idea of probabilistic logic programming was presented in 1995.

5.1 LPAD

There are several PLP programming languages, but we will use LPAD.
In LPAD, the head of a clause is extended with disjunctions, and each disjunct is annotated with a probability.
For example we sampled that

- in "people with flu", they also "sneeze in 70 % cases";
- in "people with hay fever", they also "sneeze in 80% cases".

```
sneezing(X):0.7 ; null:0.3 :- flu(X).
sneezing(X):0.8 ; null:0.2 :- hay_fever(X)
```

The ";" symbol denotes a disjunction (X is sneezing or null if is suffering the flu). The keyword "null" is reserved, we cannot write it in the Prolog program(the body). Normally we do not write the null.
What if we know something else?
Let's continue with our example, now we know that Bob has a flu and that Bob suffers of hay fever (CRISP knowledge).

```
sneezing(X):0.7 ; null:0.3 :- flu(X).
sneezing(X):0.8 ; null:0.2 :- hay_fever(X)
flu(bob).
hay_fever(bob).
```

So we have CRISP knowledge with probabilistic knowledge.
Will bob sneeze?
We know that if bob has the flu he will sneeze with a probability of 0.7 and if he has hay_fever he will sneeze with a probability of 0.8. But bob has both the flu and hay_fever so what is the probability of sneezing? We will have some joint probability, but first let's see the distribution semantic proposed bu LPAD.

5.1.1 Probability distribution over rule heads

Each rule has a probability distribution over its head, the sum of the disjunct has to be 1. Note that we can write things like:

```
a:0.2; b:0.3;c:0.1 :- d.
```

So here the probability distribution is <0.2, 0.3, 0.1, 1-(0.6)> .

5.1.2 Worlds

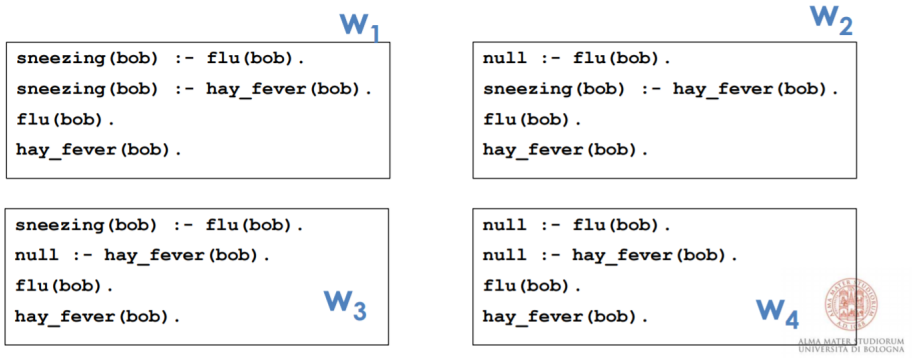
Worlds will be obtained by selecting one atom from the head of every grounding of each clause:

1. Ground the program
2. For each atom in each head, choose to include it or not

for the first point, **grounding** is done by substituting bob to X:

```
sneezing(bob):0.7 ; null:0.3 :- flu(bob).
sneezing(bob):0.8 ; null:0.2 :- hay_fever(bob)
flu(bob).
hay_fever(bob).
```

for the second step we will generate the worlds by choosing for each rule only one of the head disjuncts. Here the first rule has 2 disjunct and also the second so we will obtain 4 worlds.



this is quite expensive from a computational view point, but it is done using fast mechanism so we don't have to worry too much.
The Worlds are just regular logic programs.

5.1.3 Distribution Semantics over worlds

Once we have the worlds we want to know which is the probability distribution over each one of this worlds. So given a clause C and a substitution θ such that $C\theta$ is ground, it is defined:

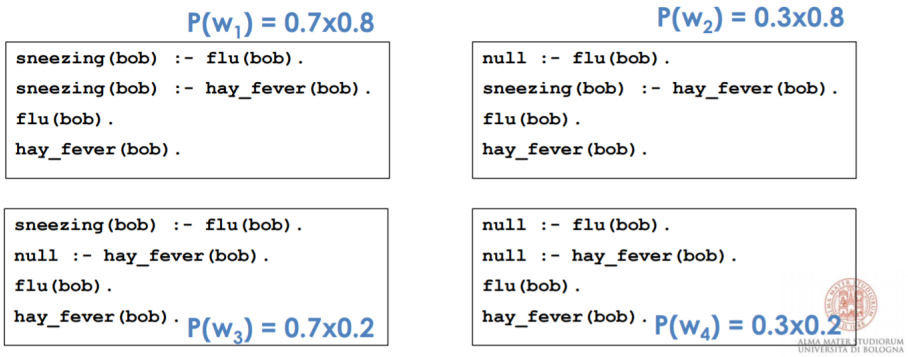
- Atomic choice: selection of the i-th atom of the head of C for grounding $C\theta$: (C,θ,i)
- Composite choice κ : consistent set of atomic choices
- Probability of a composite choice κ :

$$P(\kappa) = \prod_{(C,\theta,i)\in\kappa} P(C,i)$$

- Selection σ: a total composite choice (one atomic choice for every grounding of each clause)
- A selection σ identifies a logic program W_σ called world.
- Probability of a world is then defined as

$$P(W_\sigma) = P(\sigma) = \prod_{(C,\theta,i)\in\kappa} P(C,i)$$

This could appear quite complex but by the practical point of view we can :



5.1.4 Probability of a query

So what is the probability of a query over a world?
Given a ground query Q and a world w:

$$P(Q|w) = \begin{cases} 1 & \text{if Q is true in w} \\ 0 & \text{otherwise} \end{cases}$$

$$P(Q) = \sum_w P(Q,w) = \sum_w P(Q|w)P(w) = \sum_{w|=Q} P(w)$$

So returning to our example we have that sneezing(bob) is true in w_1,w_2,w_3.
So the probability will be:
$P(sneezing(bob)) = 0.7 \times 0.8 + 0.3 \times 0.8 + 0.7 \times 0.2 = 0.94$

6 Rule-based systems and Forward Reasoning

We have spent the previous chapter talking about backward reasoning, showing ways of exploiting rules. But backward reasoning is not the only interesting way, so we will see also forward reasoning. The most influential systems based on rules where indeed based on forward reasoning.
All big ICT vendors provide suites for the business process automation and each suite contains also a rule-based language/engine. Why? The idea is that process managers can directly define (executable) rules, so that larger parts of business process can be automated.

6.1 Rules

The simplest form of a rule is the logical implication:

$$p_1, ..., p_i \rightarrow q_1, ..., q_j$$

$p_1, ..., p_i$ are called the premises, antecedents, the body of the rule or the Left Hand Side (LHS).
$q_1, ..., q_j$ are called the consequences, the consequent, the head of the rule, or the Right Hand Side (RHS)

In Prolog the consequent has just one atom.

6.2 The Production Rules

In previous examples we have seen backward chaining applied to a static knowledge base, the same can be said for several forward chaining approaches:

- The knowledge base is defined before the reasoning step;
- The consequences are derived during the reasoning step;
- The knowledge base is **augmented** with only facts that have been the result of the application of the Modus Ponens rule to facts and rules in the knowledge base.

What if, during the reasoning step, new information/new facts become available?
A naive solution could be to restart the reasoning from scratch but how much it cost? How big is your knowledge base? When it is worthy to restart the reasoning? at every new facts? after 10 new facts...? It highly depend on the specific domain, in some domain you can't apply this.
In the production rules approach, new facts can be dynamically added to the knowledge base, if new facts matches with the left hand side of any rule, the reasoning mechanism is triggered, until it reaches quiescence again.
Production Rule systems allow to explicitly have side effects in the RHS, that are of two type:

- on the external world
- on the knowledge base

About the first type of side effect, being honest, they are allowed also in Prolog, indeed in Prolog you can print out something and this means that you are allowed to have side effects because printing out is indeed side effect.
What is really interesting is that Production Rules can have side effect also on the knowledge base and it is the base of Production Rules systems. The idea is that we have a notion of working memory and our system allow within the

system itself to insert or to delete facts from my working memory and possibly to re-trigger some rules. This can be done also in Prolog with assert and retract but it doesn't have any logical meaning.
So now we are leaving the "logical" setting, heading towards a more "procedural" framework and this raises some questions:

- What if new facts are inconsistent with the existing ones?

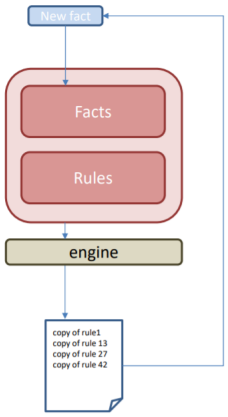
but we are not in a logical setting anymore so we are not interested in deal with logical consistency and is up to us that we are modeling our domain to decide if indeed we are interested to keep logical inconsistency or not.

- What if new facts trigger more rules? which one to start first?

One rules could have side effects on the premises of the other ones, so what we do?

- What if new facts trigger a rule that generates side effects?

Side effects can't be backtrackable
The idea of production rules based systems is a very simple structure after all (we are not interested in keeping any logical meaning).



The pink box is my working memory

1. Match: search for the rules whose LHS matches the fact and decide which ones will trigger
2. Conflict Resolution/Activation: triggered rules are put into the Agenda, and possible conflicts are solved (FIFO, Salience, etc).
3. Execution: RHS are executed, effects are performed, KB is modified, etc.
4. Go to step 1

6.2.1 Working memory

The working memory is a data structure that is the base of Production Rules Systems, it usually contains the current set of facts (initially is just a copy of the KB), the set of rules and depending on the implementation it may contains the set of the (copies of) "partially matched" rules. Since the working memory contains the rule and the facts the performances of PRS might depend on the efficiency of the working memory (WM), in particular on the step of looking into the working memory for those rules that matches with all the facts that have been added to our working memory. The faster algorithm for doing that is currently the RETE algorithm.

6.2.2 The RETE Algorithm

It focuses on the Production Rule System: **Match**. The match step consists on computing which are the rules whose LHS is "matched". Usually the LHS is expressed as a conjunction of patters. Deciding if the LHS of a rule is satisfied/is matched, consists on verifying if all the patterns do have a corresponding element (a fact) in the working memory. It is a "many pattern" (LHS) vs "many objects" (WM) pattern match problem.
Example

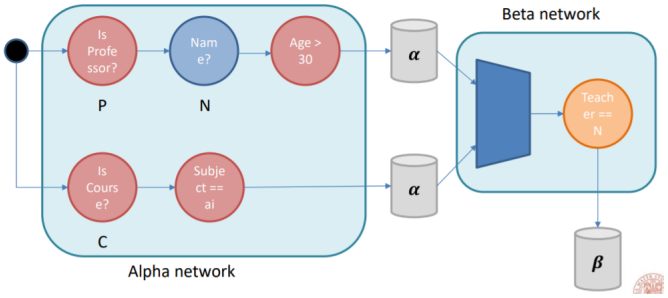
- Rule 1: "when a Person p is added to the WM, and she is a student, send her a greeting message
- Rule 2: "when a Person p is added to the WM, and she is a professor, send her a reminder about the slides.
- Fact: "a Person whose name is Sara, and whose role is student, is added to WM"

The problem is how to determine efficiently which are the rules that should be triggered.
RETE focuses on determining the so called "conflict set",the set of the rules that can be (are ready to be) executed, whose premises are satisfied at that instant of time. In other words the set of all possbile instantiations of production rules such that $\langle$ Rule(RHS), List of elements matched by its LHS $\rangle$, that means that the RHS can be executed given the list of elements matched by its LHS.
The idea of RETE is to avoid iteration over facts by storing, for each pattern, which are the facts that match it. When a new fact is added, all the patterns are confronted, and the list of matches is updated. When a fact is deleted, the patterns are updated as well. When a fact is modified the patterns are updated consequently .
The idea of RETE is to compile a LHS into a network of nodes. There are at least two type of patterns:

1. Patterns testing intra-elements features

2. Patterns testing inter-elements features

Example
When a student whose name is X, when a professor with name X, notify a warning to the deputy head of the school.
student and professor are intra-element features while "whose name is X" and "with name X" are inter-elements features.
Intra elements features are about some test on the new fact (on data inside the new fact), in other words it is about a property that can be solved looking at the fact itself. Inter element features are those condition that put a condition between different patterns, in other words in order to be solved we should look at different facts at the same time. Considering the example, if I have a new fact I just have to check if it is a student or a professor while for the name I have to check more facts (because name X have to be the same for the fact of student and for the fact of professor)
Intra-elements pattern are compiled into alpha-networks, and their outcomes are stored into **alpha-memories**, while Inter-elements are compiled into beta-networks and their outcomes are stored into **beta-memories**.
Example
Suppose we have a rule: "When a Professor P with name N and age>30, when a Course C with subject "ai" and teacher N THEN do something"



In the end of this processing the beta store will contains the conflict set, the list of the rules where the LHS has completely been matched and they are ready to be executed. So once we have computed the beta-memory we will move to the conflict resolution and we will decide for the execution of one of the rules.

6.2.3 ...and the other steps

The RETE algorithm focus on the first step, so for the other step we do not have the best possible solution.

- The **second step** is about conflict resolution, given that many rules can be fired/executed in their RHS, which one will be executed first? In which order will they be executed? Priority? rule order? complexity?

At a certain point the scientific world decide that should choose the rule that is higher related to the application domain that you are dealing with. That means that there isn't a best solution because the best solution varies with the domain.

- The **third step** is about execution, each activated instance of a rule will be executed. By default, all the rules are executed in a cycle. Possible WM modifications will be tackled in another cycle, however, it is often possible to modify this behaviour. This mean that when I insert a fact and the fact matches during the alpha network and in the end it comes up with five different rules that are ready to be triggered, I will order this five rules (following some criteria) and then in my current cycle I will execute all the five rules; this is the default behaviour. Of course I could decide to execute only the first rule and then have a look again to the alpha memory and the beta memory to see if the remaining four rules are still ready to be triggered. What about possible loop? If a consequence of a rule indeed address the working memory we can imagine that is quiet easy to write loops. It's quiet more common to write loops in forward reasoning (eg. DROOLS) then in backward reasoning (eg. Prolog).

6.3 The Drools Framework

It is Java-based, the language is proprietary.

6.3.1 The language

Rules have the structure:

```
rule "Any string as id of the rule"
  //possible rule attributes
when
  // LHS: premises or antecedents
then
  //RHS: consequents or conclusions...side effects
end
```

6.3.2 Left Hand Side - LHS

It is a conjunction of (disjunctions of) patterns. A pattern is the atomic element for describing a conjunct in the LHS. It describes a fact (that could appear in the WM) and also the conditions about the specific fact. Indeed, it is a sort of a filter for the object/facts within the WM.
Example: "When a person is inserted in the system, send him a greeting email"

```
rule "Greeting email"
when
  Person()
then
  sendEmail("greetings")
end
```

sendEmail() will be a method available in our Drool engine, that we will write ourself. Person() refers to an instance of a java class named Person that has been added to our working memory. The idea of the LHS is that it will look for objects that are instances of java class Person.
When the rule above is triggered?

1. If the engine is already running, as soon as a fact Person() is inserted in the working memory, the fact Person will go through the alpha network, it will end up in the alpha memory, it will pass to the beta network, if everything is fine it will come up in the beta memory and taken out and put into the agenda conflict resolution and then executed.
2. At the start of the rule engine, if a fact Person() was already in the working memory (initial setup of the WM)

facts are objects.
Of course this basic pattern alone is poor so it has support to constraints on the object fields (field constraints):

- classical operators, extended to primitive types and String ($==, <, >=, ...$)
- logical connectors (logical and: ',' '&&' ; logical or: '||')

Example

```
rule "Greeting email"
when
  Person(name=="Federico", age>=18)
then
  sendEmail("greetings")
end
```

We can also use variables for having inter-pattern constraints (for example). A variable has a dollar character before the normal name and it is assigned through the ":".
Example

```
rule "Greeting email"
when
  $p : Person($n : name, $n=="Federico", age>=18)
then
  sendEmail($p.getEmailAddr(), "greetings to" + $n)
end
```

What if we want to trigger rules when more facts appear at the same time? We can simply list all the patterns that are considered in logical and, so the rule will trigger only when all the patterns will be satisfied.

```
rule "Married couple"
when
  $p1 : Person($n1 : name)
  $p2 : Person($n2 : name, $p2.marriedWith() == $n1)
then
  ...
end
```

Notice that all the possible combinations are generated by using all the objects that match with the premises. if I put in my working memory two facts:

```
Person ("Federico", marriedWith="Elena")
Person("Elena", marriedWith="Federico")
```

what happens is that my rule will come up in my conflict set twice (the first n1 is unified with federico and the second n1 is unified with elena...)(scope of the variable is all the rule)
The keyword "**this**" can be used in field constraints to refer to the current object/fact:

```
rule "Married couple"
when
  $p1 : Person($n1 : name)
  $p2 : Person($n2 : name, this.marriedWith() == $n1)
then
  ...
end
```

In the LHS we can also use **Quantifiers**:

- exists P(...): there exists at least a fact P(...) in the working memory
- not P(...): the working memory does not contain any fact P(...). This is quite problematic, when is such test performed in the memory?

```
rule "Not P"
when
  not P()
then
  print("hello")
end
```

- forall P(...): trigger the rule if all the instance of P(...) match

Example

```
rule "At least one fine"
when
  $p1 : Person($n1 : name)
  exists Fine(subject == $p1)
then
  System.out.println ($n1+ " was fined at least once");
end
```

What if the person has received more fines? The rule triggers zero or one time.
Example

```
rule "Never fined"
when
  $p1 : Person($n1 : name)
  not Fine( subject ==$p1)
then
  System.out.println ($n1 +"was never fined");
end
```

Example

```
rule "Speed-only fined"
when
  $p1 : Person($n1 : name)
  forall Fine( subject ==$p1, reason="speed")
then
  System.out.println ($n1 +"was fined only for spped");
end
```

The rule triggers zero or all time.
How facts are made?
Java-based Beans, import clause at the beginning of a rule file:

```
import it.unibo.bbs.dss.Person;
```

Directly defined within the rule file, with explicit list of field (internally, mapped as Java Beans)

```
declare Person
  name : String
    age : int
end
```

6.3.3 Right Hand Side - RHS

Consequences can be of two types:

1. "Logic" consequence: they affect the working memory:
   - Insert new facts into the WM (and possibly trigger rules)
   - Retract existing facts
   - Modify existing facts (and possibly re-trigger rules)
2. "Non-Logic" consequences:
   - Any (external) side effect
   - pieces of Java code that will be executed

Example insert

```
declare Logged
  person : Person
rule "log in"
when
  $p: Person()
  $e : LogInEvent(person == $p)
then
  Logged ooo = new Logged();
    ooo.setPerson($p);
    insert(ooo);
end
```

I used the object Logged to keep track of its state, that is not a simple thing to do in AI . The only problem is that the coherence of your working memory is up to you.
Example retract

```
rule "log out"
when
  $ooo: Logged ($p:person)
  $e : LogOutEvent(person == $p)
then
  retract ($ooo);
    // delete ($ooo);
    System.out.println("Goodbye!");
end
```

Side effect will be never retracted (like the writing of "Goodby").
Modify/Update is a combination of retract and insert, applied consecutively:

- update: notifies the engine that an object has changed; changes can happen externally or in the RHS part of the rule.
- modify: takes all the modifications that should be applied, apply them, and notify the engine of the changes

Example When a person logins, update the date of the last login
update:

```
rule "log in"
when
  $p: Person()
  $e : LogInEvent(person == $p)
    $log : LastLogged(person == $p)
then
  $log.setLastLogin(new Date());
    update($log);
end
```

modify:

```
rule "log in"
when
  $p: Person()
  $e : LogInEvent(person == $p)
    $log : LastLogged(person == $p)
then
    modify ($log) {
      setLastLogin(new Date())
    }
end
```

Modify/update operations can be easily subject to loops, to avoid loop we can use the no-loop keyword. The intended meaning is that the rule shouldn't trigger, if the objects are modifies by the rule itself. But if there is a chain of rule that form a loop, then we can't do nothing.

```
rule "log in"
no-loop
when
  $p: Person()
  $e : LogInEvent(person == $p)
    $log : LastLogged(person == $p)
then
  modify ($log) {
      setLastLogin(new Date())
    }
end
```

6.3.4 Conflict resolution

Rules can have the property salience: the higher the value, the higher the priority of their execution.
Example salience

```
declare Logged
  person : Person
end
rule "log in"
  salience 100
when
  $p: Person()
  $e : LogInEvent(person == $p)
then
  Logged ooo = new Logged();
    ooo.setPerson($p);
    insert(ooo);
end
```

Salience is an easy solution to conflict resolution but a naive one. It is difficult to give the right salience score to all the rules.
Another solution to conflict resolution could be to group rules, and then decide (from an external method) to give more importance to a group instead of another

```
declare Logged
  person : Person
end
rule "log in"
  agenda-group "log"
when
  $p: Person()
  $e : LogInEvent(person == $p)
then
  Logged ooo = new Logged();
    ooo.setPerson($p);
    insert(ooo);
end
```

Also Rules can have the property activation-group, for example among all the activated rules of the same group only one of them is executed and the others are discarded.

6.4 DMN - Decision Model and Notation

The DMN is a sort of a use case to show us that forward reasoning is really used. This standard is called decision model and notation.

7 Complex Event Processing and Event Calculus

In the last 20 years we encounter the paradigm of the Internet of things, that is the presence of a huge quantity of sensor, devices that have Internet connection and that can transfer all the data that have been measured. We can have simple devices like a thermometer or complex one like webcams. This huge amount of data is called Big Data, that has some challenges:

- collecting huge amount of information
- storing huge amount of information
- **filter, analyze, and derive new knowledge** from this huge amount of information, **in a timely manner**
- decide when it would be appropriate to delete huge amount of information
- protecting user privacy and rights about this huge amount of information

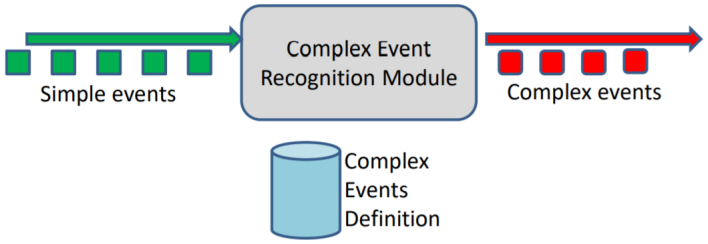
General speaking the research field of dealing with huge quantity of data coming at a time along stream of information is called **Complex Event Processing (CEP)** (older than IoT and Big Data).
Collected information is described in terms of events:

- a description of something, in some (logic?) language
- a temporal information, about when something happened.
- Events can be instantaneous, or can have a duration

Complex Event Processing is about dealing with simple events, reason upon them and derive complex events

- Simple events: events detected by the system underneath (our sensors)
- Complex events: all the events that are generated by the system itself, by aggregating simple events.

Simple and complex do not refer to the information payload, but rather to their ability to provide answers to the application questions. The language used to recognize complex event is called Complex event recognition Language, that is usually based on rules. The Logic-based approach to CEP is called **Event Calculus**.



7.1 CEP and Drools

Drools support CEP through an extension called Drools Fusion. The idea is that events are considered particular facts. Events are characterized by two things: what happened (description, an object) and when (timestamp, number).
Drools Fusion already support **Allen algebra**.
Drools fusion engine on the base of the defined rules, automatically decides the discard of events:

- it will avoids working memory cluttering and all related problems
- a problem is that it requires particular attention from the user side: no writing of rules that would imply to connect events too far in the time line
- It supports aggregation operators
- it support fixed as well as sliding windows (we will see what this means)

There are some base assumption that we have to make:

- We usually required to process a huge volumes of events, but only a small percentage of the events are of real interest.
- Events are usually immutable, since they are a record of a state change.
- Usually the rules and queries on events must run in **reactive** modes, that means react to the detection of event patterns
- Usually there are strong temporal relationships between related events
- Individual events are usually not important. The system is concerned about patterns of related events and their relationships
- Usually, the system is required to perform composition and aggregation of events.

When we have introduce the drools approach for forward reasoning we have discuss the idea of having working memory, alpha network, beta network (etc...). This way of reasoning is usually referred to as being the **Cloud mode**. Here we are not interested in the cloud mode but in the **Stream mode**, the idea is

that indeed we have a stream, a flow of events arriving in our system. In the Stream mode we have two motions:

- Every event has a timestamp
- There is a clock (the idea of). From this notion drools is able to derive the notion of "now". Is up to us deciding which form of clock to use.

Example

```
rule "Sound the alarm"
when
  $f: FireDetected()
  not (SprinklerActivated())
then
  //sound the alarm
end
```

When the "not" will be evaluated?
What I really mean is that i would like to have the evaluation of SprinklerActivated pattern timely linked with the fire detection. So the rule that we have just written is not really clear. We can write the rule in a better way:

```
rule "Sound the alarm"
when
  $f: FireDetected()
  not (SprinklerActivated( this after[0s,10s] $f))
then
  //sound the alarm
end
```

"after" is an Allen operator.

7.1.1 Sliding Windows

It would be nice to write rules about events that happened only the last 24 hours for example, or for the last 100 events only. This request are related to the concept of sliding windows. Sliding Windows are a way to restrict on the event of interest by defining a window that is constantly moving. We have two types of sliding windows:

1. time-based windows
2. length-based windows

ExampleTime-based windows

```
rule "Sound the alarm in case temperature rises above threshold"
when
  TemperatureThreshold( $max:max)
  Number (doubleValue > $max) from accumulate
       (SensorReading ($temp : temperature) over window:time(10m),
```

```
       average ($temp)
then
  //sound the alarm
end
```

Example Length-based windows

```
rule "Sound the alarm in case temperature rises above threshold"
when
  TemperatureThreshold( $max:max)
  Number (doubleValue > $max)
    from accumulate (SensorReading ($temp : temperature) over
    window:length(10), average ($temp)
then
  //sound the alarm
end
```

7.1.2 Events Expiration

When will events expire? There are two ways for determining the expiration:

1. Explicit expiration:

```
declare StockTick
  @expires (30m)
end
```

2. Implicit expiration

```
rule "correlate orders"
when
  $bo: BuyOrderEvent ( $id : id )
  $ae : AckEvent (id == $id, this after[0,10s] $bo )
then
  //do something
end
```

In real application the best way of doing things is using both the approach.

7.1.3 Temporal Reasoning

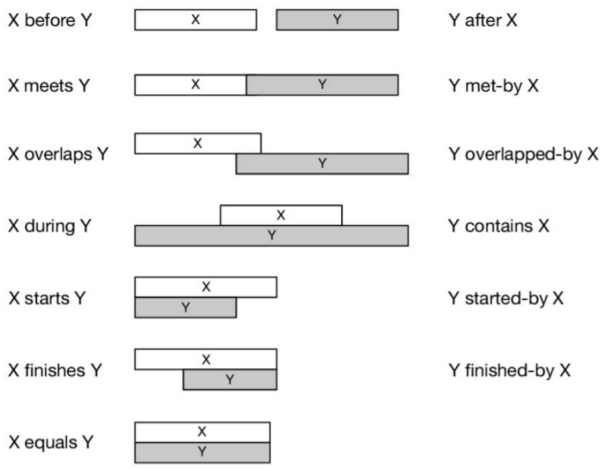
A number of Allen temporal operators are supported:

- After

```
$eventA : EventA (this after [3m30s,4m] $eventB)
meaning:
3m30s <= $eventA.startTimestamp - $eventB.endTimeStamp <= 4m
```

- Before

```
$eventA : EventA (this before[3m30s,4m] $eventB)
meaning:
3m30s <= $event<b.startTimestamp - $eventA.endTimeStamp <= 4m
```



7.2 CEP and Event Calculus

Up to now we have not seen a way in Drools to reason about the state of the system. We could do it using facts to represent the state of the system, it's a fast and easy solution for simple state representation and it works well if only a type of event affects a state property. For more complex domain how can we do it?
We adopt the Event Calculus Framework (Kowalsky):

- More complex solution (then using facts to represent states), but cleaner
- Allow to link multiple different events to the same state property
- State property changes can depend also from other state
- A system is described by a set of state that are currently true in every moment (**fluents**)
- Allows to reason on meta-event of state property change, they are not real event but event related to the transition of a state.

Fluents are properties whose truthness value changes over time.
Event Calculus Framework is composed by three elements:

1. Event calculus "ontology" (fixed)
2. Domain-independent axioms (fixed)
3. Domain-dependent axioms (application dependent) (user, we will work here)

7.2.1 Event Calculus - Ontology

- HoldsAt(F,T): it's a predicate, the fluent F holds at time T
- Happens(E,T): it is a predicate, an E (the fact that an action has been executed) happened at time T
- Initiates(E,F,T): framework define facts, an event E causes fluent F to hold at time T (used in domain-dependent axioms)
- Terminates (E,F,T): event E causes fluent F to cease to hold at time T (used in domain-dependent axioms)
- Clipped(T_1,F,T): Fluent F has been made false between T_1 and T (used in domain-independent axioms)
- Initially (F) : fluent F holds at time 0

The following are the **domain-independent** axioms, it is the most important part. They are quite the core of the solution proposed by Kovalski.

$$HoldsAt(F,T) \Leftarrow Happens(E,T_1) \wedge Initiates(E,F,T_1) \wedge (T_1 < T) \wedge \neg Clipped(T_1,F,T)$$

A fluent holds at timestamp T if there was an event E that happened at timestamp T_1 and this event E indeed initiates fluent at timestamp T_1 and T_1 was before timestamp T and between T_1 and T my fluent was not clipped. In other words it says that a fluent holds if something happened in the past that made it true and nothing disqualified it up to now. example: The light is on if someone switched it on and nobody switched it off up to now. The use of the negation with clipped raises a number of problems.

$$HoldsAt(F,T) \Leftarrow Initially(F) \wedge \neg Clipped(0,F,T)$$

Second alternative for having a fluent true, and it says that a fluent is true if it was initially true at the beginning of my system and no one clipped it from time zero up to timestamp T.

$$Clipped(T_1,F,T_2) \Leftarrow Happens(E,T) \wedge (T_1 < T < T_2) \wedge Terminates(E,F,T)$$

Clipped is true if something happened at timestamp T with T between T_1 and T_2 and this event has the property of terminating my fluent F.
Now let's see the Domain-dependent axioms:
Example light in the office Initially(light_off).
Initiates(push_button, light_on,T) $\Leftarrow$ HoldsAt(light_off, T).
Terminates(push_button, light_off,T) $\Leftarrow$ HoldsAt(light_off,T).
Initiates (push_button, light_off, T) $\Leftarrow$ HoldsAt(light_on,T).
Terminates(push_button,light_on,T) $\Leftarrow$ HoldsAt(light_on,T)

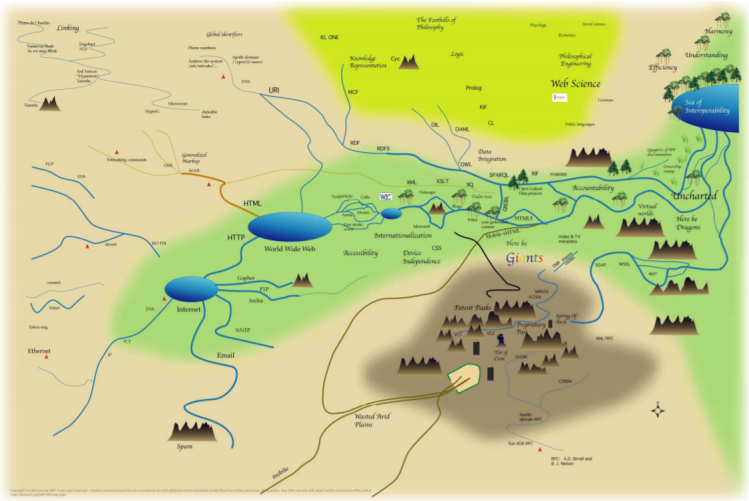
light_on and light_off are two fluent.

There is a huge BUT, the event calculus by Kowalski and Sergot was criticized because it's very easy to understand and it was easily implemented in Prolog but the event calculus is **not safe** if fluents/events contain variables because of the negation in front of the clipping test (negation in Prolog has a number of problems). The second limit is due to the Prolog basic implementation (very easy). It Takes as input the domain-dependent axioms and the set of happened events and provides as output the set of fluents that are true after all the specified events. What if a new happened event is observed? new query is needed and computes from scratch the result, computationally very costly. Event calculus doesn't work in the real world.

7.2.2 CEP and Reactive Event Calculus - Case Study

Finisci

8 Semantic Web and Knowledge Graphs

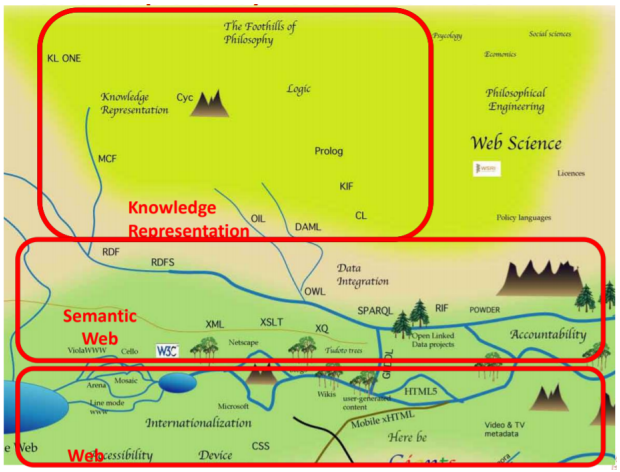


8.1 The Web 1.0

The original Web was a set of information represented by means of:

- Natural Language
- Image, multimedia, graphic rendering/aspect

human users easily exploit all this means for deducting facts from partial information, creating mental associations.



The content is publish on the web with the principal aim of being "human readable" (HTML, CSS...), there is no notion of what is represented but only how. Web pages contain links to other pages but no information on the link itself. So the actual web can be summarized by this formula:

$$ActualWeb = Layout + Routing$$

The problem is that is not possible to automatically reason about the data, the information is not structured.
The Web is global, anyone can publish anything on the web, about any topic:

- Distribution of the information
- Inconsistency of the information
- Incompleteness of the information

The problem is reason upon inconsistency and incompleteness.

8.2 Semantic Web

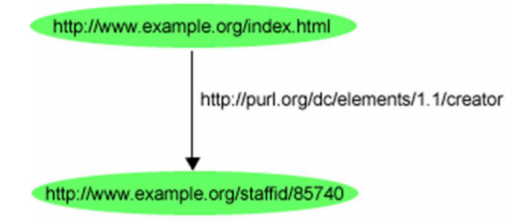
The Semantic Web was proposed in 2001 and the Goal was to use and reason upon all the available data on the internet automatically. How? The proposal of Tim Berners Lee was not to re-write the Web but to extend it somehow and adding information about the knowledge that is represented in a way such that automatic algorithms can exploit it.
But soon they realize that adding information was now enough, information should be structured: **Ontologies**. There is the need of some inference mechanism: **Logic**.

8.2.1 Resource Description Framework (RDF/RDFS)

It's a W3C standard and XML-based language for representing knowledge. It's "minimalist" toold and it is based on the concept of triple:
<subject, predicate,object>
<resource, attribute,value>
We can have different representations (N3,graph, RDF/XML).

The representation as graph is composed by nodes and arcs:

- A node for the subject
- A node for the object
- A labeled arc for the predicate



RDF supports:

- Types (classes) by means of the attrivute type
- Subject/object of a sentence can be also collections
- Meta-sentence, through reification of the sentences ("Marco says that Federico is the author of web page xy)

RDF has many similarity with the E/R model to represent databases, but it is more expressive.
RDFa is a specification for attributes to express structured data in XHTML, the rendered hypertext content of XHTML is reused by the RDFa markup.

8.2.2 SPARQL

SPARQL is a language to query data in the RDF format.

Data:

```
<http://example.org/book/book1>
   <http://purl.org/dc/elements/1.1/title>
"SPARQL Tutorial" .
```

Query:

```
SELECT ?title
   WHERE { <http://example.org/book/book1>
           <http://purl.org/dc/elements/1.1/title>
           ?title .
   }
```

8.2.3 Ontology Web Language (OWL 1.0)

8.3 Knowledge Graphs

The term knowledge graphs was coined during the '90, but no one really pay attention and in 2012 at Google some researcher start writing some paper about knowledge graphs. They needed something able to represent and reason upon knowledge, but the Semantic Web stack seems too complex, and not so fast for the Google needs.
From 2011 Google undergo a transformation: from a search engine to a query-answering engine.
How it did it?

- Create a common, simple vocabulary (schema.org)
- Create a simple (no Tbox), but robust corpus of types, properties, etc...
- Push the web to adopts these standards (well, after all it is Google)

Now based on annotations, Google KG is reported to contains 100B facts about 1B entities.
So the Knowledge Graphs just store the information in therms of nodes and arcs connecting the nodes. Logical formulas are missing, T-Box and A-Box are stored at the same "level".

9 Exam

Usually two questions:

1. one about Prolog or probabilistic logic programming
2. second about more theoretical open question

40 minute to answer (8 point max) Usually the date are the same with the ones of gaspari (16 point)

10 Question and Answers

10.1 Present the Prolog "Vanilla" Meta interpreter and shortly comment the meaning of the clauses

Meta-interpreter works as meta-programs because they execute,works, deal with other programs. In Prolog, a meta-interpreter for a language L is defined as an interpreter for L, but written in Prolog. Vanilla Meta-interpreter is a meta-interpreter for Prolog that works in the same way as the interpreter for Prolog (It represent the standard behaviourof the Prolog interpreter). It is useful because it is the starting point for constructing many other different meta-interpreter for Prolog with different behaviour.

```
1 solve(true):-!.
2 solve((A,B)):-!,solve(A),solve(B).
3 solve(A):-clause(A,Body),solve(Body).
```

In the first line we say that if we query solve with the Goal equal to true we can't backtrack (because of the cut) and Solve(true) is true.
In the second line we have that if we query solve with a conjunction of an atom A and B, where B can possible be formed again of conjunctions, we have first to solve A, and then solve(B). This line represent the leftmost rule of Prolog.
In the third line the Goal is not a conjunction but a single atom so we call the meta-predicate "clause" that find if a clause of the program unify with A and Body and then we solve(Body).

10.2 Present the notion of the cut "!" operator in the Prolog language

shortly explaining how it works. To this end, the candidate is invited to show a short example of a predicate containing the cut, and to illustrate its effects.
The cut is a meta-predicate used in Prolog that allows to interfere and control the execution process of a goal. It is very useful to write very fast Prolog program, but unfortunately it has not logical meaning and is very "low level" operator. When we use the cut (!) in a clause, if executed it remove the choice point from the backtracking stack of the goals before it. So in other words it make the choice taken definitive. An example could be the implementation of the "if then else":

```
1 if_then_else(Cond, A, B):- Cond, !, A.
2 if_then_else(Cond,A,B):-B.
```

or
predicate that receives a list of integers, and returns a new list containing only positive numbers

```
filter([], []):-!.
filter([H|T], [H|R]):- H > 0, !,  filter(T,R).
filter([_|T], R):- filter(T, R).
```

10.3 How can we infer negative clause in Prolog?

If we use the Close world assumption to prove that something is negative if it is not specified in the KB. But this a Non-monotonic inference rule: if we add a new fact to our KB then the inference made in the beginning could not hold anymore. So what we can do is to use negation as failure that derives only the negation of atoms whose proof terminates with failure in a finite time. To solve goals containing also negative atoms, SLDNF has been proposed.It extends SLD resolution with Negation as Failure (NF). Prolog does not use a safe selection rule: it selects always the left-most lit-eral, without checking if it is ground. Indeed, it is a non-correct implementation of SLDNF.

10.4 What is a meta-predicate? list them all

A meta-predicate is a predicate that work (execute, deal) with other predicates.

- The first predicate we are interested in is the *call* predicate. It has arity one, a term T: The term T is considered as a atom, and the Prolog interpreter is requested to evaluate it (execute it). With cut

```
if_then_else (Cond,Goal1,Goal2) :-
      call(Cond), !,
            call(Goal1).
if_then_else (Cond,Goal1,Goal2) :-
      call(Goal2).
```

- Fail Predicate: for example let us consider a KB with facts p/1, apply a predicate q(X) on all X that satisfy p(X)

```
iterate :- call(p(X) ),
           verify (q(X)),
           fail.
           iterate.

verify(q(X)) :- call(q(X)), ! .
```

  To implement the negation as failure We can implement it as failure through a predicate not(P), that is true if P is not a consequence of the program:

```
463
not(P) :- call(P), !, fail.
```

```
464
not(P).
465
```

if call(P) succeed then it will be executed the CUT and then it will fail, if it will not succeed it will backtrack and choose the second not(P) and it will succeed. supporting exceptions.

```
fly(X) :- penguin(X), !, fail.
fly(X) :- ostrich(X), !, fail.
...
fly(X) :- bird(X).
```

- setof, bagof = setof(X,P,S), L is the list of instances X that satisfy the goal P

```
p(1).
p(2).
p(0).
p(1).
q(2).
r(7).

:- setof (X,p(X),S).
  yes   S = [0,1,2]
       X = X

:- bagof (X,p(X),S).
  yes   S=[1,2,0,1]
       X = X
```

```
father (giovanni, mario).
father (giovanni, giuseppe).
father (mario, paola).
father (mario, aldo).
father (giuseppe, maria).

:- setof (X, father(X,Y), S).
  yes X=X Y=aldo S=[mario];
         X=X Y=giuseppe S=[giovanni];
           X=X Y=maria S=[giuseppe];
           X=X Y=mario S=[giovanni];
           X=X Y=paola S=[mario];
     no
```

- findall
  - **var(Term)** →true if Term is currently a variable.
  - **nonvar(Term)** → true if Term currently is not a free variable.
  - **number(Term)** → true if Term is a number.
  - **ground(Term)** →true if Term holds no free variables.
- LIST

```
Term =.. List
```

- Clause

```
clause(Head, Body)
```

  we are able to query the program itself and see indeed there is or not a fact of the type of the program. clause is true if (Head,Body) is unified with a clause stored within the database program

10.5 How can I dynamically modify the Prolog program?

A Prolog program is loaded into a table in memory, this table is often referred to as the program database, indeed is managed with DBMS technique. So if it is a table we can change it with:
assert(T)
retract(T)

10.6 What is LPAD? How it works?

LPAD (Logic Program with Annotated Disjunctions) is a probabilistc logic programming language, extension of Prolog. In LPAD the head of the clause is formed by disjunction and each disjunct is annotated with a probability. When I make a Query I have to

1. Ground the Program
2. We build worlds choosing only a disjunction for each clause. So we obtain a specific Number of worlds. Each world has as probability the product of the probability of the chosen disjunction.

The response of my query will be the sum of the probability of the worlds in which my query is true.
So in this way we can unify two aspect: the logic with the probabilistic.
This can be useful in expert system, where the program works with an expert of the field. The program could ask to the expert about a probability of an term.

10.7 What Forward and Backward reasoning are?

In backward reasoning (the one used by Prolog) we start from the goal and then we reason backward to find facts that support our goal in the knowledge base. In the Forward reasoning approach we start from the data, from the facts in our knowledge base and infer new knowledge until the goal is reached. The Forward reasoning is commonly used in the production rule system.

10.8 What is a Production Rule system?

A Production rule system is a system that use rules and forward reasoning to infer (create, deduce) some new knowledge. Usually it uses a dynamic knowledge base indeed it is possible to have side effects in the RHS of the rules which can modify both the external world and the knowledge base itself.
When we add a new fact in the knowledge base (working memory, that is the data structure at the base of Production rule system):

1. First we have a match algorithm that search for the rules whose LHS matches with the facts in the KB and decide which one will trigger
2. Then the possible triggered rules are put in an Agenda and possible conflicts are solved
3. The RHS are executed and the effects are performed.

10.9 Talk about The RETE algorithm

The RETE algorithm is used in Production rule systems for the match step. The match step consist of searching for rules whose LHS match with the facts in the KB. A naive approach would consist of iterating over all the rules, but this is computationally very expencive. The RETE algorithm solve this problem by storing for each pattern the facts that match it. There are two type of pattern, the ones testing intra-element features, that are called alpha network, and the ones testing inter-element feature, that are called beta network. The outcome of the alfa network goes to alfa memories while the outcome of the beta network goes to beta memories. In the end the beta memories will contain the conflict set of rules whose LHS has been completely matched and so we move them to the conflix set resolution step.

10.10 Talk about the DROOLS framework

DROOLS is a framework for Production rule systems, it is based on forward reasoning and it uses the RETE algorithm for the match step. It can support Complex event processing through an extension called drools fusion. It is Java based, but the language is proprietary. The programmer has to define Java Bean. For the Conflict Resolution Step it supports Salience, Agenda Group.

10.11 Talk about Complex Event Processing

Complex event processing is the research field that deal with huge amount of data at a time along stream of information. The information are collected as events that is the description of something with a temporal information. The main task of complex event processing is to derive complex event from the aggregation of simple event.

10.12 Talk about Drools fusion

Drools fusion is a module of Drools that enables it to deal with Complex Event Processing. Event are defined as particular facts that have a timestamp, it support Allen algebra. It supports sliding windows. The problem with Drools fusion is that is difficult to reason on states, if we have a lot of variable we should reason with Event Calculus.

10.13 Talk about Event Calculus Framework

Event Calculus Framework is a logic-based approach to Complex Event Processing. It enable us to reason upon events and also on the state of the system through fluents that are properties whose truthness value can change over time. It allow also to reason on meta-event on state property change. The event calculus framework is composed by three elements:

1. Event calculus ontology: contains predicate like HoldsAt, Happens, Initiates, Terminates, Clipped, Initially
2. Domain-independent axioms
3. Domain-dependent axioms

It can be easily implemented in Prolog because is logic-based but it has two problems:

1. Computationally costly because I have to restart the inference task every time we change a fluent or we add a new one.
2. In the definition of HoldsAt there is a NOT Clipped, and we know that the negation in prolog, if the term in not ground bring some problems.

We should implement something similar to Drools but for event calculus. To do that we should keep separate the events from the fluents that represent the state.

other answer The Event Calculus Framework is a framework based on Complex Event Processing and can be easily implemented in Prolog. In particular, it is used to deal with properties whose truthness value changes over time (fluents). It uses some predicates such as HoldsAt or Happens to check if some fluents hold (or some events happen) at specific moments.

10.14 Talk about Semantic Web and Knowledge Graph

The Semantic Web is an idea born with Tim Berners Lee that consist of enable to use all the data the the World Wide Web produce. In the beginnign the Web had to be readable only by humans, with the semantic web we want that also the machines can understand it. With the semantic web we want to define a data format that is inter-operable, RDF/XML is choosen. Data are represented as Graph where the nodes are URI and archs define relationship between URI.

So we have triplets (¡subject, predicate, object¿). This metadata can be insert into web pages (HTML5). We can query this data with a query language called SPARQL. In Semantic Web We have the A-box and T-box, so we keep separate the data (A-box) from the ontology(T-box) (OWL).
Google think only about efficiency so it can't lose time with the reasoning (inference) so they don't use the T-box and they put all in the A-box.

10.15 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.

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

10.16 Write a meta-predicate how_many(Pred, Num_of_clauses)

that, given in input a predicate Pred, returns in Num_of_clauses the number of clauses defining the predicate Pred in the current program.

```
how_many(P, N) :- findall(Body, clause(P, Body), Bodies), length(
    Bodies, N).
```

or

```
how_many(P,N):-setof(Body, clause(P,Body),Bodies), length(Bodies,N)
    .
```

or

```
how_many(P,N):-bagof(Body, clause(P,Body),Bodies), length(Bodies,N)
    .
```

10.17 The candidate is invited to write a meta-interpreter verbose(Goal)

that is evaluated to true if Goal can be proved; moreover, before trying to solve any sub-goal, the meta-interpreter must print (at the standard output) the sub-goal itself.

```
verbose(true) :- !.
verbose((A,B)):- !, verbose(A), verbose(B).
verbose(A):- print("trying to solve"), print(A), nl, clause(A, B),
     verbose(B).
```

10.18 The candidate is invited to write a meta-interpreter verbose(Goal)

that is evaluated to true if Goal can be proved; moreover, after having solved any subgoal, the meta-interpreter must print (at the standard output) the sub-goal itself. Hence, only goals that have been proved will be printed at the output.

```
verbose(true) :- !.
verbose((A,B)):- !, verbose(A), verbose(B).
verbose(A):- clause(A, B), verbose(B), print("solved:"), print(A),
    nl.
```

10.19 Other exercises

```

/*FACTORIAL*/

factorial(0,1):-!.
factorial(N, R):- N>0, N1 is N-1, factorial(N1,R1), R is R1*N.

/*FACTORIAL PRINTING THE INTERMEDIATE RESULTS*/

factorial(0,1):-!.
factorial(N, R):-N>0, write(N),nl,  N1 is N-1, factorial(N1,R1), R
    is R1*N.


/*Write a predicate that given a list, it returns the last element
    */
last_element([], null):-!.
last_element([A|[]], A):- !.
last_element([_|R], LE):- last_element(R,LE).

/*Write a predicate  memeber that succeed if an element is in a
    list*/
my_member(X, [X|_]):-!.
my_member(X, [_|R]):-my_member(X,R).


/*Write a predicate that given two lists L1 and L2, returns true if
     and only
if L1 is a sub-list of L2. */

sub_list([], _):-!.
sub_list([A1|R1], L2):- my_member(A1,L2), sub_list(R1,L2).

/*Write a predicate that returns true if and only if a list is a
    palindrome.
palindrome([]):-!.*/
palindrome([_|[]]):-!.
palindrome([A|R]):- last_element(R,L), A == L, remove_last_element(
    R, R1), palindrome(R1).

last_element([A|[]],A):-!.
```

```
last_element([_|R],L):-  last_element(R,L).

remove_last_element([_|[]],[]):-!.
remove_last_element([A|R], [A|L]):- remove_last_element(R,L).



/*Write a predicate that given a term T and a list L, counts the number of
occurrences of T in L.*/
count_occurrence(_,[], 0):-!.
count_occurrence(T, [T|R], N):- count_occurrence(T,R,N1), N is N1 +1, !.
count_occurrence(T, [_|R], N):- count_occurrence(T,R,N).


/*Write a predicate that, given a list, returns a new list obtained by
flattening the first list. Example: given the list [1,[2,3,[4]],5,[6]] the
predicate should return the list [1,2,3,4,5,6].
*/
%% The empty list is already flat.
myflatten([],[]).

%% For a non-empty list: flatten the head and flatten the tail and
%% append the results.
myflatten([Head|InTail],Out) :-
  myflatten(Head,FlatHead),
  myflatten(InTail,OutTail),
  append(FlatHead,OutTail,Out).
%% When trying to flatten the head, you might find that the head is
%% not itself a list. In this case the head of the input list simply
%% becomes the head of the output list.
myflatten([Head|InTail], [Head|OutTail]) :-
  Head \= [],
  Head \= [_|_],
        myflatten(InTail,OutTail).
```