



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Prolog – A (not so) quick recall

Prof. Ing. Federico Chesani

DISI

Department of Informatics – Science and Engineering

Prolog Interpreters

- **SICStus Prolog**
 - Probably, the fastest Prolog interpreter and compiler
 - Commercial tool (€€€)
 - <https://sicstus.sics.se/>
- **SWI-Prolog (my suggested interpreter)**
 - Well supported Prolog, quite used in AI research
 - Free
 - Available as a stand-alone, and also as a web app (nice for quick tests)
 - <https://www.swi-prolog.org/>
 - <https://swish.swi-prolog.org/>
- **ECLiPSe**
 - Mind the name!!! (indeed, they arrived first...)
 - Strong support to Constraint Logic Programming
 - <https://eclipseclp.org/>
- **TuProlog**
 - Completely java-based
 - Comes as a jar library, or with a GUI
 - Developed by our colleagues in Cesena
 - <https://gitlab.com/pika-lab/tuprolog/2p>



Prolog Interpreters – seriously...

- **For small exercise:**
 - TuProlog <https://gitlab.com/pika-lab/tuprolog/2p>
 - SWI-Prolog web app <https://swish.swi-prolog.org/>
- **For serious use:**
 - SWI-Prolog
 - Editor: Visual Studio Code + Prolog plugin for syntax highlighting



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Few instructions...

- When the prolog interpreter is executed, it assumes a “working directory”: it will look for files only in the current working directory.
 - To discover the current working dir: “**pwd.**”
 - To change working dir: “**working_directory(Old, New).**”
- Programs must be loaded, through the pre-processing of source files
 - Given a file “**test.pl**”, it can be loaded through “**consult(test).**”
- When a program source is modified, it needs to be reloaded into the database clause
 - Command “**make.**” reloads all the changed files
 - It is always a good practice to close the interpreter, and run it again, from time to time...
- Debugging?
 - Command “**trace.**” ... but before, read the official documentation...



Terminology

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

- Fact **A.**
- Rule **A :- B₁, B₂, ..., B_n.**
- Goal **: - B₁, B₂, ..., B_n.**

Where **A** and **B_i** atomic formulas:

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



Terminology

- An **atomic formula** has the form:

$$p(t_1, t_2, \dots, t_n)$$

- p is a predicate symbol (alphanumeric string starting with low-capital letter)
- t_1, t_2, \dots, t_n are terms

- A **term** is recursively defined as:

- **constants** (integer/floating numbers, alphanumeric strings starting with a low-capital letter) are terms
- **variables** (alphanumeric strings starting with a capital letter or starting with the symbol "`_`") are terms.
- **$f(t_1, t_2, \dots, t_k)$** is a term if " f " is a **function symbol** with k arguments and t_1, t_2, \dots, t_k are terms.
 $f(t_1, t_2, \dots, t_k)$ is also known as structure. Constants can be viewed as functions with zero arguments (arity zero).



Terminology – few examples

- Constants: `a`, `goofey`, `aB`, `9`, `135`, `a92`
- Variables: `x`, `x1`, `Goofey`, `_goofey`, `_x`, `_`
 - variable `_` is usually named as the anonymous variable
- Compound terms: `f(a)`, `f(g(1))`, `f(g(1),b(a),27)`
- Atomic formulas: `p`, `p(a,f(X))`, `p(Y)`, `q(1)`
- Definite clauses:
 - `q.`
 - `P :- q, r.`
 - `r(Z).`
 - `p(X) :- q(X, g(a)).`
- Goal:
 - `:- q, r.`
- **NOTICE: NO DISTINCTION between constants, function symbols and predicate symbols!!!**



Declarative interpretation

Variables within a clause are universally quantified

- For each fact: $p(t_1, t_2, \dots, t_m)$.
If x_1, x_2, \dots, x_n are the variables appearing in t_1, t_2, \dots, t_m the intended meaning is:
 $\forall x_1, \forall x_2, \dots, \forall x_n (p(t_1, t_2, \dots, t_m))$
- For each rule: $A :- B_1, B_2, \dots, B_k$.
 - If y_1, y_2, \dots, y_o are the variables appearing in the body only
 - If x_1, x_2, \dots, x_n are the variables appearing in both the body and the head
$$\forall x_1, \forall x_2, \dots, \forall x_n, \forall y_1, \forall y_2, \dots, \forall y_o ((B_1, B_2, \dots, B_k) \rightarrow A)$$
$$\forall x_1, \forall x_2, \dots, \forall x_n ((\exists y_1, \exists y_2, \dots, \exists y_n (B_1, B_2, \dots, B_k)) \rightarrow A)$$



Declarative interpretation – Examples

father (X, Y) “**X** is the father of **Y**”

mother (X, Y) “**X** is the mother of **Y**”

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

“for each **X, Y**, **X** is the grandfather of **Y** if it exists **Z** s.t. **X** is father of **Z** and **Z** is father of **Y**”

grandfather (X, Y) :- father (X, Z), mother (Z, Y) .

“for each **X, Y**, **X** is the grandfather of **Y** if it exists **Z** s.t. **X** is father of **Z** and **Z** is mother of **Y**”



Execution of a (Prolog) program

- A computation is the attempt to prove, through resolution, that a formula logically follows from the program (it is a theorem).
- Moreover, it aims to determine a **substitution** for the variables in the goal, for which the goal logically follows from the program.

- Given a program **P** and the goal/query:

$$:- \ p(t_1, t_2, \dots, t_m) .$$

- if **x₁, x₂, ..., x_n** are the variables appearing in **t₁, t₂, ..., t_m** then the meaning is:

$$\exists x_1, \exists x_2, \dots, \exists x_n \ p(t_1, t_2, \dots, t_m)$$

- The objective is to determine a substitution

$$\sigma = \{x_1/s_1, x_2/s_2, \dots, x_n/s_n\}$$

- where **s_i** are terms such that **P** $\models [p(t_1, t_2, \dots, t_m)]\sigma$



SLD Resolution

- The resolution adopted by LPis the SLDthat has two non-determinisms:
 - The rule of computation
 - The search strategy

Prolog adopts SLD with the following choices:

- Rule of computation:
 - **Rule "left-most"**; given a "query":
 ?- G1, G2, ..., Gn.
 It is selected the left-most literal (G1).
- Search strategy
 - **Depth-first** with chronological backtracking

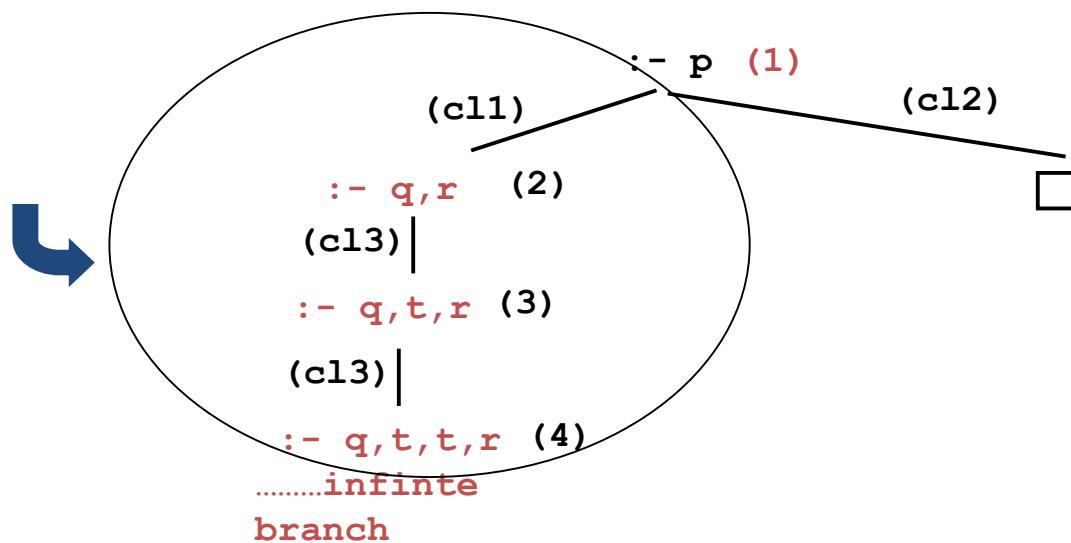


Depth-first and incompleteness in Prolog

- The order of the clauses in the program may greatly affect the termination, and consequently the completeness

P_2

```
(cl1) p :- q,r.  
(cl2) p.  
(cl3) q :- q,t.  
:- p.
```



Order of the clauses in Prolog

- Clauses order in Prolog is highly relevant

P2

(c11) $p :- q, r.$
(c12) $p.$
(c13) $q :- q, t.$

P3

(c11') $p.$
(c12') $p :- q, r.$
(c13') $q :- q, t.$

- Programs P2 and P3 are NOT EQUIVALENT
- Given the query "query": $: - p.$ you get:
 - The proof with P2 does not terminate;
 - The proof with P3 successfully terminates (immediately).
- However, a depth first strategy can be **efficiently** implemented using techniques similar to those adopted in procedural programming (**TAIL RECURSION**).



Multiple solutions, and the disjunction

- There may exist multiple answers for a query.
- How to get them? After we get an answer, we could force a failure: backtracking should be started then, looking for the "next" solution.
- Practically, it means to ask the procedure to explore the remaining part of the SLD tree.
- In Prolog standard we can ask for more solutions through the operator ";".

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

- Operator ";" can be interpreted as:
 - A disjunction operator, that looks for alternative solutions
 - Within a Prolog program, to express the disjunction.



Procedural Interpretation of Prolog

- A procedure is a set of clauses of a program P whose:
 - Heads have the same predicate symbol
 - Predicate symbols have the same arity (same number of arguments)
- Arguments appearing in the head of the procedure are the **formal parameters**
$$:- \ p(t_1, t_2, \dots, t_n).$$
This can be viewed as the **call** to procedure p.
Arguments of p (terms t_1, t_2, \dots, t_n) are the **actual parameters**.
- Unification then can be viewed as the mechanism for **passing the parameters**.
- There is no distinction between input parameters and output parameters (**reversibility**)



Procedural Interpretation of Prolog

- 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.
- All the variables are "**single assignment**": they assume a single value along the proof, except when looking for alternative paths in the SLD tree ("backtracking").



Example

```
play_sport(mario,football).  
play_sport(giovanni,football).  
play_sport(alberto,football).  
play_sport(marco,basket).  
live(mario,torino).  
live(giovanni,genova).  
live(alberto,genova).  
live(marco,torino).
```

```
:- play_sport(X,football). %"does exists X such that X plays  
football?"
```

```
yes      X=mario;
```

```
X=giovanni;
```

```
X=alberto;
```

```
No
```

```
:- play_sport(giovanni,Y).
```

```
yes      Y=football;
```

```
no
```



Example

```
:  
:- play_sport(X,Y) .  
% "Do exist X and Y s.t. X plays the sport Y?"  
  
yes      X=mario          Y=football;  
          X=giovanni        Y=football;  
          X=alberto          Y= football;  
          X=marco            Y=basket;  
  
no
```



```
:  
:- play_sport(X,football), live(X,genova) .  
% "Does exist X s.t. plays football and lives in Genova?"  
  
yes      X=giovanni;  
          X=alberto;  
  
no
```



Arithmetic

- In Logic there is not any way to evaluate functions. That might be a problem for us...
- Integer numbers can be represented through the Peano notation:
$$s(s(s(\dots s(0)\dots)))$$

`prod(X, 0, 0).`

`prod(X, s(Y), Z) :- prod(X, Y, W), sum(X, W, Z).`

- Non practicable...



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Arithmetic

- In Prolog, both integers and floating point numbers are atoms.
- Math operators are function symbols predefined in the language.
- Every expression is then a Prolog term.
- Binary operators are supported with a pre-fix notation, as well as the in-fix notation.

Unary operators

`-, exp, log, ln, sin, cos, tg`

Binary operators

`+, -, *, \, div, mod`

- `+(2,3)` and `2+3` are equivalent.



Evaluation of expressions

However, when we write an expression, we would like to have it evaluated...

- Special pre-defined predicate **is**.

$$T \text{ is Expr} \quad (\text{is}(T, \text{Expr}))$$

- **T** can be a numerical atom or a variable
- **Expr** must be an expression
- Expression **Expr** is evaluated and the result is unified with **T**

Variables in **Expr** **MUST BE COMPLETELY INSTANTIATED** at the moment of evaluation.



Evaluation of expressions – examples

```
: - x is 2+3.
```

yes x=5

```
: - x1 is 2+3, x2 is exp(x1), x is x1*x2.
```

yes x1=5 x2=148.413 x=742.065

```
: - 0 is 3-3.
```

yes

```
: - x is Y-1.
```

No

(or Instantiation Fault, depending on the prolog system)

```
: - x is 2+3, x is 4+5.
```

no



Evaluation of expressions – examples

```
: - x is 2+3, x is 4+1.
```

yes x=5

In this example the second goal is:

```
: - 5 is 4+1.
```

X has been instantiated by the evaluation of the first goal.

```
: -      x is 2+3, x is x+1.
```

No

No way: there is not the assignment like in procedural languages.

Variables are write-once....



Evaluation of expressions – examples

With the operator **is**, the order of the goals is very important:

(a) :- X is 2+3, Y is X+1.

(b) :- Y is X+1, X is 2+3.

- Goal (a) succeeds and returns
X=5, Y=6
 - Goal (b) fails.
-
- The predefined predicate "is" is not reversible.
Procedures that use it are not (generally speaking) reversible.



Expressions and Terms

A term representing an expression is evaluated only if it is the second argument of a predicate "is"

```
p(a,2+3*5) .
```

```
q(X,Y) :- p(a,Y), X is Y.
```

```
: - q(X,Y) .
```

```
yes X=17 Y=2+3*5 (Y=+(2,* (3,5)))
```

Notice: Y is not evaluated, but unified with a structure that has "+" as operator, and arguments "2" and another structure with * as operator and arguments 3 and 5.



Relational operators

- It is possible to compare expression values
- Such operators can be used as goals within clauses, and have in-fix notation

OPERATORI RELAZIONALI

>, <, >=, =<, ==, /=



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Comparing expressions

Expr1 REL Expr2

- where **REL** is a relational operator, and **Expr1** and **Expr2** are expressions
- **Expr1** and **Expr2** are evaluated: Mind it! They both have to be completely instantiated.
- The results are then compared on the basis of **REL**



Math functions...

- Given the math operators and "is", it is possible to define functions in Prolog
- Given f with arity n :
 - it can be implemented through a $(n+1)$ -arity predicate
 - $f : x_1, x_2, \dots, x_n \rightarrow y$ is written as
 $f(X1, X2, \dots, Xn, Y) :- <\text{compute } Y>$
 - Example: compute the factorial

fatt: $n \rightarrow n !$ (n positive integer)

fatt(0) = 1

fatt(n) = $n * \text{fatt}(n-1)$ (for $n > 0$)

fatt(0, 1).

fatt(N, Y) :- $N > 0$, $N1 \text{ is } N - 1$, fatt(N1, Y1), $Y \text{ is } N * Y1$.



Iteration and recursion in Prolog

- In Prolog there is no iteration (no while, for, repeat...)
- But... you can get iterative behaviour through recursion!!!
- A function f is tail-recursive if f is the "most external call" in the recursive definition
- In other terms, if the result of the recursive call is not subject of any other call.
- Tail-recursion is equivalent to iteration
 - It can be evaluated in constant space, like in iteration.
 - However, in Prolog things get complicated: non determinism, choice points...



Non-tail recursion, and conversion to tail recursion

- There are many cases where a non-tail recursion can be re-written as a tail recursion

```
fatt1(N, Y) :- fatt1(N, 1, 1, Y).  
fatt1(N, M, ACC, ACC) :- M > N.  
fatt1(N, M, ACCin, ACCout) :- ACCtemp is ACCin*M,  
                                M1 is M+1,  
                                fatt1(N, M1, ACCtemp, ACCout).  
  
Input  
accumulator  
Output  
accumulator
```



Non-tail recursion, and conversion to tail recursion

- The factorial is computed using an accumulator, initialized to 1, and incremented at every step, and unified only at the termination of the recursion.

$$\text{ACC}_0 = 1$$

$$\text{ACC}_1 = 1 * \text{ACC}_0 = 1 * 1$$

$$\text{ACC}_2 = 2 * \text{ACC}_1 = 2 * (1 * 1)$$

...

$$\text{ACC}_{N-1} = (N-1) * \text{ACC}_{N-2} = N-1 * (N-2 * (\dots * (2 * (1 * 1)) \dots))$$

$$\text{ACC}_N = N * \text{ACC}_{N-1} = N * (N-1 * (N-2 * (\dots * (2 * (1 * 1)) \dots)))$$



Non-tail recursion, and conversion to tail recursion

Another solution to the tail-recursive version of the factorial:

```
fatt2(N,Y)
    "Y is the factorial of N"

fatt2(N,Y)  :-  fatt2(N,1,Y) .

fatt2(0,ACC,ACC) .

fatt2(M,ACC,Y)  :-  ACC1 is M*ACC,
                    M1 is M-1,
                    fatt2(M1,ACC1,Y) .
```



Lists in Prolog – some exercises

1. Write a predicate that given a list, it returns the last element.
2. Write a predicate that given two lists L1 and L2, returns true if and only if L1 is a sub-list of L2.
3. Write a predicate that returns true if and only if a list is a palindrome.
4. Write a predicate that, given a list (possibly with repeated elements), returns a new list with repeated elements.
5. Write a predicate that given a term T and a list L, counts the number of occurrences of T in L.
6. 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].
7. Write a predicate that given a list, returns a new list that is the first one, but ordered.



The CUT



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Controlling a (Prolog) program

- There are a number of pre-defined predicates that allows to interfere and control the execution process of a goal.
- The cut "!" is among them.
- No logic meaning, no declarative semantics...
- ... but it heavily affects the execution process



Controlling a (Prolog) program – some insight on the execution process

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

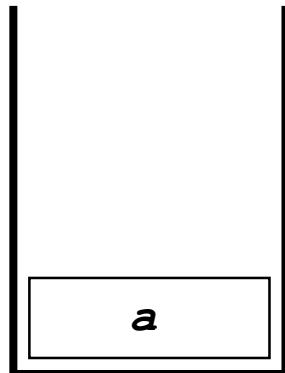
- **Execution stack**: it contains the activation records of the procedures/predicates
- **Backtracking stack**: it contains the set of open choice points.



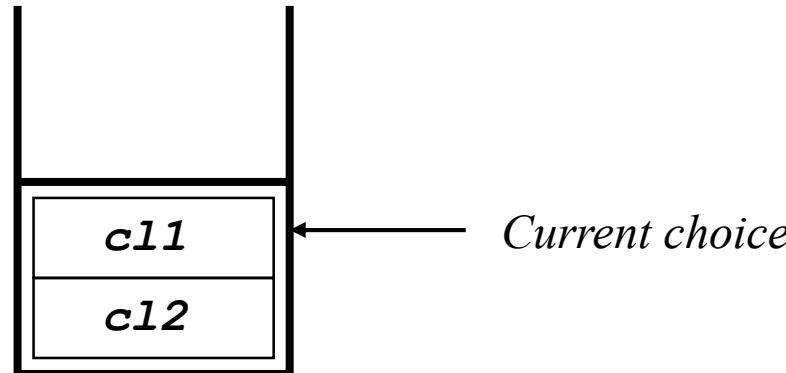
Controlling a (Prolog) program – example

```
(c11)      a :- p,b.  
(c12)      a :- r.  
(c13)      p :- q.  
(c14)      p :- r.  
(c15)      r.
```

– Query :-a.



Execution stack

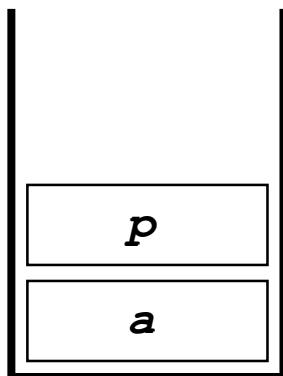


Backtracking stack

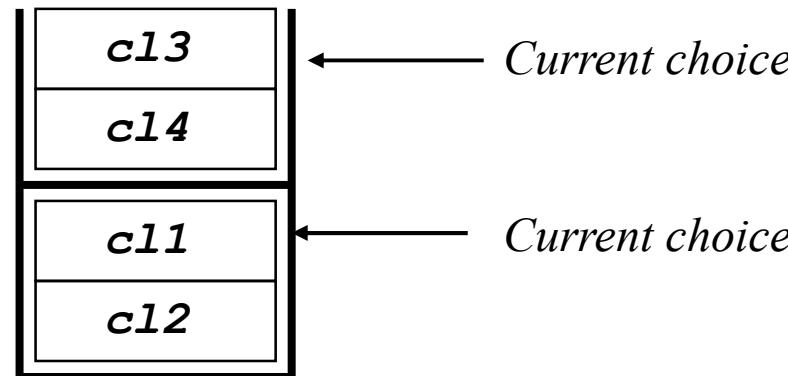


Controlling a (Prolog) program – example

```
(c11)      a :- p,b.  
(c12)      a :- r.  
(c13)      p :- q.  
(c14)      p :- r.  
(c15)      r.
```



Execution stack

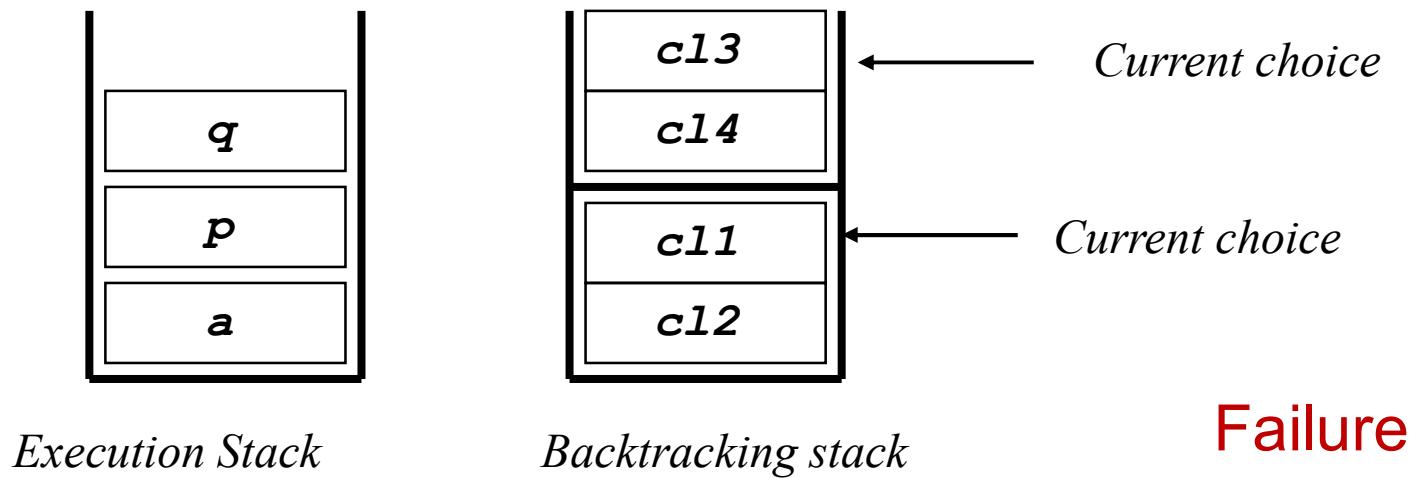


Backtracking stack



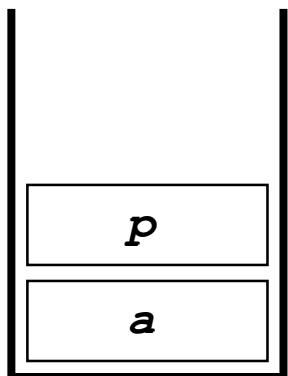
Controlling a (Prolog) program – example

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

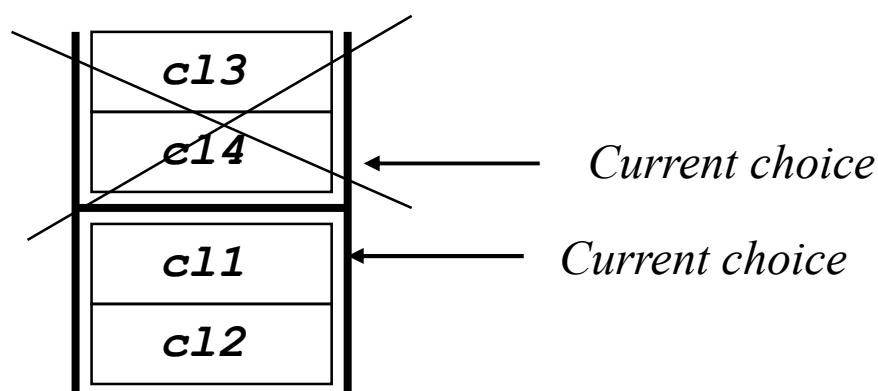


Controlling a (Prolog) program – example

```
(c11)      a :- p,b.  
(c12)      a :- r.  
(c13)      p :- q.  
(c14)      p :- r.  
(c15)      r.
```



Execution stack

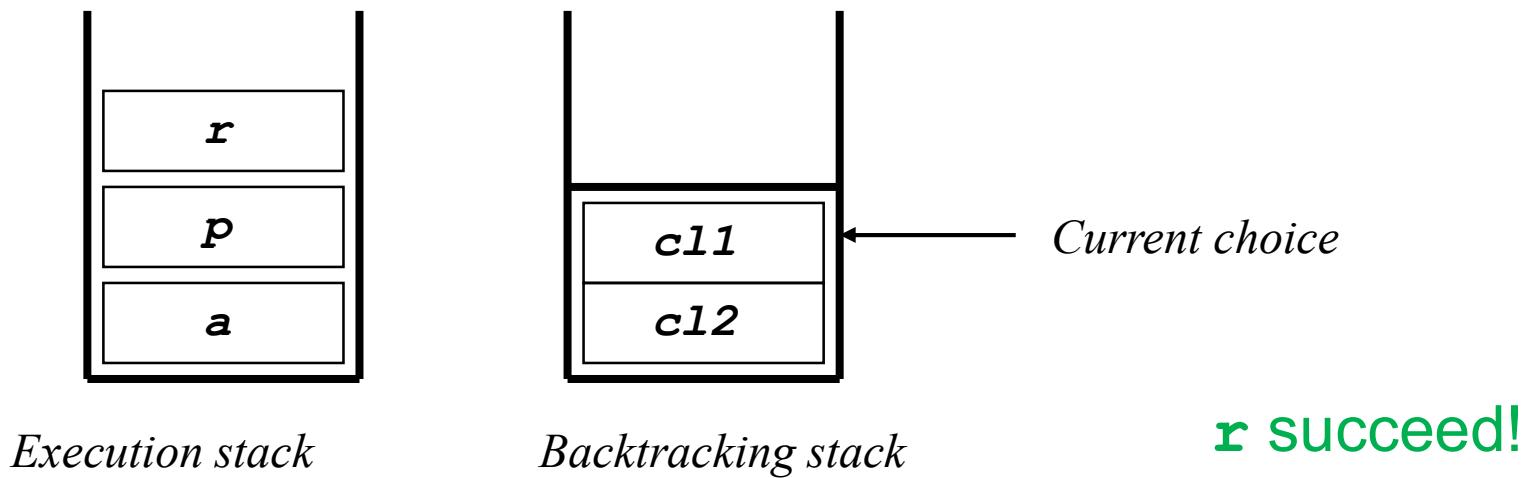


Backtracking stack



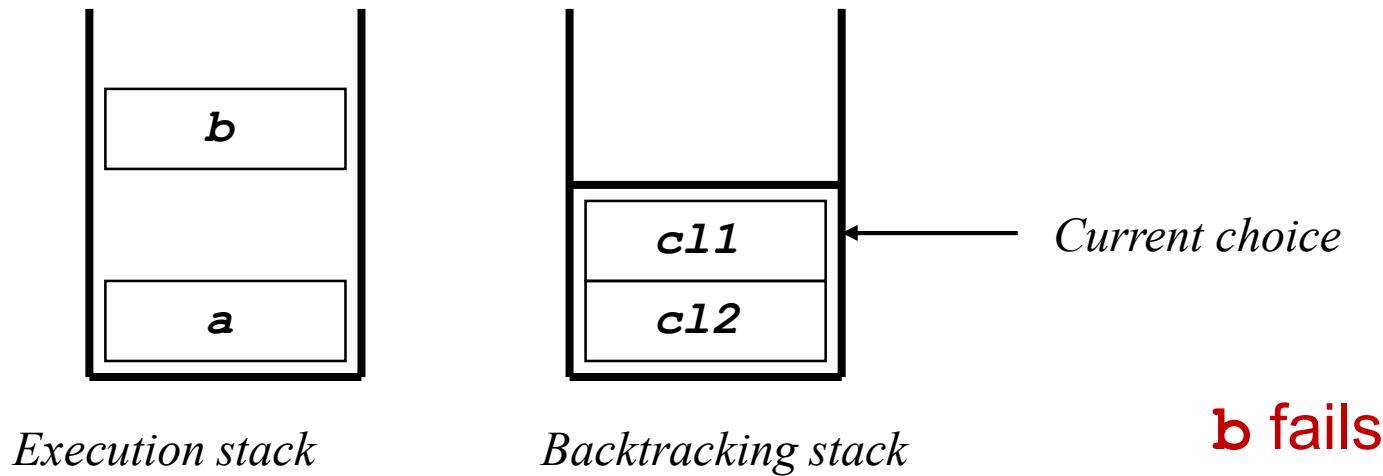
Controlling a (Prolog) program – example

```
(c11)      a :- p,b.  
(c12)      a :- r.  
(c13)      p :- q.  
(c14)      p :- r.  
(c15)      r.
```



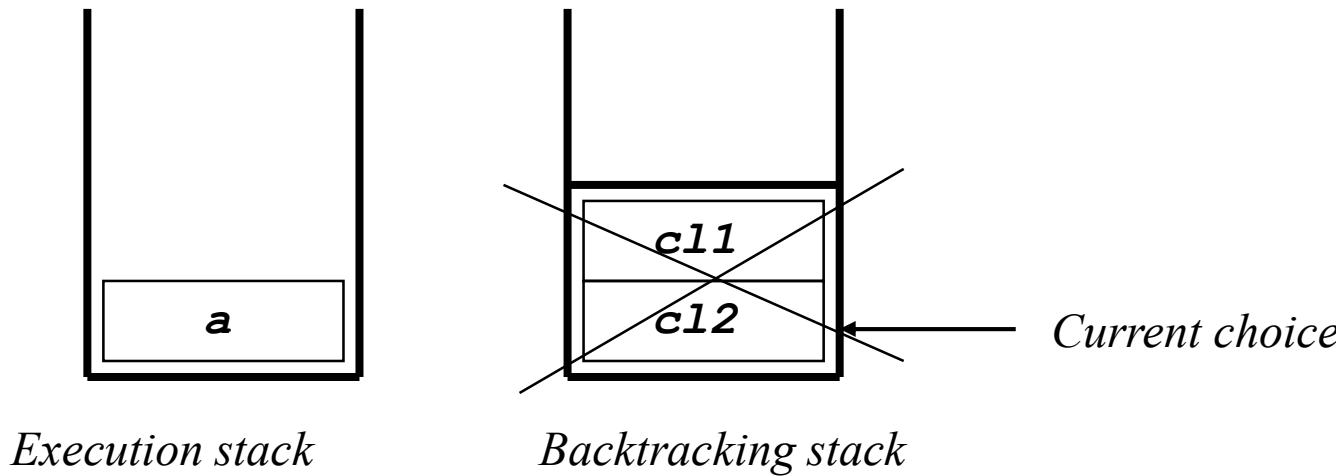
Controlling a (Prolog) program – example

```
(c11)      a :- p,b.  
(c12)      a :- r.  
(c13)      p :- q.  
(c14)      p :- r.  
(c15)      r.
```



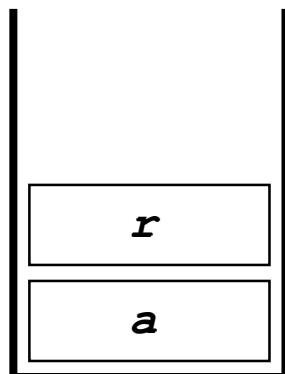
Controlling a (Prolog) program – example

```
(c11)      a :- p,b.  
(c12)      a :- r.  
(c13)      p :- q.  
(c14)      p :- r.  
(c15)      r.
```

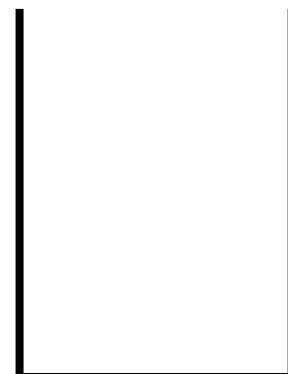


Controlling a (Prolog) program – example

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



Execution stack



Backtracking stack

Success



How CUT works

- 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....
- When used, declarativeness is partially lost



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

How CUT works

Consider the clause:

$p :- q_1, q_2, \dots, q_i, !, q_{i+1}, q_{i+2}, \dots, q_n.$

- The evaluation of $!$ succeeds, and it is ignored in backtracking;
- All the choices made in the evaluation of goals q_1, q_2, \dots, q_i and goal p are made definitive; in other words, the choice points are removed from the backtracking stack.
- Alternative choices related to goals placed after the cut are not touched/modified.



How CUT works

Consider the clause:

$$p :- q_1, q_2, \dots, q_i, !, q_{i+1}, q_{i+2}, \dots, q_n.$$

- If the evaluation of $q_{i+1}, q_{i+2}, \dots, q_n$ fails, then the "whole" p fails. Even if there were other alternatives for p , these would have been removed by the cut.
- The cut removes branches of the SLD tree, hence it cannot be defined in a declarative way.



CUT – example

```
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
```



CUT – example

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

```
p(X) :- q(X), !, r(X).  
q(1).  
q(2).  
r(2).  
  
:- p(X).  
no
```



The CUT to achieve mutual exclusion between two clauses

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

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

Using the "cut":

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

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

- 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.



The CUT to achieve mutual exclusion between two clauses – example

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

```
filter([], []).  
filter([H|T], [H|Rest]) :- H>0, filter(T, Rest).  
filter([_|T], Rest)      :- H=<0, filter(T, Rest).
```

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



The Negation



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The problem of the 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, since it cannot be proved by using the facts in our KB.



The Close World Assumption

- This intuition is usually granted in the database field, where only positive information are recorded
- In Logic, this intuition is formalized **Closed World Assumption (CWA)** [Reiter '78].
If a ground atom A is not logical consequence of a program P, then you can infer $\sim A$

$CWA(P) = \{\sim A \mid \text{it does not exist a refutation SLD for } P \cup \{A\}\}$



Close World Assumption – example

```
capital(rome) .
```

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

```
city(bologna) .
```

- Using CWA, we can infer **~capital(bologna)** .
- CWA is a **non-monotonic** inference rule: adding new axioms to the program might change the set of theorems that previously held.



Close World Assumption – are we happy?

- 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.

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



Close World Assumption... and Negation as Failure

- Let's drop the CWA: Prolog adopts a less powerful rule, the Negation as Failure
- Negation as Failure [Clark 78], derives only the negation of atoms whose proof terminates with failure in a finite time.
- Given a program P, we name $\text{FF}(P)$ the set of atoms for which the proof fails in a finite time
- NF rule:

$$\text{NF}(P) = \{ \neg A \mid A \in \text{FF}(P) \}$$



Negation as Failure

- If an atom A belongs to $\text{FF}(P)$, then A is not logical consequence of P
- Not all the atoms that are not logical consequence of P belong to $\text{FF}(P)$

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

```
city(bologna) .
```

- `city(rome)` is not logical consequence of P... but it does not belong to $\text{FF}(P)$



SLDNF

- To solve goals containing also negative atoms, SLDNF has been proposed [Clark 78]. It extend SLD resolution with Negation as Failure (NF).
- Let $: - L_1, \dots, L_m$ be the goal, where L_1, \dots, L_m are literals (atoms or negation of atoms).
- A SLDNF step is defined as:
 - Do not select any negative literal L_i , if it is not "ground";
 - If the selected literal L_i is positive, then apply a normal SLD step
 - If L_i is $\sim A$ (with A "ground") and A fails in finite time (it has a finite failure SLD tree), then L_i succeeds, and the new resolvent is:

$: - L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_m$



SLDNF

- Proving with success a negative literal does not introduce any substitution, since the literal is ground
- No substitution is applied to the new resolvent

$\text{:- } L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_m$

- 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 and Prolog

- 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



NAF in Prolog – example

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

```
: - city(X), ~capital(X).  
      |  
:- capital(X),  
  ~capital(X).  
      |  
      X/rome  
:- ~capital(rome).  
  
      |  
:- chief_town(X),  
  ~capital(X).  
      |  
      X/bologna  
:- ~capital(bologna).
```

```
: - capital(rome).  
      |  
      □
```

fail

```
: - capital(bologna).  
      |  
      fail
```

success



NAF in Prolog – Are we happy now?

- 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
- Mmmmh... okay but... what happens if a non-ground negative literal is selected?



SLDNF in Prolog – example

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

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



```
:- capital(X)  
| x/roma  
□
```

fail

INCORRECT!!!



SLDNF and Quantification of variables

- 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?"

$$F = \exists \ x \ \sim \text{capital} (x) .$$

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



SLDNF and Quantification of variables

- Instead, in SLDNF, we are looking a proof for
:- **capital (X)** .

with the explicit quantifier:

$$F = \exists x \text{ capital}(x) .$$

- Then, the result is negated:

$$F = \sim (\exists x \text{ capital}(x)) .$$

syntactic transformation:

$$F = \forall x (\sim \text{capitale}(x))$$

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



Some exercises...

- Define a Prolog predicate **maxlist/2** that takes in input a list of list of integers, and returns a new list, containing the max element of each list of integers provided in input. Define both the recursive and the tail-recursive versions.

```
?- maxlist([[3,10,2], [6,9], [1,2]], X).  
yes, X = [10,9,2]
```



Exercise – Solution

```
maxlist([],[]).  
maxlist([X|Y],[N|T]) :- max(X,N),maxlist(Y,T).
```

Recursive version:

```
max([X],X) :- !.  
max([X|T],X) :- max(T,N), X >= N, !.  
max([X|T],N) :- max(T,N).
```

Tail recursive (iterative) version:

```
max([X|T],M) :- max(T,X,M).  
max([],M,M) :- !.  
max([H|T],MT,M) :- H > MT, !, max(T,H,M).  
max([H|T],MT,M) :- max(T,MT,M).
```



Exercise – Solution – Mind the CUT!!!!

```
max( [X] ,X) :- ! .  
max( [X|T] ,X) :- max(T,N) ,X>=N , ! .  
max( [X|T] ,N) :- max(T,N) .
```

Is it correct?

- Yes, if the second parameter is variable
- No, if the second parameter is ground... try:
`max([1,2,4,3] ,3) .`
- Why?



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Exercise

- Given a list L1 and a number N, write a predicate domanda1(L1, N, L2) that return in L2 the list of elements of L1 that are **lists**, and that **contain only two positive integers in the range 1..9**, whose sum amounts to N.

```
:- domanda1 (  
    [ 3,1] , 5, [2,1,1] , [3] , [1,1,1] , a,[2,2] 1,  
    4,  
    L2) .  
yes, L2 = [[3,1], [2,2]]
```



Exercise – Solution

```
domanda1([], _, []).  
domanda1([ [A,B] | R ], N, [ [A,B] | S ] ) :-  
    A >= 1, A = <9, B >= 1, B = <9,  
    N is A + B, !,  
    domanda1( R, N, S ).  
domanda1( _ | R ), N, S ) :- domanda1( R, N, S ).
```





ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Prolog – Meta-predicates

Prof. Ing. Federico Chesani

DISI

Department of Informatics – Science and Engineering

Meta-predicates

- In Prolog predicates (i.e., programs) and terms (i.e., data) have the same syntactical structure...
- ... as a consequence, predicates and terms can be exchanged and exploited in different roles!!!
- There are several pre-defined predicates that allows to deal with these structures, and work with them



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The `call` predicate

- If terms are predicates:
 - I could “prepare/create” a term in a “proper” way...
 - ... and then... EXECUTE IT!
- **call (T)** : the term **T** is considered as a atom (a predicate), and the Prolog interpreter is requested to evaluate (execute) it.
- Obviously, at the moment of the evaluation, **T** must be a non-numeric term
 - A number is a constant, and cannot be “evaluated” in logical sense



The `call` predicate

- The predicate `call` is considered to be a meta-predicate since:
 - its evaluation directly interfere with the Prolog interpreter underneath, within the same evaluation instance
 - It directly “alterates” the program.
- The predicate `call` takes as input a term that can be interpreted as a predicate:

```
p(a).
```

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

```
:- call(q(Y)).
```

```
yes Y = a.
```

When executed, `call` asks to the Prolog interpreter of proving `q(Y)`



The call predicate

- The predicate call can be used also within programs:

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

```
q(a).
```

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

```
yes Y = a.
```

- Some Prolog interpreters allow the following notation

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

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



The call predicate – Example

- Define a program that behaves as a procedural if_then_else construct

if_then_else(Cond,Goal1,Goal2)

- If Cond is evaluated to true, then Goal1 is evaluated
- If Cond is evaluated to false, then Goal2, is evaluated

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

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



The `fail` predicate

- `fail` takes no arguments (its arity is zero)
- Its evaluation always fails!
- As a consequence, it forces the interpreter to explore other alternatives...
- ... in other words, it explicitly activates the backtracking
- Why on the earth should we force the failure of a proof?
 - To obtain some form of iteration over data;
 - To implement the negation as failure;
 - To implement a logical implication



The `fail` predicate – the iteration

- Let us consider a KB with facts `p/1`. Apply a predicate `q(X)` on all `X` that satisfy `p(X)`.
- A possible solution (not the only one) :

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

```
iterate.
```

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



The `fail` predicate – the negation as failure

- Implement the negation as failure through a predicate `not(P)`
- `not(P)` is true if `P` is not a consequence of the program

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



Combining `fail` and `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, 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).
```



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.
- Sometimes, it might be interested to answer to queries like "**which is the set S of element x such that `p(x)` is true?**" (second-order query)
- Some Prolog interpreters support this type of second-order queries by providing pre-definite predicates.



Predicates `setof` and `bagof`

- **`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, `setof` should not contain repetitions. Not always true: it depends by the implementation.



Predicates `setof` and `bagof` – examples

Knowledge Base:

`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`

NOTICE: variable X, at the end, has not been unified with any value.



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Predicates `setof` and `bagof` – examples

Parameters are reversible. Knowledge Base:

`p(1) .`

`p(2) .`

`p(0) .`

`p(1) .`

`q(2) .`

`r(7) .`

`:– setof(X, p(X), [0,1,2]) .`

`yes X = X`

`:– bagof(X, p(X), [1,2,0,1]) .`

`yes X = X`



Predicates `setof` and `bagof` – examples

Conjunction of goals. Knowledge Base:

`p(1) .`

`p(2) .`

`p(0) .`

`p(1) .`

`q(2) .`

`r(7) .`

`: - setof(X, (p(X), q(X)), S) .`

`yes S = [2]`

`X = X`

`: - bagof(X, (p(X), q(X)), S) .`

`yes S = [2]`

`X = X`

`: - setof(X, (p(X), r(X)), S) .`

`no`

`: - bagof(X, (p(X), r(X)), S) .`

`no`



Predicates `setof` and `bagof` – examples

Non existing goals. Knowledge Base:

`p(1) .`

`p(2) .`

`p(0) .`

`p(1) .`

`q(2) .`

`r(7) .`

`:– setof(X, s(X), S) .`

`no`

`:– bagof(X, s(X), S) .`

`no`

*NOTICE: this is the expected behaviour...
however, some interpreters return an error
calling an undefined procedure
`s(X)`*



Predicates `setof` and `bagof` – examples

Complex terms in the answer. Knowledge Base:

`p(1)` .

`p(2)` .

`p(0)` .

`p(1)` .

`q(2)` .

`r(7)` .

```
: - setof(p(X) , p(X) , S) .
```

```
yes S=[p(0) ,p(1) ,p(2) ]
```

X=X

```
: - bagof(p(X) , p(X) , S) .
```

```
yes S=[p(1) ,p(2) ,p(0) ,p(1) ]
```

X=X



Predicates `setof` and `bagof` -- quantification of variables

- Knowledge base:

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

I was expecting all the X for which father(X,Y) is true...

INSTEAD, it returned those X for which, for the same value of Y, father(X,Y) is true.

?????

```
:- 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



Predicates `setof` and `bagof` -- quantification of variables

- Knowledge base:

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

We need to specify that Y has to be quantified existentially...

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



Predicates `setof` and `bagof` -- quantification of variables

- Compound terms in the answer. Knowledge base:

```
father(giovanni,mario).
```

```
father(giovanni,giuseppe).
```

```
father(mario, paola).
```

```
father(mario,aldo).
```

```
father(giuseppe,maria).
```

```
:– setof( (X,Y) , father(X,Y) , S ).
```

```
yes S=[ (giovanni,mario) , (giovanni,giuseppe) ,  
        (mario, paola) , (mario,aldo) ,  
        (giuseppe,maria) ]
```

X=X

Y=Y



Predicate **findall**

- Often, we are interested in setof and bagof, with the semantics of the variables not appearing in the first argument being quantified existentially...

findall(X, P, S)

- true if S is the list of instance X (without repetitions) for which predicate P is true.
- If there is no X satisfying P, then findall returns an empty list.



Predicate `findall` – example

- Knowledge base:

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

```
:– findall(X, father(X,Y) , S) .  
yes S=[giovanni, mario, giuseppe]  
X=X  
Y=Y
```

Equivalent to:

```
:– setof(X, Y^father(X,Y) , S) .
```



Bagof, setof, and findall are not limited to facts

- Predicates **setof**, **bagof** e **findall** works also when the property to be verified is not a simple fact, but it is defined by rules.

```
p(X,Y) :- q(X), r(X).
```

```
q(0).
```

```
q(1).
```

```
r(0).
```

```
r(2).
```

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

```
yes S=[0]
```

```
X=X
```

```
Y=Y
```



Verification of implication through setof

- Let us have predicates of the type **father(X,Y)** and **employee(Y)**
- We want to verify if it is true that for every **Y** for which **father(p,Y)** holds, then **Y** is an employee (all the sons of p are employees)
 $\text{father}(p,Y) \rightarrow \text{employee}(Y)$

```
imply(Y) :- setof(X, father(Y,X), L), verify(L).
```

```
verify([]).
```

```
verify([H|T]) :- employee(H), verify(T).
```



Iteration through setof

- Execute the procedure q on each element for which p is true

```
iterate:- setof(X, p(X), L),  
          filter(L).  
  
filter([]).  
  
filter([H|T]) :- call(q(H)),  
                  filter(T).
```

- Which difference with the implementation made through fail? What about backtrackability?



Verifying properties of terms

- **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.



Accessing the structure of a term

- Term =.. List
[SWI documentation]
 - 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))
```



Accessing the clauses of a program

- In Prolog, terms and predicates are represented with the same structure
- In particular, a clause (a query) is represented as a term
- Example: given
`h.`
`h :- b1, b2, ..., bn.`
- They correspond to the terms:
`(h, true)`
`(h, ' ', ' (b1, ' , ' (b2, ' , ' (... ', ' (bn-1, bn) ...)))`



Accessing the clauses of a program – the predicate clause

- **clause (Head, Body)**
true if (Head, Body) is unified with a clause stored within the database program
- When evaluated
 - Head must be instantiated to a non-numeric term
 - Body can be a variable or a term describing the body of a clause
- Its evaluation opens choice points, if more clauses with the same head are available



The predicate clause – example

- Program:

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

```
?- clause(p(1),BODY).  
      yes      BODY=true  
?- clause(p(X),true).  
      yes      X=1  
?- clause(q(X,Y), BODY).  
      yes      X=_1      Y=a      BODY=p(_1),r(a);  
                  X=2      Y=_2      BODY=d(_2);  
      no  
?- clause(HEAD,true).  
      Error - invalid key to data-base
```



Other amenities... Loading modules and libraries

- Modern Prolog interpreters come equipped with a huge library of code.
- To load a library:
`use_module(library(XXX)).`
- Example (SWI Prolog):
`: - use_module(library(lists)).`
Load the pre-defined predicates for dealing with lists
- `library.aggregate)`
- `library(ansi_term)`
- `library(apply)`
- `library(assoc)`
- `library(broadcast)`
- `library(charsio)`
- `library(check)`



Other amenities... Loading modules

- library(**clpb**)
- library(**clpfd**)
- library(**clpqr**)
- library(csv)
- library(**dccgbasics**)
- library(dcghighorder)
- library(debug)
- library(dicts)
- library(error)
- library(explain)
- library(help)
- library(intercept)
- library(summaries.d/intercept.tex)
- library(iostream)
- library(lists)
- library(main)
- library(occurs)
- library(option)
- library(optparse)
- library(ordsets)
- library(persistency)
- library(predicate_options)
- library(prologjiti)
- library(prologpack)
- library(prologxref)
- library(pairs)
- library(pio)
- library(random)
- library(readutil)
- library(record)
- library(registry)
- library(settings)
- library(simplex)
- library(ugraphs)
- library(url)
- library(www_browser)
- library(solution_sequences)
- library(thread)
- library(thread_pool)
- library(varnumbers)
- library(yall)



Other amenities... packages

- SWI-Prolog Semantic Web Library 3.0
- Constraint Query Language A high level interface to SQL databases
- SWI-Prolog binding to GNU readline
- SWI-Prolog ODBC Interface
- SWI-Prolog binding to libarchive
- Transparent Inter-Process Communications (TIPC) libraries
- JPL: A bidirectional Prolog/Java interface
- Pengines: Web Logic Programming Made Easy
- SWI-Prolog SSL Interface
- Google's Protocol Buffers Library
- SWI-Prolog Natural Language Processing Primitives
- Prolog Unit Tests
- SWI-Prolog Unicode library
- SWI-Prolog YAML library
- SWI-Prolog HTTP support



Other amenities... packages

- SWI-Prolog Regular Expression library
- Managing external tables for SWI-Prolog
- A C++ interface to SWI-Prolog
- SWI-Prolog SGML/XML parser
- SWI-Prolog binding to zlib
- Paxos -- a SWI-Prolog replicating key-value store
- SWI-Prolog Source Documentation Version 2
- SWI-Prolog C-library
- SWI-Prolog binding to BSD libedit
- SWI-Prolog RDF parser



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Prolog Meta-Interpreters

Prof. Ing. Federico Chesani

DISI

Department of Informatics – Science and Engineering

The starting point

- In Prolog, no difference between programs and data, or, using the Prolog terminology, ...
- ...No difference (in the representation) between predicates and terms.
- We can ask the Prolog interpreter to provide us the clauses of the (currently loaded) program
 - `clause(Head, Body)` .
- We can also ask the interpreter to "execute" a term
 - `call(T)` .



Meta-interpreters

- Meta-interpreters as meta-programs, i.e., programs who execute/works/deal with other programs
 - Programs as input of meta-programs
- Used for rapid prototyping of interpreters of symbolic languages
- In Prolog, a meta-interpreter for a language L is defined as an interpreter for L, but written in Prolog
- Given the premises, would it be possible to write a Prolog interpreter for the language Prolog?



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) :- !.
```

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

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

- Notice: it does not deal with pre-defined predicates...
 - For each pre-defined predicate, needs to add a specific `solve` clause that deal with it



Meta-interpreter for Pure Prolog aka the vanilla meta-interpreter

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

- Notice: no need to "call" any predicate. The vanilla meta-interpreter explores the current program, searching for the clauses, until it can prove the goal, or it fails.
- As it is, the vanilla meta-interpreter mimic the standard behaviour of the Prolog interpreter...
- ... but now, we can modify it and get different behaviours



Meta-interpreters for Pure Prolog – Example

Right-most selection rule

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

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

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



Meta-interpreters for Prolog – Example

Define a Prolog interpreter **solve(Goal, Step)** that:

- It is true if **Goal** can be proved
- In case **Goal** is proved, **Step** is the number of resolution steps used to prove the goal
 - In case of conjunctions, the number of steps is defined as the sum of the steps needed for each atomic conjunct



Meta-interpreters for Prolog – Example

Given the program:

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

```
b :- d.
```

```
c.
```

```
d.
```

```
?- solve(a, Step)
```

```
yes   Step=4
```

Why?



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Meta-interpreters for Prolog – Example

Define a Prolog interpreter **solve(Goal, Step)** that:

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



Meta-interpreters for Prolog – 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).

Example:

```
rule(a, (b, c) , 10) .  
rule(b, true, 100) .  
rule(c, true, 50) .
```



Meta-interpreters for Prolog – Example

- Define a meta-interpreter **solve(Goal, CF)**, that is true if **Goal** can be proved, with certainty **CF**.
- For conjunctions, the certainty is the minimum of the certainties of the conjuncts
- For rules, the certainty is the product of the certainty of the rule itself times the certainty of the proof of the body (eventually divided by 100).



Meta-interpreters for Prolog – Example

```
rule(a, (b,c), 10) .
```

```
rule(a, d, 90) .
```

```
rule(b,true, 100) .
```

```
rule(c,true, 50) .
```

```
rule(d,true, 100) .
```

```
?-solve(a,CF) .
```

```
yes CF=5;
```

```
yes CF=90
```



Meta-interpreters for Prolog – Example

```
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) .
```



Meta-interpreters – a simple Expert System

- 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).
```

ERROR: Undefined procedure: has_feathers/1

- Does it mean that we do not know if `tweety` has feathers?
- Does it mean that we do not know anything about the concept "having feathers"?



Meta-interpreters – a simple Expert System

- Idea: extend your **KB** and/or **reasoning tool**, so that:
 - It tries to prove a Goal using the given KB
 - If it fails, the reasoner could also ask help to the user

(alternative) Solutions:

1. Modify our knowledge base
2. [Implement a new/extend existing] reasoning tool

Example taken from:

https://swish.swi-prolog.org/example/expert_system.pl



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Meta-interpreters – a simple Expert System

- Idea: extend your **KB** and/or **reasoning tool**, so that:
 - It tries to prove a Goal using the given KB
 - If it fails, the reasoner could also ask help to the user

```
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), get_code(_),  
    Answer = yes.
```

=/2 tries to unify the two arguments. If it succeeds, the two arguments are unified later.

`get_code(_)` is needed due to the implemented behaviour of `read/1`, that leaves a char in the buffer. Notice: it behaves differently on windows, and on the swish web app.



Meta-interpreters – a simple Expert System

- But... asked predicate can be asked in this way...
cannot we limit the user question to a user-specified list of predicates?

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

```
% Only askable predicates can be  
asked to the user  
  
askable(tweets(_)).  
askable(small(_)).  
askable(cuddly(_)).  
askable(has_feathers(_)).  
  
%%%  
% The KB  
  
good_pet(X) :-  
    bird(X), small(X).  
  
good_pet(X) :-  
    cuddly(X), yellow(X).  
bird(X) :-  
    has_feathers(X), tweets(X).  
  
yellow(tweety).
```



Meta-interpreters – Exercise

- 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)
```



Meta-interpreters – Exercise – Solution

- The solution is obtained by simply modifying the meta interpreter vanilla

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



Meta-interpreters – Exercise (variation)

- Write a meta-interpreter for the Prolog language that prints out, before and after the execution of a subgoal, the subgoal itself. Subgoals should also be "tabbed" on the right depending on the depth of the resolution tree. Example:

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

```
q(1).
```

```
q(2).
```

```
?- s(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)
```



Meta-interpreters – Exercise (variation) – Solution

```
s(true, N) :- !.  
s((A,B), N) :- !, s(A, N), s(B, N).  
s(A, N) :-  
    tt(N), write('Solving: '), write(A), nl,  
    clause(A,B),  
    N1 is N+1,  
    tt(N1), write('Selected Rule: '), write(A), write(":-"), write(B), nl,  
    s(B,N1),  
    tt(N1), write('Solved: '), write(B), nl.  
  
tt(0).  
tt(N) :-  
    N>0,  
    tab(3),  
    N1 is N-1,  
    tt(N1).
```



Dynamically modifying the program

- When a Prolog program is consulted/loaded, its representation in terms of data structures (terms) is loaded into a table in memory
- Such table is often referred as the program database
 - Indeed, it is managed using DBMS techniques for increasing performances
 - For example, functors of the heads are indexed, to speed-up the search for possible candidates for unification with a goal
- If it is a table, can we change it?
 - Add entries to the table?
Means adding new clauses for a predicate
In procedural terms, it would be like adding new methods
 - Remove entries from the table?



Dynamically modifying the program – assert

assert(T)

Clause **T** is added to the database program.

- 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
 - Non declarative behaviour
 - Added clauses are not removed by backtracking
- For efficiency reasons, functors of predicates that will be added must be declared as "dynamic":
 : - **dynamic(foo/1)** .



Dynamically modifying the program – assert

assert(T)

Clause **T** is added to the database program.

- However, the order of the clause definitions in Prolog does have a (important!) meaning
- Two variations available:
 - **asserta(T)**
T is added at the beginning of the database
 - **assertz(T)**
T is added ad the end of the database
- The behaviour can greatly change...



Dynamically modifying the program – assert

```
?- assert(a(2)).
```

```
?-dynamic(a/1).  
a(1).  
b(X) :- a(X).
```

```
?- asserta(a(3)).
```

```
a(1).  
a(2).  
b(X) :- a(X).
```

```
?- assertz(a(4)).
```

```
a(3).  
a(1).  
a(2).  
b(X) :- a(X).
```

```
a(3).  
a(1).  
a(2).  
a(4).  
b(X) :- a(X).
```



Dynamically modifying the program – **retract**

retract(T)

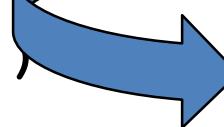
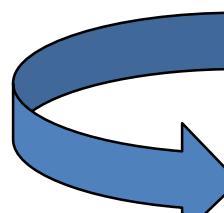
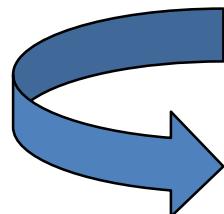
The first clause in the database that unifies with **T** is removed.

- When evaluated, **T** should be instantiated to a term denoting a clause
- If more than one clauses unify with **T**, the first one is removed; some Prolog implementations keep tracks with a backtrackable choice point.
- Some Prolog implementations provide the predicate **abolish/retract_all**, that remove all the occurrences of the specified **term** with **arity**



Dynamically modifying the program – retract

```
?- retract(a(X)).  
yes X=3  
  
?- abolish(a,1).  
  
?- retract( (b(X):-BODY) ).  
yes BODY=c(X),a(X)
```



```
?-dynamic(a/1).  
?-dynamic(b/1).  
a(3).  
a(1).  
a(2).  
a(4).  
b(X) :- c(X), a(X).
```

```
a(1).  
a(2).  
a(4).  
b(X) :- c(X), a(X).
```

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



Dynamically modifying the program – retract

```
?- retract(a(X)).  
yes X=3;
```

```
yes X=1;
```

```
yes X=2;
```

```
yes X=4;
```

```
no
```

```
a(3).  
a(1).  
a(2).  
a(4).  
b(X) :- c(X), a(X).
```

```
a(1).  
a(2).  
a(4).  
b(X) :- c(X), a(X).
```

```
a(2).  
a(4).  
b(X) :- c(X), a(X).
```

```
a(4).  
b(X) :- c(X), a(X).
```

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



assert and retract – few issues...

- When using assert and retract, the declarative semantics of Prolog is lost
- Consider an empty program, and the following queries:

```
?- assert(p(a)), p(a).  
?- p(a), assert(p(a)).
```
- The first query succeeds; the second query fails.
- The order of the literals plays a fundamental role (but the same holds for the cut, for the negation with unbound variables, etc. etc.)



assert and retract – few issues...

Another example:

a(1). (P1)

p(X) :- assert(b(X)), a(X).

a(1). (P2)

p(X) :- a(X), assert(b(X)).

The query `:- p(X).` produces the same answer,
but two different database modifications/

- in P1, **b(X)** is added to the database: $\forall X \ p(X)$
- in P2, **b(1)** only is added to the database.



assert and retract – few issues...

- A further problem is about the quantification of variables
- Variables in clauses are quantified universally...
- Variables in queries are quantified existentially.

Consider the query :- **assert((p(X))) .**

- **X** is existentially quantified.
- However, the database is extended with the clause
p(X) .
- Formula: $\forall X \ p(X)$



assert and retract – example: the lemma generation

- Simple recursive solutions for computing Fibonacci are usually very inefficient

```
% fib(N,Y) "Y is the Nth Fibonacci number"
```

```
fib(0,0) :- !.  
fib(1,1) :- !.  
fib(N,Y) :- N1 is N-1, fib(N1,Y1),  
           N2 is N-2, fib(N2,Y2),  
           Y is Y1+Y2.,  
           generate_lemma(fib(N,Y)).
```



assert and retract – example: the lemma generation

```
genera_lemma (T) :- asserta(T) .
```

- Alternatively:

```
genera_lemma (T) :- clause(T,true), !.
```

```
genera_lemma (T) :- asserta(T) .
```

- The second solution checks that the same lemma is not added multiple times to the database





ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Prolog – Definite Clause Grammars, CLP, CHR

Prof. Ing. Federico Chesani

DISI

Department of Informatics – Science and Engineering

Formal languages and their grammars

- The syntax of a language is formally defined through grammar, typically a set of rules.
- A grammar allows to establish if a sequence of symbols (belonging to a defined, finite alphabet) is indeed a sentence of the language or not.
- Usually:
 - ε stands for the empty string
 - X^+ stands for a non-empty string that can be obtained from X
 - $X^* = X^+ \cup \{\varepsilon\}$



Grammars

- A grammar is defined as a tuple $G=\{V_n, V_t, P, S\}$
 - V_n is the set of **non-terminal symbols**
 - V_t is the set of **terminal symbols**
 - P is the set of **production rules** (or rewriting rules):
 $\alpha ::= \beta$ or $\alpha \rightarrow \beta$
dove $\alpha \in (V_n \cup V_t)^+$ $\beta \in (V_n \cup V_t)^*$
 - S is the **start symbol** (the initial symbol) of the grammar
- Given a production rule $\alpha \rightarrow \beta$, from a sentence of the form $\mu\alpha\nu$ we can **derive (in one step)** $\mu\beta\nu$, for every $\mu, \nu \in (V_n \cup V_t)^*$



Grammars

- The notion of derivability can be extended by considering the reflexive and transitive closures of the relation \rightarrow
- $\alpha \rightarrow^* \alpha$
- $\alpha \rightarrow^* \beta$ if $\alpha \rightarrow \gamma$ and $\gamma \rightarrow^* \beta$
- The language $L(G)$ defined by the grammar G is the set of sentences of terminal symbols that can be derived starting from the initial symbol of the grammar

$$L(G) = \{\alpha \mid \alpha \in V_t \text{ and } S \rightarrow^* \alpha\}$$



Grammars – example in the EBNF syntax

- $G = (V_n, V_t, P, S)$
- $V_n = \{U, V, Z\}$
- $V_t = \{0, 1\}$
- $P = \{$

$Z ::= U0 \quad | \quad V1$

$U ::= Z1 \quad | \quad 1$

$V ::= Z0 \quad | \quad 0$

}

- $S = Z$

Symbol | stands for disjunction

Derivable sentences:

$\{01, 0101, 0110, 1010, 10, \dots\}$



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Grammars – why they matter

- Modern programming languages are defined by their grammar
- Few examples:
 - Python grammar:
<https://docs.python.org/3/reference/grammar.html>
 - C Language Grammar, as supported by Microsoft in its development tools: <https://docs.microsoft.com/en-us/cpp/c-language/summary-of-statements?view=vs-2019>



Types of grammars

- Type 0: **recursively enumerable**
 - Strings appearing in production rules $\alpha \rightarrow \beta$ do not have any constraint (but with α and β not empty)
- Type 1: **Context-dependent**
 - Production rules have the form $\alpha A \beta \rightarrow \alpha \pi \beta$, where A is a non-terminal, and π is not empty
- Type 2: **Context-free**
 - Production rules have the form $A \rightarrow \pi$, where A is a non-terminal, and π is not empty
- Type 3: **Regular (finite states)**
 - Production rules have only the form $A \rightarrow a$ or $A \rightarrow aB$ or $A \rightarrow Ba$, where A e B are non terminal symbols, and a is a terminal symbol
 - Grammars with only rules $A \rightarrow aB$ are linear, right
 - Grammars with only rules $A \rightarrow Ba$ are linear, left



Syntactic analysis

- Given a grammar G , the task of verifying if a string α belongs to the language $L(G)$ is called **syntactic analysis** or **parsing** of α
- A simple case is the one of regular languages (i.e., derived from type 3 grammars): such analysis is called **lexical analysis**, and can be solved through a finite-state automata
- Unfortunately, regular grammars usually generate languages that are not very interesting...



Syntactic analysis

- A far more interesting class of grammars is the Type 2, i.e. the context-free grammars
 - Programming language grammars belong to this class
- The syntactic analysis of a context-free grammar is a decidable problem
- It is always possible to:
 - translate a context-free grammar in a set of Prolog clauses
 - exploit resolution to perform the syntactic analysis



Syntactic analysis of (sentences of languages of) Type 2 grammars

- How? Several methods; a class of methods is named top-down
- They starts from the initial symbol
- Exploit the production rules as rewriting rules
- Succeed if they can derive the string

Two methods in this class:

- Stack-based automaton
- Recursive descent (from the initial symbol)

Notice: top-down methods suffer **infinite loops** when dealing with left-recursive rules; proper transformation is required before applying them.



Recursive Descent method

- Terminal symbols are represented by means of predicates
- Non-terminal symbols are represented by means of predicates as well (hence, no distinction with terminals)
- Production rules are represented through clauses
- Each predicate has two arguments, a list containing the symbols of the string to be parsed; and the output list of the "remaining symbols"
- The output list is a sublist of the input list where:
 - Elements derived from production rules about a non-terminal X have been removed
 - X is removed as well, if it is a terminal symbol



Recursive Descent – example

$$G = (V_n, V_t, P, S)$$

$$V_n = \{E, T\}$$

$$V_t = \{ +, a \}$$

$$P = \{$$

$$E ::= T + E \mid T$$

$$T ::= a$$

}

$$S = E$$

```
e(ListIn, ListOut) :-  
    t(ListIn, ListTemp),  
    plus(ListTemp, ListTemp1),  
    e(ListTemp1, ListOut).  
  
e(ListIn, ListOut) :-  
    t(ListIn, ListOut).  
  
t(ListIn, ListOut) :-  
    a(ListIn, ListOut).  
  
a([a| List], List).  
plus([+| List], List).  
  
% Queries:  
:- e([a, +, a, +, a], []).  
yes  
:- e([a, +, +, a], []).  
no
```



Definite Clause Grammars – DCG

- The Definite Clause Grammars are an extension of free-context grammars
- DCG grammars can be directly and automatically translated into the corresponding Prolog programs

How?

- In input to the DCG module, each production rule is presented as a fact, with functor "`-->`" (a binary operator with infix notation)
- Non-terminal symbols are represented through atoms
- Terminal symbols are represented through atoms in square brackets [...], or through strings "..."
- Empty string is denoted through []



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DCG – Example

$$G = (V_n, V_t, P, S)$$

$$V_n = \{E, T\}$$

$$V_t = \{ +, a \}$$

$$P = \{$$

$$E ::= T + E \mid T$$

$$T ::= a$$

}

$$S = E$$

```
e --> t, plus, e.
```

```
e --> t.
```

```
t --> [a].
```

```
plus --> [+].
```

```
% Query:
```

```
?- phrase(e, [a,+ ,a]).  
true .
```

```
?- phrase(e, [a,+ ,+ ,a]).  
false.
```

`phrase(DCGBody, List) : true when DCGBody applies to List.`



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DCG – an example for natural language

sentence --> **nominalSent**, **verbalSent**.

nominalSent --> **article**, **name**, **relativeSent**.

verbalSent --> **verb**, **nominalSent**.

relativeSent --> [] .

relativSent --> [that], **verbalSent**.

article --> [the] .

article --> [a] .

name --> [dog] .

name --> [cat] .

verb --> [loves] .

verb --> [eats] .



DCG – Extension

- As it is, DCG parse the sentence, and determines if it belongs to the language or not.
- To do so, it implicitly construct the parse tree (indeed, it would "match" the SLD branch of success)
- *Why not “grab” the parse tree and reuse it?*
- DCGs non-terminals can have arguments, that will be used as output parameters
- The user specify how these parameters are constructed. i.e. we have fully control of the abstract parse tree construction process
- Moreover, predicates can be called in the right part of the rules...



DCG – Example – parsing integer expressions...

$G = (V_n, V_t, P, S)$

$V_n = \{E, T\}$

$V_t = \{ +, \{\text{digit}\} \}$

$P = \{$

$E ::= T "+" E$

$E ::= T "-" E$

$E ::= T$

$T ::= \text{factor} "*" T$

$T ::= \text{factor} "/" T$

$T ::= \text{factor}$

$\text{factor} ::= \text{digit}$

$\text{factor} ::= "(" E ")"$

}

$S = E$

```
e (plus(Op1,Op2)) --> t(Op1), plus_sign, e(Op2) .  
e (minus(Op1,Op2)) --> t(Op1), minus_sign, e(Op2) .  
e (Op) --> t(Op) .  
  
t(mult(Op1, Op2)) --> factor(Op1), mult_sign, t(Op2) .  
t(div(Op1, Op2)) --> factor(Op1), div_sign, t(Op2) .  
t(Op) --> factor(Op) .  
  
factor(X) --> digit(Code), {number_string(X,[Code])} .  
% factor ::= "(" E ")"  
  
digit(Digit) --> [Digit], {  
    char_code('0',Zero),  
    char_code('9',Nine),  
    Zero =< Digit,  
    Digit =< Nine} .  
  
plus_sign --> "+".  
minus_sign --> "-".  
mult_sign --> "*".  
div_sign --> "/".
```



DCG – Example – parsing integer expressions...

```
e(plus(Op1,Op2)) --> t(Op1), plus_sign, e(Op2).
e(minus(Op1,Op2)) --> t(Op1), minus_sign, e(Op2).
e(Op) --> t(Op).

t(mult(Op1, Op2)) --> factor(Op1), mult_sign, t(Op2).
t(div(Op1, Op2)) --> factor(Op1), div_sign, t(Op2).
t(Op) --> factor(Op).

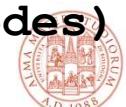
factor(X) --> digit(Code), {number_string(X,[Code])}.
% factor ::= "(" E ")"
digit(Digit) --> [Digit], {
    char_code('0',Zero),
    char_code('9',Nine),
    Zero =< Digit,
    Digit =< Nine}.

plus_sign --> "+".
minus_sign --> "-".
mult_sign --> "*".
div_sign --> "/".
```

```
?- string_codes("1", Codes), phrase(e(X), Codes).
Codes = [49],
X = 1.
```

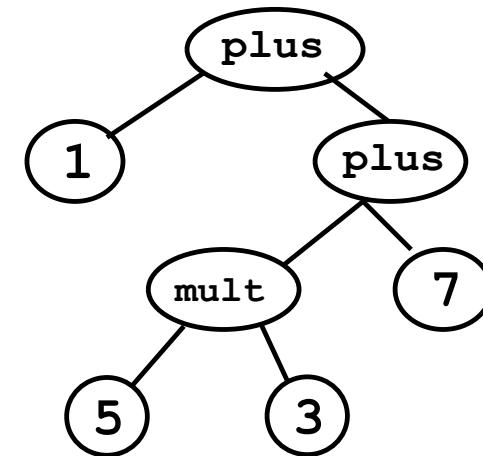
```
?- string_codes("1+5", Codes), phrase(e(X), Codes).
Codes = [49, 43, 53],
X = plus(1, 5)
```

```
?- make, string_codes("1+5*3+7", Codes), phrase(e(X), Codes).
Codes = [49, 43, 53, 42, 51, 43, 55],
X = plus(1, plus(mult(5, 3), 7))
```



DCG – Example – parsing integer expressions...

```
?- make, string_codes("1+5*3+7", Codes), phrase(e(X), Codes).  
Codes = [49, 43, 53, 42, 51, 43, 55],  
X = plus(1, plus(mult(5, 3), 7))
```



- In the example above, X has been unified to a structured term...
- ... following our specification in the DCG rules.
- The next step... why not construct a small calculator/evaluator?



DCG – Example – evaluating integer expressions...

```
eval(plus(A,B) , Result) :-  
    eval(A,AResult) , eval(B,BResult) ,  
    Result is AResult+BResult.  
  
eval(minus(A,B) , Result) :-  
    eval(A,AResult) , eval(B,BResult) ,  
    Result is AResult-BResult.  
  
eval(mult(A,B) , Result) :-  
    eval(A,AResult) , eval(B,BResult) ,  
    Result is AResult*BResult.  
  
eval(div(A,B) , Result) :-  
    eval(A,AResult) , eval(B,BResult) ,  
    Result is AResult/BResult.  
  
eval(A,A) :- number(A) .
```

```
?- string_codes("1+5*3+7", Codes), phrase(e(X), Codes), eval(X, Result).  
Codes = [49, 43, 53, 42, 51, 43, 55],  
X = plus(1, plus(mult(5, 3), 7)),  
Result = 23
```



DCG – Example – evaluating integer expressions... ... and some very simple user interface

```
start :-  
    write('Please type an expression, or \'quit\' to leave the  
calculator...'), nl  
    % read_string(+Stream, +SepChars, +PadChars, -Sep, -String)  
    , read_string(user_input, "\n", "\r", _End, String)  
    , proceed(String).  
  
proceed(String) :-  
    String = "quit", !.  
proceed(String) :-  
    string_codes(String, Codes)  
    , phrase(e(X), Codes)  
    , !  
    , eval(X, Result)  
    , write('Result: '), write(Result), nl  
    , start.  
proceed(_String) :-  
    write('Sorry, can\'t understand the expression.'), nl  
    , start.
```



DCG and Prolog to build compilers...

Quick considerations...

- Nowadays, programming language grammars are defined through the EBNF syntax.
- Several tools exists for building up parsers, and get as output the abstract parse tree
 - Available for any modern language
 - Very efficient tools
- On the other side, Prolog is not the most “efficient” programming language out there...
- ... however, it allows a very **fast prototyping** for both the parsing, as well as the compiling steps...
- ... within a declarative approach, making Prolog one of the best possible choices for small and quick projects.



CLP: Constraint Logic Programming



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Let us recall the N-Queens problem

Problem: put n queens on a chessboard so that they do not attack each other.



Currently tackled:

- by Prof. Milano in the lesson about Constraint problems (Module 1 of this course)
- by Prof. Kiziltan in these days



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The N-Queens Problem

Several ways for solving it

- A posteriori algorithms
 - Generate and Test
 - Standard Backtracking
- Propagation-based algorithms
 - Forward Checking
 - (Partial and Full) Look Ahead
- Consistency-based techniques
 - Node-consistency
 - Arc-consistency
 - Path- and K-consistency
 - MAC (AC-3...)



N-Queens and "generate and test"

- How to represent the problem in Prolog?
 - List of 8 variables
 - The first represents the position of the queen in the first column, the second var the pos. of the queen in the second column, etc. etc.
- Let us suppose to already have a predicate:
safe(S)
that is true if is a list of positions of the queens, such that they do not attack each other
- Homework: develop the **safe(s)** predicate.



N-Queens and "generate and test"

The solution in Prolog:

```
solution(S) :- generate(S), safe(S).
```

```
% we have to make sure that generate/1 provides a  
different solution each time,
```

```
% otherwise we will not terminate
```

```
generate(S) :- permutation([1,2,3,4,5,6,7,8], S).
```

```
permutation([], []).
```

```
permutation([Head| Tail], Result) :-
```

```
    permutation(Tail, PermTail),
```

```
    delete1(Head, Result, PermTail).
```

```
delete1(Head, [Head|Tail], Tail).
```

```
delete1(E1, [Head|Tail], [Head|TailDeleted]) :-
```

```
    delete1(E1, Tail, TailDeleted).
```



N-Queens and "generate and test"

We are still missing the `safe/1` predicate

- If `s` is an empty list, then it is safe.
- If `s` is a list `[Queen | Others]`, then `s` is safe if:
 - `Others` is safe
 - `Queen` does not attack any of `Other`

```
safe([]).  
safe([Queen|Others]) :-  
    safe(Others), noattack(Queen, Others, 1).  
noattack(_, [], _).  
noattack(Y, [Y1|Ylist], Xdist) :-  
    Y - Y1 =\= Xdist  
    , Y1 - Y =\= Xdist  
    , Dist1 is Xdist + 1  
    , noattack(Y, Ylist, Dist1).
```



N-Queens and "Standard Bakctracking"

- Standard backtracking is even easier in Prolog

```
solution_stdback(X) :-  
    queens(X, [], [1,2,3,4,5,6,7,8]).  
  
queens([], _Placed, []).  
queens([X|Xs], Placed, Values) :-  
    delete1(X, Values, NewValues),  
    noattack(X, Placed, 1),  
    queens(Xs, [X|Placed], NewValues).
```



Few limits of Logic Programming

- Objects manipulated by logic programs are structures
 - They are not interpreted
 - Unification works on syntactical equivalence
- Being based on depth-first and chronological backtracking in the solution space, it is very easy, immediate, and natural to come up with solutions like "generate and test"



Constraint Logic Programming

- Proposed by Jaffar and Lassez in 1989, CLP is an extension of standard Prolog, where typical constructs of Constraint Programming have been ported in Prolog
- "Objects" can be subjected to properties, and primitives methods
 - Domains (real numbers, integers, rationals, booleans, finite domains)
 - Specific methods (constraints can be posted depending on the object types)
- Constraint algorithms can be exploited to perform a more "intelligent" search for a solution
- Optimization algorithms are supported as well



The schemes CLP(X)

- In the original proposal, there are different schemes depending on the domain type (and different algorithms as well)
- CLP(X), where X is a specified domain
 - Objects belonging to X have a different semantics (w.r.t. Prolog terms)
 - Special primitives are defined for these objects
 - Special meaning and, consequently, special behaviour
- For example $X+3=Y-2$
 - It is not resolved until one of the two variables are grounded.
 - Then, automatically the other variable is assigned the proper value (the constraint is propagated)
- The CLP(X) scheme is supported by many prolog interpreters; one of the best is the ECLIPSE CLP interpreter, ad-hoc optimized for CLP...



SWI CLP(B): CLP over boolean domains

- Based on reduced and ordered Binary Decision Diagrams (BDDs).

0	false
1	true
<i>variable</i>	unknown truth value
<i>atom</i>	universally quantified variable
$\sim Expr$	logical NOT
$Expr + Expr$	logical OR
$Expr * Expr$	logical AND
$Expr \# Expr$	exclusive OR
$Var \wedge Expr$	existential quantification
$Expr =:= Expr$	equality
$Expr =\backslash= Expr$	disequality (same as #)
$Expr =< Expr$	less or equal (implication)
$Expr =\geq Expr$	greater or equal
$Expr < Expr$	less than
$Expr > Expr$	greater than
$card(Is,Exprs)$	cardinality constraint
$+ (Exprs)$	n-fold disjunction
$* (Exprs)$	n-fold conjunction

sat (+Expr)

True iff the Boolean expression *Expr* is satisfiable.

taut (+Expr, -T)

If *Expr* is a tautology with respect to the posted constraints, succeeds with *T = 1*. If *Expr* cannot be satisfied, succeeds with *T = 0*. Otherwise, it fails.

labeling (+Vs)

Assigns truth values to the variables *Vs* such that all constraints are satisfied.

<https://www.swi-prolog.org/pldoc/man?section=clpb>



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SWI CLP(B): CLP over boolean domains

Examples

```
?- use_module(library(clpb)).  
true.
```

```
% is the formula (X and Y) satisfiable?  
?- sat(X*Y).  
X = Y, Y = 1.
```

```
% is the formula (X or not X) satisfiable? Is it a tautology?  
?- taut(X + ~X, T).  
T = 1,  
sat(X=:=X).
```



SWI CLP(FD): CLP over finite domains

Allows to work over integers...

Arithmetic constraints:

Expr1 #= Expr2	Expr1 equals Expr2
Expr1 #\= Expr2	Expr1 is not equal to Expr2
Expr1 #>= Expr2	Expr1 is greater than or equal to Expr2
Expr1 #<= Expr2	Expr1 is less than or equal to Expr2
Expr1 #> Expr2	Expr1 is greater than Expr2
Expr1 #< Expr2	Expr1 is less than Expr2

Membership constraints:

?Var in +Domain

Var is an element of Domain. Domain is one of:

- **Integer:** Singleton set consisting only of Integer.
- **Lower .. Upper:** All integers / such that Lower = \leq / = \leq Upper. Lower must be an integer or the atom **inf**, which denotes negative infinity. Upper must be an integer or the atom **sup**, which denotes positive infinity.
- **Domain1 \ Domain2:** The union of Domain1 and Domain2.

+Vars ins +Domain: Vars is a list of variables

Arithmetic Expressions:

integer	Given value
variable	Unknown integer
?(variable)	Unknown integer
-Expr	Unary minus
Expr + Expr	Addition
Expr * Expr	Multiplication
Expr - Expr	Subtraction
Expr ^ Expr	Exponentiation
min(Expr,Expr)	Minimum of two expressions
max(Expr,Expr)	Maximum of two expressions
Expr mod Expr	Modulo induced by floored division
Expr rem Expr	Modulo induced by truncated division
abs(Expr)	Absolute value
Expr // Expr	Truncated integer division
Expr div Expr	Floored integer division



SWI CLP(FD): CLP over finite domains

Other constraints:

- **Global constraints**
 - all distinct/all different, sum, scalar product, global cardinality, circuit, disjoint, etc. etc.
- **Reification constraints:** "Many CLP(FD) constraints can be reified."
 - The truth value of constraints is itself turned into a CLP(FD) variable
 - It is possible to explicitly reason about whether a constraint holds or not.
- **Reflection constraints:** allows to access the internals of the constraint store, and reason upon such information



SWI CLP(FD): CLP over finite domains

Enumeration predicates

indomain (`?Var`)

Bind **Var** to all feasible values of its domain on backtracking. The domain of **Var** must be finite.

label (+Vars)

Equivalent to labeling([], Vars). See labeling/2.



SWI CLP(FD): CLP over finite domains

Enumeration predicates

labeling(+Options, +Vars)

Assign a value to each variable in **Vars**. Labeling means systematically trying out values for the finite domain variables **Vars** until all of them are ground. The domain of each variable in **Vars** must be finite.

Options is a list of options that let you exhibit some control over the search process.

- Option about the variable selection strategy: which var is chosen to be labelled next?
 - **leftmost**, **ff** (First fail), **ffc**, **min**, **max**
- Value order options
 - **up**, **down**
- Branching strategy options
 - **step**, **enum**, **bisect**



SWI CLP(FD) – Example

Crypto arithmetic: solve the sum

$$\text{SEND} + \text{MORE} = \text{MONEY}$$

where different letters denote distinct integers between 0 and 9.

```
:  
- use_module(library(clpfd)).  
  
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-  
    Vars = [S,E,N,D,M,O,R,Y],  
    Vars ins 0..9,  
    all_different(Vars),  
    S*1000 + E*100 + N*10 + D +  
    M*1000 + O*100 + R*10 + E #=  
    M*10000 + O*1000 + N*100 + E*10 + Y,  
    M #\= 0, S #\= 0.
```

```
?- puzzle(As+Bs=Cs), label(As).  
  
As = [9, 5, 6, 7],  
Bs = [1, 0, 8, 5],  
Cs = [1, 0, 6, 5, 2] .
```



SWI CLP(FD) – Example

The N-Queens problem:

```
:-
    use_module(library(clpfd)).

n_queens(N, Qs) :-  
    length(Qs, N),  
    Qs ins 1..N,  
    safe_queens(Qs).

safe_queens([]).
safe_queens([Q|Qs]) :-  
    safe_queens(Qs, Q, 1),
    safe_queens(Qs).

safe_queens([], _, _).
safe_queens([Q|Qs], Q0, D0) :-  
    Q0 #\= Q,  
    abs(Q0 - Q) #\= D0,  
    D1 #= D0 + 1,  
    safe_queens(Qs, Q0, D1).
```

```
?- n_queens(8, Qs), label(Qs).  
Qs = [1, 5, 8, 6, 3, 7, 2, 4] .  
  
?- time((n_queens(90, Qs),  
labeling([ff]),  
% 6,702,216 inferences, 0.641 CPU in  
0.668 seconds (96% CPU, 10454701  
Lips)  
Qs = [1, 3, 5, 50, 42, 4, 49, 7,  
59|...] Qs))).  
  
% How many solutions for 8-queen?  
?- findall(Qs,  
(n_queens(8,Qs),label(Qs)),  
ResultList), length(ResultList,  
Result).  
  
ResultList = [[1, 5, 8, 6, 3, 7, 2,  
4], [1, 6, 8, 3, 7, 4, 2|...], [1,  
7, 4, 6, 8, 2|...], [1, 7, 5, 8,  
2|...], [2, 4, 6, 8|...], [2, 5,  
7|...], [2, 5|...], [2|...],  
[...|...|...]],  
Result = 92.
```



SWI CLP(FD) – Example

Taken from the Italian Facebook page of math puzzles solvers (they periodically organize competitions among themselves).

Fill the empty boxes with integers, so that:

- the total sum is 35,
- The sum of the three numbers on the left is 22
- The sum of the three numbers on the right is 25

How much is the product of the pink boxes?

Indovinello della settimana

Antonio deve inserire dei numeri nelle caselle della griglia qui sotto, per cui la somma di tutti i numeri sia 35, che la somma dei primi tre numeri (da sinistra) sia 22 e che la somma degli ultimi tre (a destra) sia 25. Qual è il prodotto dei numeri sulle caselle rosa?

3				4
---	--	--	--	---



SWI CLP(FD) – Example

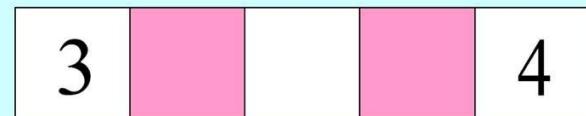
Taken from the Italian Facebook page of math puzzles solvers (they periodically organize competitions among themselves).

```
boxes(Result) :-  
    [A,B,C,D,E] ins 0..sup  
    , A #= 3  
    , E #= 4  
    , sum([A,B,C,D,E], #=, 35)  
    , sum([A,B,C], #=, 22)  
    , sum([C,D,E], #=, 25)  
    , Result #= B * D  
    , label([A,B,C,D,E]).
```

Indovinello della settimana

Antonio deve inserire dei numeri nelle caselle della griglia qui sotto, per cui la somma di tutti i numeri sia 35, che la somma dei primi tre numeri (da sinistra) sia 22 e che la somma degli ultimi tre (a destra) sia 25.

Qual è il prodotto dei numeri sulle caselle rosa?



CLP(FD) – How to program

A very common structure of CLP programs is the following:

1. Post constraints
 1. Domain
 2. Unary constraints
 3. Binary constraints
 4. Global constraints
2. Search for solution
 1. Label strategies



SWI CLP(FD) – something unexpected...

- Try the following query:

```
?- A#>B, B#>C, C#>A.
```

Output of the SWI interpreter:

```
A#=<C+ -1,
```

```
B#=<A+ -1,
```

```
C#=<B+ -1.
```

- It does not complain at all. The interpreter replies with the current constraints.
- There is no existing solution for the problem, however it returns success... why?



SWI CLP(FD) – something unexpected...

```
?- A#>B, B#>C, C#>A.
```

Output of the SWI interpreter:

```
A#=<C+ -1,
```

```
B#=<A+ -1,
```

```
C#=<B+ -1.
```

- For efficiency reasons, many implementations of constraint solvers do not implement k-consistency
- As a consequence, infeasibility is not detected until the actual labelling of the variables is requested:

```
?- A#>B, B#>C, C#>A, [A,B,C] ins 0..100000, label([A,B,C]).  
false.
```

NOTE: before replying false, it tries all the possible labelling of the variables...



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CHR: Constraint Handling Rules



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Flexibility of CLP

- CLP is very powerful, and the almost totality of CSP problems you will encounter, can be solved through already implemented constraints
- For performance reasons, usually CLP constraints and solvers are implemented "below" the Prolog level...
- ... constraints and strategies for solving are pre-defined in the chosen interpreter.
- What if we want to write down a novel constraint?



CHR – Constraint Handling Rules

- Introduced in 1991 by Thom Frühwirth
- It was thought as a simple language for developing constraint solvers

Main ideas:

- Constraints are represented through terms
- When constraints are specified over variables, they are stored into a constraint store
- The developer simply specifies, through rules, what happens when a constraint is posted into the store
 - Only three type of rules...



CHR – Constraint Handling Rules

Three types of rules:

- **Simplification**: the simplification rule removes the constraints in its head and calls its body.
head $\Leftarrow\Rightarrow$ **body**.
head is a list of constraints, **body** is a Prolog goal that is executed (it will contain novel constraints, rewritten constraints, but also usual Prolog code)
- It is executed when a constraint that matches the **head** is posted in the constraint store.

- Example: the "less or equal" constraint:
reflexivity @ **leq(X,X) $\Leftarrow\Rightarrow$ true**.

Effect: the constraint is removed



CHR – Constraint Handling Rules

Three types of rules:

- **Propagation:** The propagation rule calls its body exactly once for the constraints in its head.
head ==> body.
head is a list of constraints, **body** is a Prolog goal that is executed (it will contain novel constraints, rewritten constraints, but also usual Prolog code)
- It is executed when a constraint that matches the **head** is posted in the constraint store.
- Example: the "less or equal" constraint:
transitivity @ leq(X,Y) , leq(Y,Z) ==> leq(X,Z).
Effect: a constraint is added



CHR – Constraint Handling Rules

Three types of rules:

- **Simpagation:** The simpagation rule removes the constraints in its head after the \ and then calls its body.

head1 \ head2 <=> body .

Head1 , head2 are lists of constraints, **body** is a Prolog goal that is executed (it will contains novel constraints, rewritten constraints, but also usual Prolog code). It's syntactic sugar

- It is executed when a constraint that matches the **head1 , head2** is posted in the constraint store.

- Example: the "less or equal" constraint:

idempotence @ leq(X,Y) \ leq(X,Y) <=> true .

Effect: a copy of the constraint is removed



CHR – Example

- The "less or equal" constraint

```
:– use_module(library(chr)).
```

```
% leq/2 is declared to be a constraint
```

```
:– chr_constraint leq/2.
```

```
% rules:
```

```
reflexivity @ leq(X,X) <=> true.
```

```
antisymmetry @ leq(X,Y), leq(Y,X) <=> X = Y.
```

```
idempotence @ leq(X,Y) \ leq(X,Y) <=> true.
```

```
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```



CHR – Does it matter to us?

- Prolog is based on backward reasoning
- However, there is an entire class of rule-based systems that behaves like ECA rules... i.e., forward...
- Forward reasoning behaves differently:
 - There is a store
 - Every time a new object is inserted in the store, one or more rules are triggered
- **CHR provides the base for forward reasoning in Prolog**





ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Rule-based systems and Forward Reasoning

Prof. Ing. Federico Chesani

DISI

Department of Informatics – Science and Engineering

Why rules?

- Rules are a very common way for eliciting knowledge, and how decisions should be in **given** and **known contexts**.
- Rules are easy to understand
- Rules are easy to be expressed and elicited
- Rules are very similar to a number of high-level cognitive tasks: humans adopt rules in a number of situations (default rules sometimes are applied at a unconsciousness level)
- Rules have a solid formal background in logic, and have been investigated since classical philosophy

- Backward Chaining (e.g., Prolog)
- Forward Chaining, such as production rules (e.g., RedHat Drools)



Rule-based frameworks and process automation

- All the 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.
- Microsoft: BizTalk Framework, and the Business Rules Engine
<https://docs.microsoft.com/en-us/biztalk/core/business-rules-engine>
- IBM: Operational Decision Manager
<https://www.ibm.com/automation/software/business-rules-management>
- Oracle: JBoss Enterprise Application Platform and Drools
<https://www.drools.org/>
- OpenRules
<http://openrules.com/>



The Decision Model and Notation: a curious case

- Few years ago, a novel research area emerged, namely the Business Process Management research field
- In few years it became a large community with a strong industry participation
- Initially, the focus was on how to describe business process...
- ... until the BPMN standard has been proposed by OMG, and widely accepted



The Decision Model and Notation: a curious case

- After the definition of the BPMN standard, in the BPM community they soon realized that decisions were a fundamental part of any business process
- How decision are taken in the business?
- Which criteria?
- How to describe these criteria?

The answer was again RULES!!! But in a "nicer" form...

- The DMN notation was proposed...
- ...a simplified version of forward rules!!!



How rules are made?

The simplest form of a rule is the logical implication:

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

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

Examples:

- If it rains, then the road will become wet and slippery
- If you don't study, you will fail the exam
- If don't pay the tax within the deadline, you will be fined for 15% of the due amount, plus 43.15€ for each day of delay in the payment



How to reason with rules?

Reasoning with rules has been systematized in classic greek philosophy (Aristotle, syllogism). The default reasoning mechanism is **Modus Ponens**:

Peirce Notation:

$$\frac{p_1, \dots, p_i \quad p_1, \dots, p_i \rightarrow q_1, \dots, q_j}{q_1, \dots, q_j}$$

- We know for sure that p_1, \dots, p_i are true
- We know for sure that the rule holds
- We derive the truthness of q_1, \dots, q_j
- NOTICE: this is not the only possible inference mechanism...



From static knowledge bases to dynamic ones: the "Production Rules" approach

- In previous examples we have seen backward chaining applied to a static knowledge base
- The same can be said for a number of 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 solution (naive): **when needed**, restart the reasoning from scratch...
 - Which cost?
 - When it is worthy to restart the reasoning?



From static knowledge bases to dynamic ones: the "Production Rules" approach

- In the production rules approach, **new facts can be dynamically added to the knowledge base**
 - Which difference with the lemma generator seen in prolog meta-interpreters?
- if new facts matches with the left hand side of any rule, the reasoning mechanism is **triggered**, until it reaches **quiescence** again...

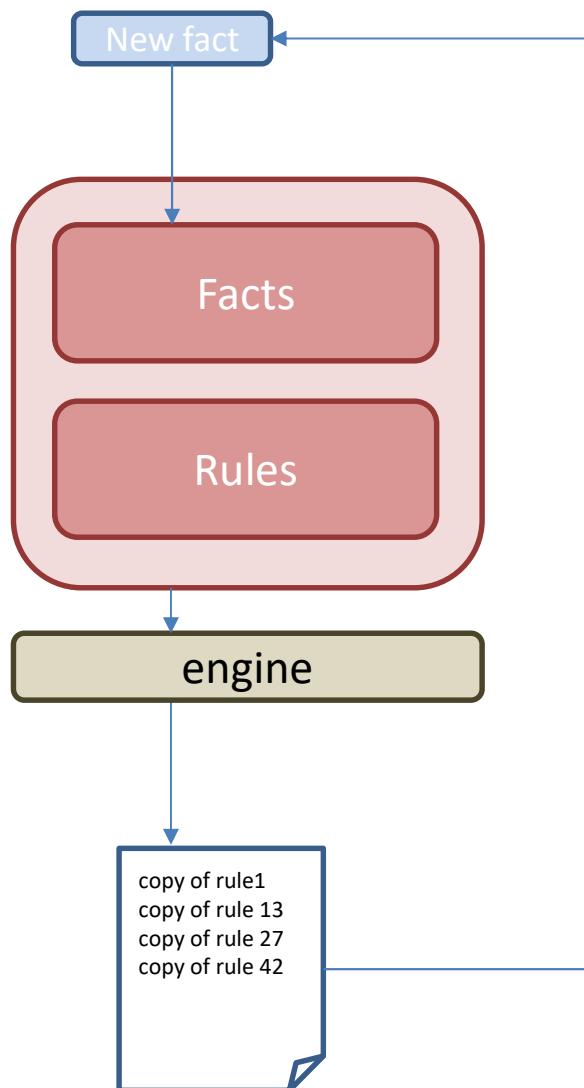


From static knowledge bases to dynamic ones: the "Production Rules" approach

- Production Rule systems: they allow to explicitly have **side effects** in the RHS
 - side effects on the external world
 - side effects on the knowledge base
- Which side effects on the working memory?
 - **Insertion** of facts
 - **Deletion** of facts
 - ... and consequently **retriggering** of rules
- we are leaving the "logical" setting, heading towards a more "procedural" framework
 - what if new facts are **inconsistent** with the existing ones?
 - what if new facts **trigger more rules**? which one to start first?
 - what if new facts trigger a rule that generates **side effects**?
 - what if a new fact trigger a rule, and that rule **deletes a fact** that was the antecedent of a rule already triggered?



Production Rules



Steps

When a new fact is **added** to the knowledge base, or **initially**:

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



Working memory

- Data structure at the base of Production Rules Systems.
- It contains:
 - the current set of facts (initially, it is just a copy of the KB)
 - the set of rules
 - the set of the (copies of) "partially matched" rules
- Performances of PRS might depend on the efficiency of the WM...



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The RETE Algorithm

- Charles Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence, 19, pp 17-37, 1982
- **Focuses on the step 1: "Match"**
- The match step consists on computing which are the rules whose LHS is "matched"
- LHS is usually expressed as a conjunction of patterns
- 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 patterns" vs "many objects" pattern match problem



The RETE Algorithm

Example

- Rule1: "when a Person p is added to the WM, and she is a student, send her a greeting message"
- Rule2: "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 the WM"
- The problem is how to determine efficiently which are the rules that should be triggered...
- ... in this example, the cost is linear with the number of rules
- ... iteration over the rules



The RETE Algorithm

Another example:

- Rule1: "when a Person p is added to the WM, and she is a student, and for every other Person stored in the WM, that are student themselves, send a message about the newcomer."
- Fact: "a Person whose name is Sara, and whose role is student, is added to the WM"
- The problem, again, is how to determine efficiently which are the rules that should be triggered...
- ... in this example, if we have to evaluate the LHS at the insertion (from scratch) the cost is linear with the dimension of the facts
- ... iteration over the facts



The RETE Algorithm – avoid iteration over facts

- RETE focuses on determining the so called "conflict set"...
- ... i.e., the set of all possible instantiations of production rules such that
 $\langle \text{Rule (RHS)}, \text{List of elements matched by its LHS} \rangle$

Idea: 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 RETE Algorithm – avoid iteration over the rules

Idea: "compile" a LHS into a network of nodes.

(At least) Two types 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, raise an alarm".



The RETE Algorithm – avoid iteration over the rules

Idea: "compile" a LHS into a network of nodes.

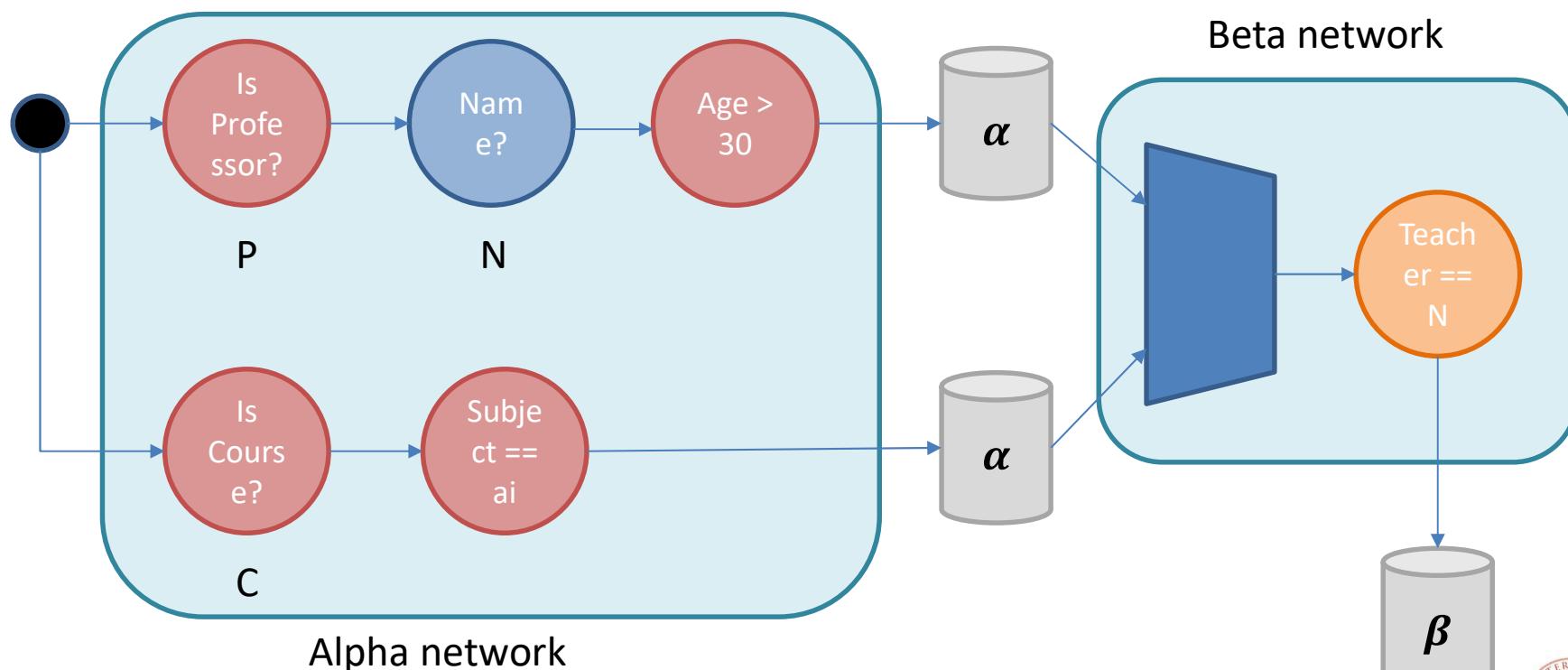
(At least) Two types of patterns:

1. Patterns testing intra-elements features
 2. Patterns testing inter-elements features
-
- Intra-elements patterns are compiled into alpha networks, and their outcomes are stored into alpha-memories
 - Inter-elements are compiled into beta-networks, and their outcomes are stored into beta-memories

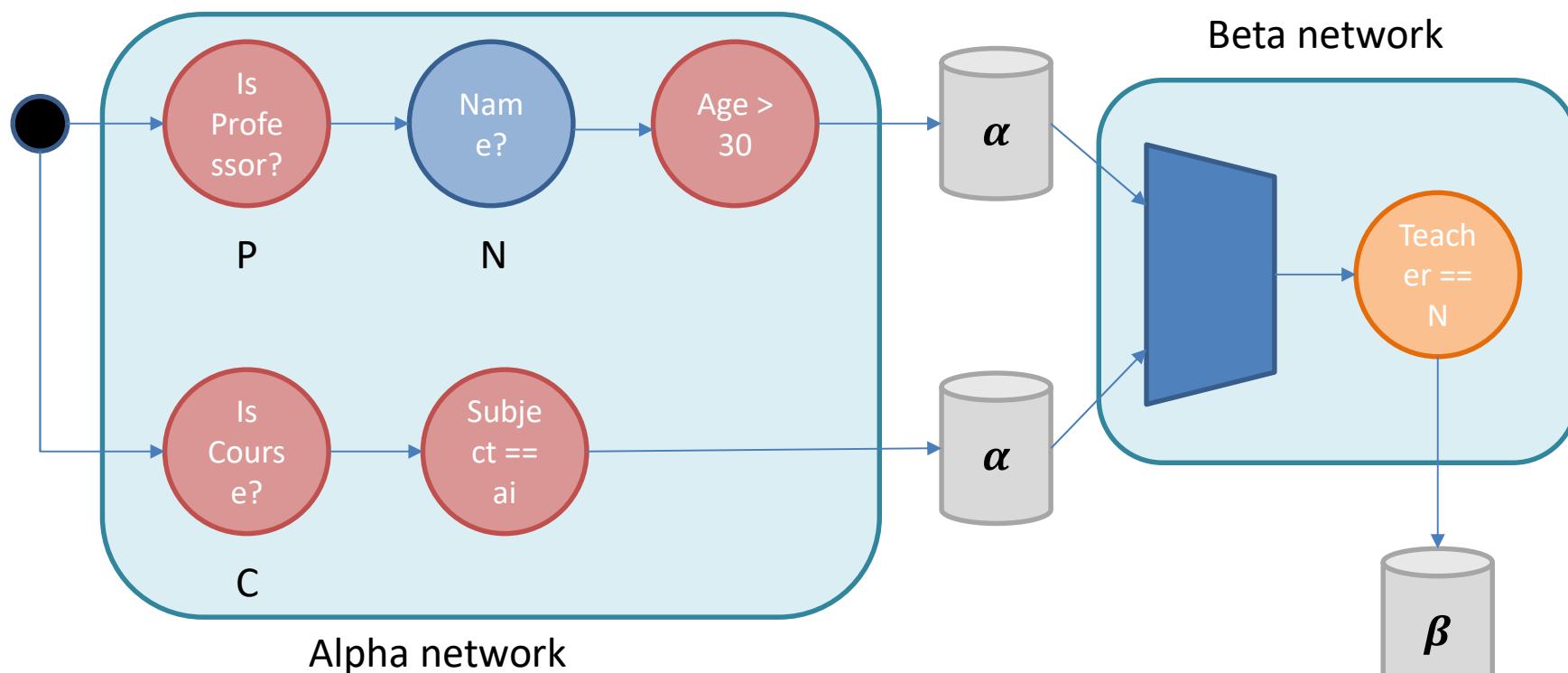


Rete algorithm – an example

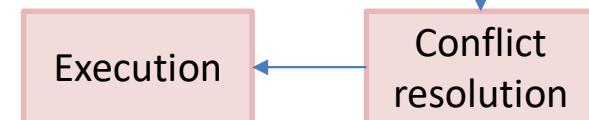
When a Professor P with name N and age>30, when a Course C with subject "ai" and teacher N THEN do something



Rete algorithm – an example



- In the end, the beta store contains the conflict set...
- ... i.e., the list of $\langle \text{Rule}, \text{LHS} \rangle$, with LHS completely matched, and ready to be executed.



... and the other steps?

- **Step2 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?
 - rule priority, aka salience
 - rule order
 - some temporal attribute (e.g., the time at which facts were inserted into the WM)
 - complexity of a rule
 - ...



... and the other steps?

- **Step3 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
- What about possible loops?



The DROOLS Framework



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The Drools Framework

- Supported by JBoss/RedHat
- It comes with a plethora of tools for supporting the definition the business logic. Using JBoss terminology, Drools is a "Business Logic Integration Platform".
- Drools Expert – the Rule Engine itself
- Drools Fusion – support for the Event Processing
- jBPM – workflow engine
- OptaPlanner – Automated planning



The Drools Framework

- From the Drools manual:
- ‘Drools was also born as a rules engine several years ago, but following the vision of becoming a single platform for behavioral modelling, it soon realized that it could only achieve this goal by crediting the same importance to the three complementary business modelling techniques:
 1. Business Rules Management
 2. Business Processes Management
 3. Complex Event Processing”



Drools Expert

Few characteristics:

- Forward reasoning approach
- **Based on the RETE algorithm, and current version on Phreak algorithm (an evolution of RETE and ReteOO)**
- Open Source
- Java-based
- The language is proprietary, but easy to extend it with custom operators, etc. (Java...)
- Provides an IDE (Eclipse), but also web interfaces
- Requires only a superficial comprehension of the Java language
- Open community of developers, very keen to provide quick answers on blogs and relay chats



The Drools 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



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.
 - describes a **fact** (that could appear in the WM)
 - describes also the **conditions** about the specific fact
 - Indeed, it is a sort of a filter for the objects/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



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Left Hand Side – LHS

Example: "When a Person is inserted in the system, send him a greeting email"

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

- The Person () pattern is a base one, and corresponds to the constraint:
- **x instanceof Person ?**

- Notice that Drools is Java-based, hence it inherits the instanceof definition from Java (e.g., no multiple inheritance).



Left Hand Side – LHS

Example: "When a Person is inserted in the system, send him a greeting email"

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

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;
2. At the start of the rule engine, if a fact Person() was already in the working memory (initial setup of the WM).



Left Hand Side – LHS

The basic pattern alone is poor (from the expressiveness viewpoint)...

... support to constraints on the object fields (**field constraints**):

- classical operators, extended to primitive types and String
 - ==, <, >=, ...
- connected through:
 - logical and: ' , ' or ' &&'
 - logical or: ' || '

Example: "When a Person of age, whose name is "federico", is inserted in the system, send him a greeting email"

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



Left Hand Side – LHS

Example: "When a Person of age, whose name is "federico", is inserted in the system, send him a greeting email"

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

Again, the meaning is inherited from Java...

```
x.getName().equals("Federico") && x.getAge() >= 18
```

What if name and age fields are not accessible through standard java bean notation?

COMPILE-TIME or RUN-TIME ERROR!



Left Hand Side – LHS

What if we need to keep in mind certain values/fields/objects?

Variables:

- any literal, starting with the ‘ \$’ character
- assigned through the ‘ : ’

Variables can be assigned to both objects/facts, as well as fields (constrained or not)

Example: "When a Person of age, whose name is “Federico”, is inserted in the system, send him a **personalized** greeting email"

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



Left Hand Side – LHS

What if we want to trigger rules when more facts appear at the same time? **Join**: allows to specify multiple patterns in the LHS

- all the pattern must be intended in logical AND: the rule trigger when all the pattern are satisfied
- all the possible combinations are generated by using all the objects that match with the premises

Constraints between different patterns can be imposed by using the variables, whose visibility scope is the whole rule

Example: "When a married couple is inserted in the system, do ..."

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



Left Hand Side – LHS

What if we want to trigger rules when more facts appear at the same time? **Join**: allows to specify multiple patterns in the LHS

- **all the possible combinations are generated** by using all the objects that match with the premises

Example: "When a married couple is inserted in the system, do ..."

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

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



Left Hand Side – LHS

What if we want to trigger rules when more facts appear at the same time? **Join**: allows to specify multiple patterns in the LHS

- Constraints between different patterns can be imposed by using the variables, whose visibility scope is the whole rule
- The keyword “this” can be used in field constraints to refer to the current object/fact

Example: "When a married couple is inserted in the system, do ..."

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



Left Hand Side – LHS

Quantifiers

- It is also possible to filter on the basis of three quantifiers:
- **exists P(...)** : there exists at least a fact $P(\dots)$ in the working memory
- **not P(...)** : the working memory does not contain any fact $P(\dots)$
 - when is such test performed?
- **forall P(...)** : trigger the rule if all the instance of $P(\dots)$ match



Left Hand Side – LHS

Quantifiers

It is also possible to filter on the basis of three quantifiers:

`exists P(...)` : there exists at least a fact `P(...)` in the working memory

Example: "Print the name of all those persons that have been fined at least once ..."

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.



Left Hand Side – LHS

Quantifiers

It is also possible to filter on the basis of three quantifiers:

- `not P(...)` : the working memory does not contain any fact `P(...)`

Example: "Print the name of all those persons that have never been fined"

rule "Never fined"

when

```
$p1 : Person( $n1 : name)  
not Fine( subject == $p1)
```

then

```
System.out.println($n1 + " was never fined");
```

end



Left Hand Side – LHS

Quantifiers

It is also possible to filter on the basis of three quantifiers:

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

Example: "Print the name of the persons that have been fined only for speed"

rule "Speed-only fined"

when

```
$p1 : Person( $n1 : name)  
      forall Fine( subject == $p1, reason="speed")
```

then

```
System.out.println($n1 + " was fined only for  
speed");
```

end

- The rule triggers zero or one time.



Left Hand Side – LHS

Facts... how are they made?

- LHS are defined through the use of patterns, that provide a way for expressing match constraints
- Facts are put into the WM during execution, or they are already there (initialization)

How facts are made?

- a) Java-based Beans, import clause at the beginning of a rule file
`import it.unibo.bbs.dss.Person;`
- b) Directly defined within the rule file, with explicit list of fields
(internally, mapped as Java Beans)

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



Right Hand Side – RHS

Consequences can be of two types:

a) "Logic" consequences: 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)

b) "Non-Logic" consequences:

- Any (external) side effect
- pieces of Java code that will be executed



Right Hand Side – RHS

Insert

- In the RHS it is possible to create new facts, and directly insert them into the WM.
- If the new fact matches with the LHS of any rule, then the rule will possibly trigger.

"When a registered person logins correctly, mark it as logged-in; when it logs out, check if it was logged in, and in positive case, say "goodbye""

```
declare Logged

    person : Person

end

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

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



Right Hand Side – RHS

Insert

In the RHS it is possible to create new facts, and directly insert them into the WM.

- If the new fact matches with the LHS of any rule, then the rule will possibly trigger.

"If you receive an email, immediately reply with "out-of-office" "

```
declare EMail

    sender : String
    dest : String

end

rule "automatic reply"
when
    $e : EMail ($s:sender, $d:dest)
then
    Email em = new Email();
    em.setSender($d);
    em.setDest($s);
    sendMail(em);
    insert(em);
end
```



Right Hand Side – RHS

InsertLogical

In the RHS it is possible to create new facts, and directly insert them into the WM.

- If the new fact matches with the LHS of any rule, then the rule will possibly trigger.
- the facts that were inserted are **automatically retracted** when the conditions in the rules that inserted the facts are no longer true.

TMS?



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Right Hand Side – RHS

Retract

When desired, it is possible to remove facts from the WM.

- Rules partially instantiated are discarded coherently
- Side effects are not retracted (it couldn't be possible)

"When a registered person logins correctly, mark it as logged-in; when it logs out, check if it was logged in, and in positive case, say "goodbye" "

```
declare Logged

    person : Person
end

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

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



Right Hand Side – RHS

Modify/Update

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

"When a person logins, update the date of the last login "

```
declare LastLogged
    person : Person
    lastLogin : Date
end

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

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



Right Hand Side – RHS

Insert, Modify/Update and loops

Insert and modify/update operations can be easily subject to 1-step loops.

To avoid loops, there is the no-loop keyword:

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

The intended meaning is that the rule shouldn't trigger, if the objects are modified by the rule itself...



Left Hand Side – LHS

Other features

It is also possible to filter on the basis of three quantifiers:

- `from Collection<P(...)>` : allows to evaluate LHS against a Collection of objects even if these objects are not in the WM
- `collect(P(...))` : constructs a Collection of objects stored in the WM
- `accumulate (P(...), aggregate_functions)` : allows to collect a set of instances of P(...), and to extract aggregated information

```
rule "Customers age"
when
    accumulate ( Person( $a : age),
                 $max : max( $a ),
                 $min : min ( $a ),
                 $avg : average ($a )
)
then
...
end
```



Conflict resolution ... salience

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

```
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
```



Conflict resolution ... agenda group

- Rules can have the property agenda-group: group of rules are selected to be executed from an external methods `setFocus(...);`

```
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
```



Conflict resolution ... activation group

Rules can have the property activation-group

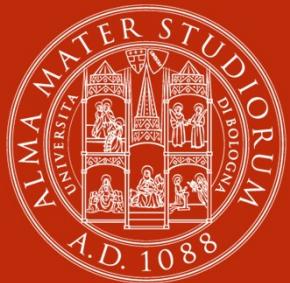
- Among all the activated rules of the same group...
- ... only one of them is executed...
- And the other are discarded.

```
declare Logged
    person : Person
end

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

rule "log in with premium"
activation-group "log"
when
    $p : Person ()
    $e : LogInEvent(person == $p)
    Logged ooo = new Logged();
    ooo.setPerson($p);
    insert(ooo);
    System.out.println("Congrats! . . .");
end
```





ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DMN – Decision Model and Notation

Federico Chesani

Department of
Computer Science and engineering
University of Bologna

Motivations

Decision making activities are usually addressed by two different viewpoints:

- workflow languages: specific tasks are devoted, decision points can be represented, no space for the logics, no semantics of the logics, some ambiguity resulting, automatization possible only through human intervention, error-prone process...
- Decision Logics tools: good to specify an (executable) decision criteria, scarce integration with process perspective, and few aspects still uncaptured...

However, taking a decision (usually) is a more complex a sophisticated process... that needs to be **shared** among managers, customers, developers, properly **documented**, and possibly **executed** without ambiguities or further human interventions.

Which types of Decisions?

Roughly speaking, three main types of decisions (from a management perspective):

1. **Strategic Decisions:** few times, huge impact, many high managers involved, many documents and business studies, perspectives, business analysis, consultants, etc.
2. **Tactical Decisions:** focused on management and control, medium impact, based more on technical studies, impact the internal organizational structure in the production depts.
3. **Operational Decisions:** day-to-day, repeatable schemas, impacting the single business cases, time-pressure to act quickly, left to operations managers.

How to model a decision

As a first, simplified instance, a decision can be modeled as an iterative process:

1. Identify the Decision
2. Describe the Decision
3. Specify Decision Requirements
4. Decompose and Refine (go back to step 1.)

1. Identifying the Decision

Decisions are part of the *requirement gathering* process. They come up as the consequence of:

- business processes analysis
- use cases
- user detailed requirement
- ... in any step of the process elicitation

Questions you should always ask yourself:

- What's the issue we should decide on?
- Do we all agree? Is there a consensus?
Is there one decision to be taken, or more than one?
Might be not a decision at all (straightforward)?
- Who is usually involved in this decision?
Ask her/him
- Who can provide us information about this decision?
Discuss with her/him and gather all the information you can

2. Describe the Decision

Decisions are a first entity class of DMN, they are the core of the whole standard

- Which is the **question**?
Is the question simple enough?
Can be understood by everyone involved in the business process elicitation?
- Which are the **allowed answers**?
Are they mutually exclusive or not?
Is some answer equivalent/replaceable by another one?
- In which **context** the **decision process** is valid?
which business context?
which organizational context?
which application context?

3. Specify Decision Requirements

Which are the **information** that are used to make the decision?

- Which **input data**?

- Example:

Decision: is the user allowed to use Whatsapp?

(Within Europe) People under-16 is not allowed to use Whatsapp.
Which is the input data?

Input data might depend on the single process case, but also by the current state of the system

Hence, input data might be resolved dynamically, at decision time.

- Which **knowledge** is involved in the decision?

Any internal/external policy?

Which regulations are involved?

Are there best practices currently in use?

Is human expertise involved in?

Do we need to consider any business analytics?

The Decision Model and Notation

- OMG Standard
- version 1.0, September 2015
- version 1.1, mainly bug fixes, June 2016
- <https://www.omg.org/spec/DMN>

A number of companies contribute it. Officially acknowledged:

- Decision Management Solutions
- Escape Velocity
- FICO
- International Business Machines
- Oracle
- KU Leuven
- Knowledge Partners International
- Model Systems
- TIBCO

The Decision Model and Notation

GOAL (from the normative document):

- (primary): to provide a common notation that is **readily understandable by all business users...**
 - from the business analysts...
 - ...to the technical developers responsible for automating ...
 - ... and finally, to the business people who will manage and monitor those decisions.
- DMN creates a standardized bridge for the **gap between the business decision design and decision implementation.**
- DMN notation is designed to be useable alongside the standard **BPMN** business process notation.

DMN – in practice...

DMN highlights:

- a graphical language for business decision modeling
- a standard notation for decision tables
- an expression language "Friendly Enough Expression language – FEEL"
and the "Simplified FEEL – S-FEEL"
- a metamodel and an interchange format
- three conformance levels for tools

Three main tasks:

- Modeling Human decision-making processes
 - Modeling requirements for automated decision-making
 - Implementing automated decision-making
- and any combination of the above...

Conformance Levels

Three Conformance Levels have been defined for tools supporting DMN:

1. Conformance Level 1: requires support for

- Decision Requirements Diagrams
- Decision Logic Diagrams
- Decision Tables

In practice...
...documentation!

2. Conformance Level 2: ask for support to CL1, plus

- S-FEEL
- decision logic modelled in S-FEEL, and capability of executing it

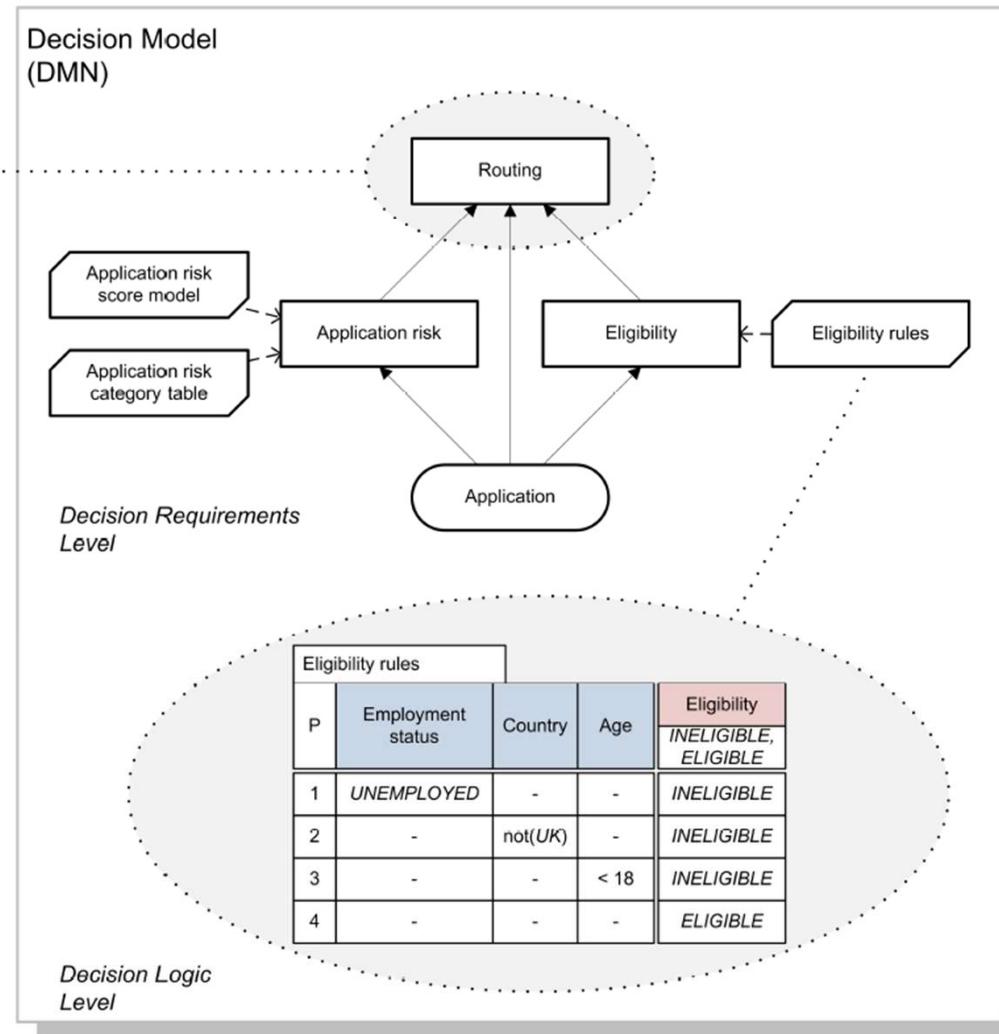
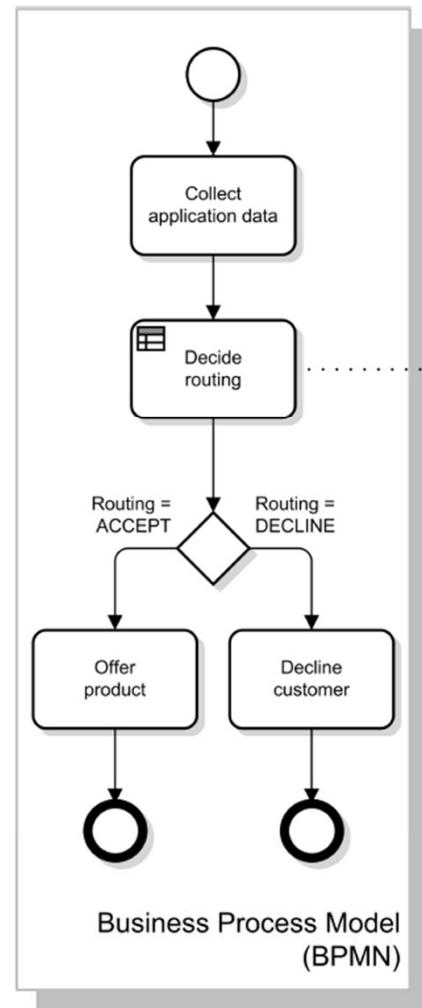
3. Conformance Level 3: ask for CL2, plus

- FEEL
- additional modelling elements

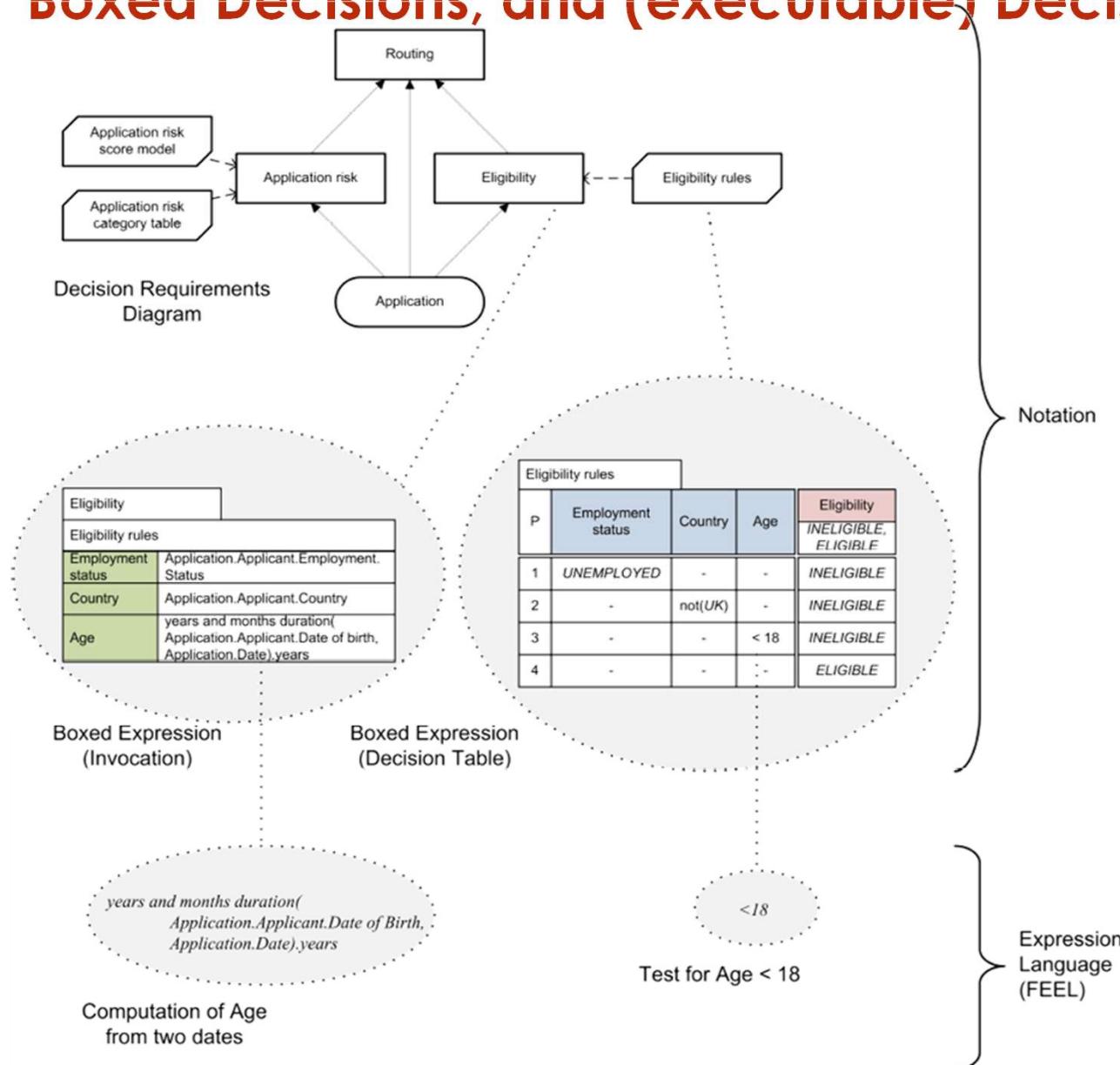
DMN and BPMN – Full Integration...

- Business process models will define tasks within business processes where decision-making is required to occur.
- **Decision Requirements Diagrams** will define the **decisions** to be made in those tasks, their **interrelationships**, and their **requirements** for decision logic.
- **Decision logic** will define the required decisions in sufficient detail to allow validation and/or automation

DMN and BPMN – Full Integration...



DMN and BPMN – Decision Requirement Diagrams, Boxed Decisions, and (executable) Decision Logic



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

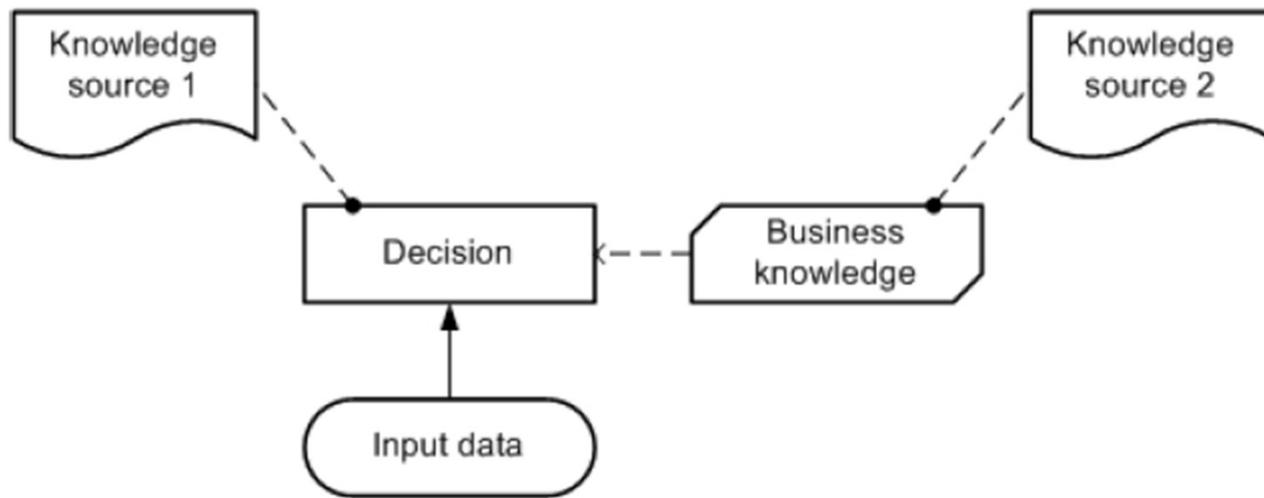
DMN – Decisions

From the OMG definition:

"A **decision** is the act of determining an **output** value (the chosen option), from a number of **input** values, using logic defining how the output is determined from the inputs."

This **decision logic** may include one or more **business knowledge models** which encapsulate business know-how in the form of business rules, analytic models, or other formalisms."

DMN – Business Knowledge and Knowledge sources



- **Business knowledge** captures the essence/algorithm of the decision. The decision logic might be defined in the decision element itself, or might be defined within one or more business knowledge elements.
- **Knowledge sources** capture the knowledge that is related to the source, such as business policies, regulations, laws, but also domain experts

DMN – DRGs and DRDs, and main elements

- Decision Requirements Graph (DRG): the domain of one (or more) decisions.
- Decision Requirement Diagram (DRD): diagrams depicting the decision requirements

Main elements:

- Decision
- Business Knowledge Model
- Input Data
- Knowledge Sources

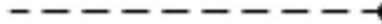
DMN – Main elements

Component	Description	Notation
Elements	Decision	A decision denotes the act of determining an output from a number of inputs, using decision logic which may reference one or more business knowledge models.
	Business Knowledge Model	A business knowledge model denotes a function encapsulating business knowledge, e.g., as business rules, a decision table, or an analytic model.
	Input Data	An input data element denotes information used as an input by one or more decisions. When enclosed within a knowledge model, it denotes the parameters to the knowledge model.
	Knowledge Source	A knowledge source denotes an authority for a business knowledge model or decision.



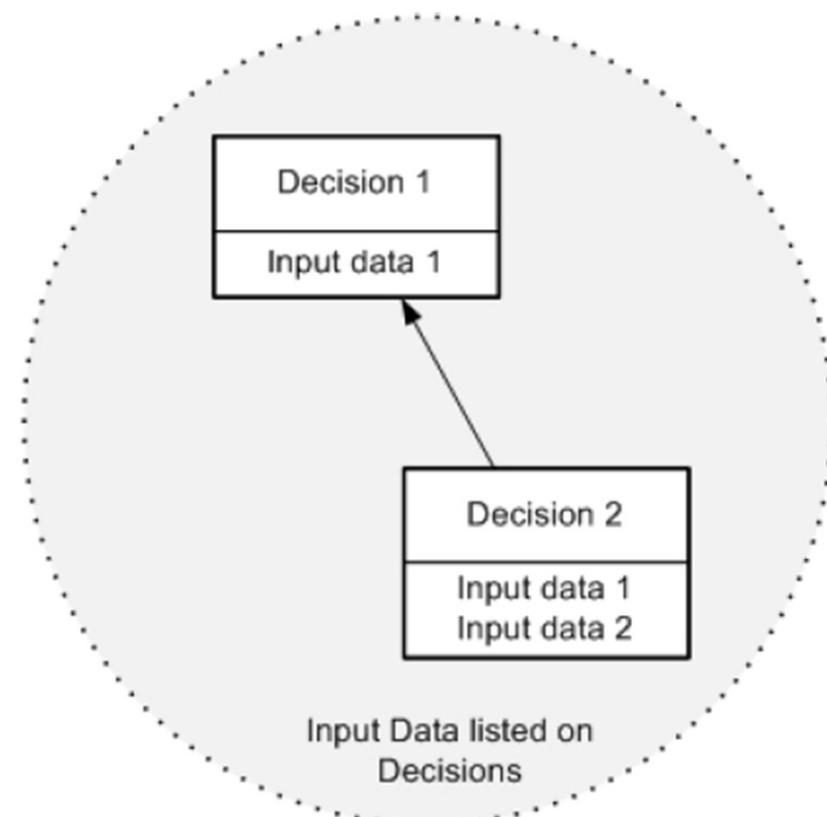
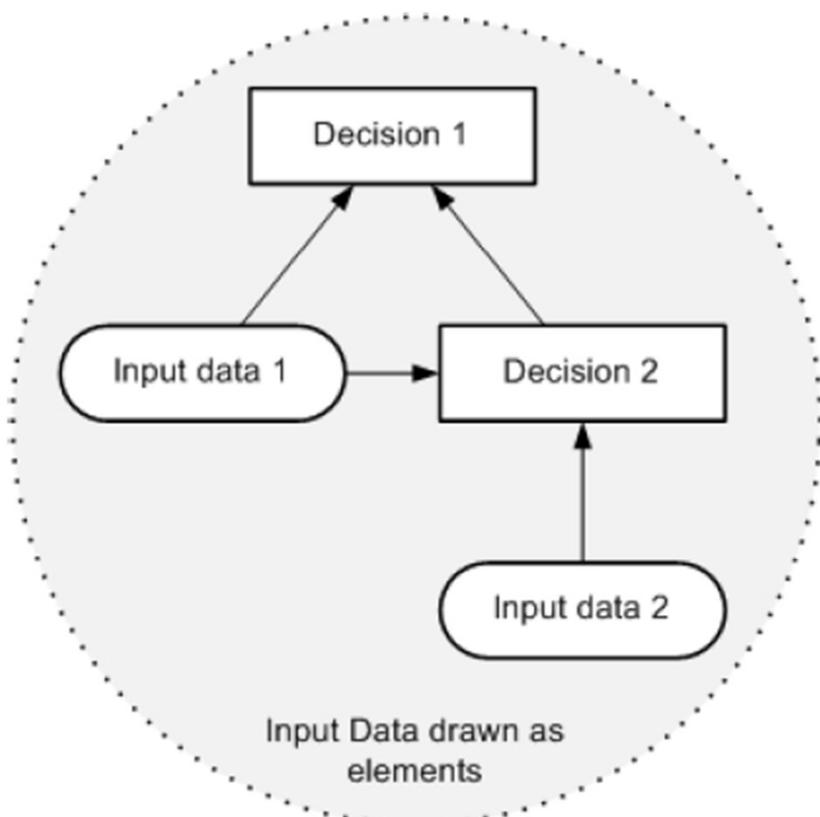
DMN – Requirements

- **Information Requirement:** input data, or a decision output used as input data
- **Knowledge Requirement:** a business knowledge model, needed to apply the decision logic
- Authority Requirement: requirement of some other knowledge(papers, experts, but also other DRG/DRD...)

Requirements	Information Requirement	An information requirement denotes input data or a decision output being used as one of the inputs of a decision.	
	Knowledge Requirement	A knowledge requirement denotes the invocation of a business knowledge model.	
	Authority Requirement	An authority requirement denotes the dependence of a DRD element on another DRD element that acts as a source of guidance or knowledge.	

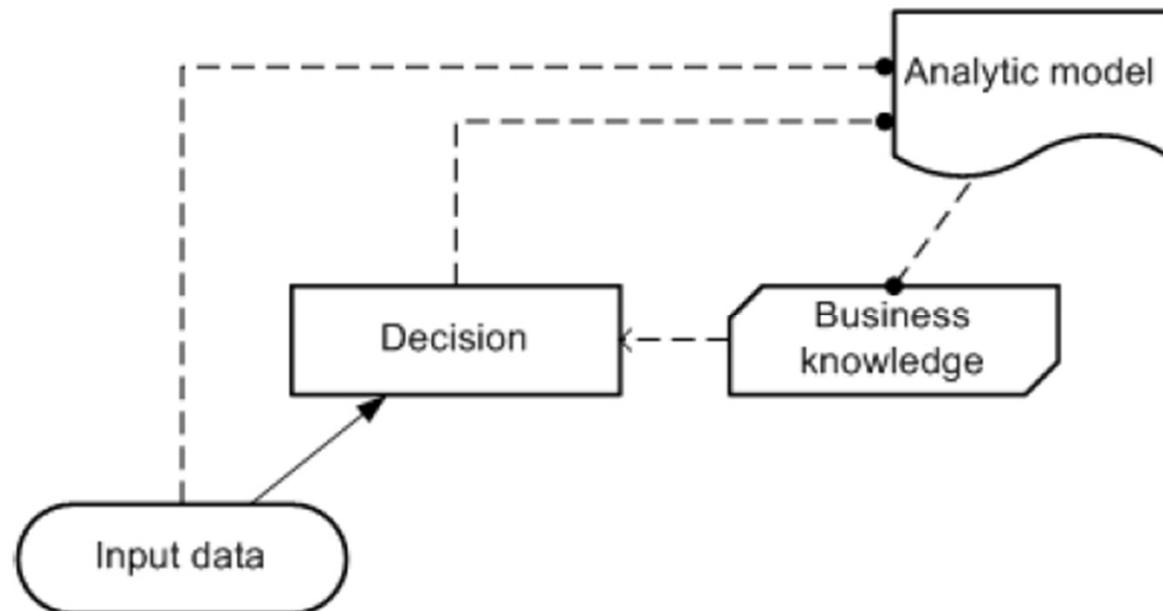


DMN – Input data – two variations



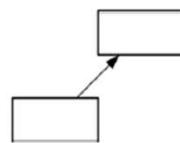
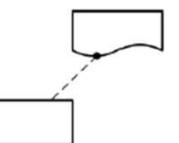
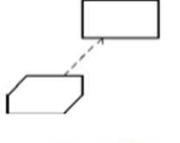
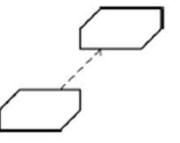
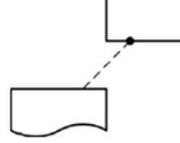
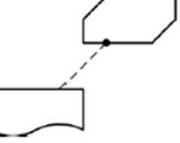
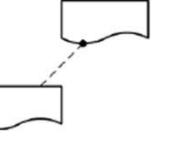
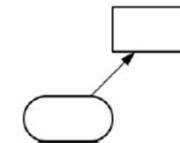
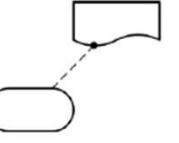
DMN – Requirements

- An example of predictive analytics:



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DMN – Connection rules – summing up...

		To			
		Decision	Business Knowledge Model	Knowledge Source	Input Data
	Decision	 Information Requirement	not allowed	 Authority Requirement	not allowed
	Business Knowledge Model	 Knowledge Requirement	 Knowledge Requirement	not allowed	not allowed
	Knowledge Source	 Authority Requirement	 Authority Requirement	 Authority Requirement	not allowed
	Input Data	 Information Requirement	not allowed	 Authority Requirement	not allowed



DMN – Decision Logic Level

- Decision Requirement Diagrams provide an overview of the decision process...
- ... but they do not tell anything on how to take the decision...

The decision criteria are specified at the Decision Logic Level. Three ways:

- Decision Tables
- S-FEEL expressions
- FEEL expressions
- ... but also (mind! at Level 1!!!) any other criteria comprising formal languages as well as natural language (mind the ambiguities!)

DMN – Decision Tables

From the DMN reference:

"A decision table is a tabular representation of a set of related input and output expressions, organized into rules indicating which output entry applies to a specific set of input entries."

Two basic requirements:

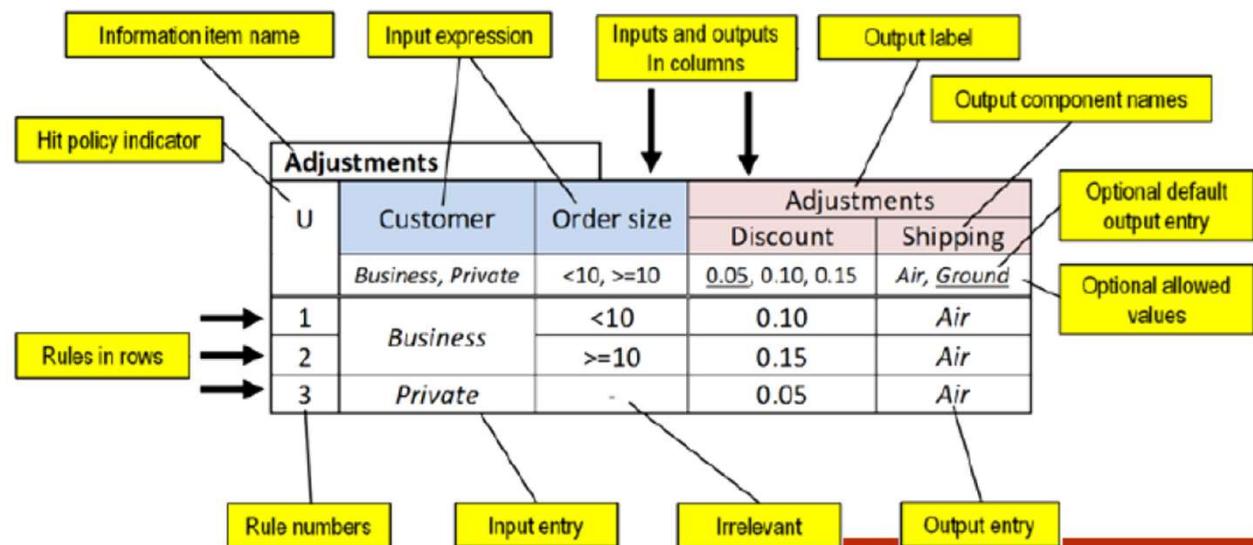
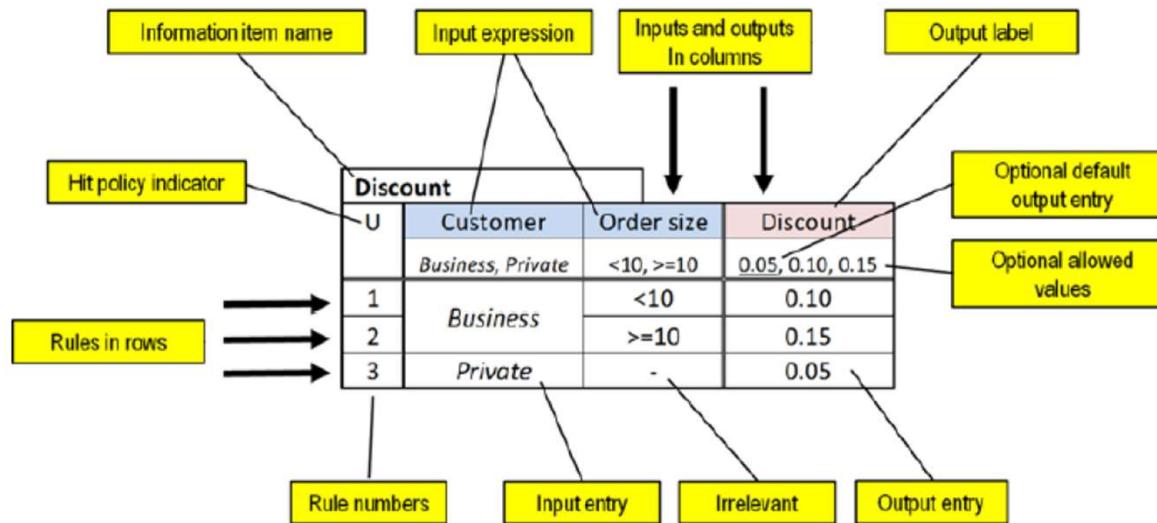
- The decision table contains all (and only) the inputs required to determine the output.
- Moreover, a complete table contains all possible combinations of input values (all the rules).

DMN – Decision Tables

Elements:

- **Information Item Name:** usually, the name of the decision, or of the Business Knowledge Model
- **Output Label:** simple label describing the meaning of the output. Mind: when referring to the output, the information item name is used for indicating both the output, as well as the decision table itself.
- **Set of Inputs (zero or more):** each input is made of an input expression and a number of input entries
- **Set of outputs** (one or more): one output implies no labels, more outputs require distinguishing labels
- **List of rules**

DMN – Decision Tables – rules as rows



DMN – Decision Tables – Interpretation of a rule

input expression 1	input expression 2	Output label
input entry a	input entry b	output entry c

The rule (a row, in this case) must be read as:

If the value of input expression 1 **satisfies** input entry a
and the value of input expression 2 **satisfies** input entry b
then the rule **matches** and the result of the decision table is output c.

An input expression value **satisfies** an input entry if:

1. the value is equal to the input entry, or
2. the value belongs to the list of values indicated by the input entry (e.g., a list or a range), or
3. If the input entry is '-' (meaning *irrelevant*), every value of the input expression satisfies the input entry and that particular input is irrelevant in the specified rule.

A rule **matches** if the value of every input expression satisfies the corresponding input entry. If there are no input entries, any rule matches.

DMN – Decision Tables – Rules overlapping

- If two input entries of the same input expression share no values, the entries (cells) are called *disjoint*.
- If there is an intersection, the entries are called *overlapping* (or even equal). ‘Irrelevant’ (‘-’) overlaps with any input entry of the input expression.
- Two rules are overlapping if *all corresponding input entries are overlapping*. A specific configuration of input data may then match the two rules.
- Two rules are *disjoint* (non-overlapping) if at least one pair of corresponding input expressions is disjoint. No specific configuration of input data will match the two rules.

What if two rules overlap? We need a **hit policy...**

DMN – Decision Tables – Rules and input values

Two properties, in input values, are required as good modeling practice:

- Input values should be exclusive. I.e., input values are disjoint.
- Input values should be complete. I.e., all relevant input values from the domain are present.

Moreover, "evaluation of the input expressions in a decision table does not produce side-effects that influence the evaluation of other input expressions.

This means that evaluating an expression or executing a rule should not change the evaluation of other expressions or rules of the same table."

→ Input evaluation is idempotent, and does not change other inputs!!! No Side-effects!!!

DMN – Rules overlapping and Hit Policies

A hit policy specify how to resolve overlapping rules, i.e. which rule is chosen in case there are overlaps. Several policies are allowed. First let us focus on **single hit tables**:

1. **Unique**: no overlap is possible and all rules are disjoint. This is the default.
2. **Any**: there may be overlap, but all of the matching rules show equal output entries for each output, so any match can be used.
If the output entries are non-equal, the hit policy is incorrect and the result is undefined.
3. **Priority**: multiple rules can match, with different output entries. This policy returns the matching rule with the highest output priority. Output priorities are specified in the ordered list of output values, in decreasing order of priority. Note that priorities are independent from rule sequence.
4. **First**: multiple (overlapping) rules can match, with different output entries.
The first hit by rule order is returned (and evaluation can halt).
Not a good practice: the meaning depends on the order of the rules.

The policy must always be written in the decision table

DMN – Decision Tables – Examples

U	Applicant Age	Medical History	Applicant Risk Rating
1	> 60	<i>good</i>	<i>Medium</i>
2		<i>bad</i>	<i>High</i>
3	[25..60]	-	<i>Medium</i>
4	< 25	<i>good</i>	<i>Low</i>
5		<i>bad</i>	<i>Medium</i>

Person Loan Compliance				
A	Persons Credit Rating from Bureau	Person Credit Card Balance	Person Education Loan Balance	Person Loan Compliance
1	A	< 10000	< 50000	<i>Compliant</i>
2	Not(A)	-	-	<i>Not Compliant</i>
3	-	>= 10000	-	<i>Not Compliant</i>
4	-	-	>= 50000	<i>Not Compliant</i>



DMN – FEEL and S-FEEL

Decision tables are indeed a powerful method for representing decision criteria...

... but are scarcely interpretable by a machine.

As a consequence, Decision Tables are good for human-sharing, not so good for automatizing processes.

Solution:

- Definition of the Friendly Enough Expression Language.
Aimed to be usable by any user "capable of using Excel macros"
- Definition of the Simplified FEEL (S-FEEL): a simpler subset of FEEL.

Note: previous examples where already adopting the FEEL syntax...

DMN – FEEL and S-FEEL

Features:

- Side-effect free
- Simple data model with numbers, dates, strings, lists, and contexts
- Simple syntax designed for a wide audience
- Three-valued logic (**true**, **false**, **null**) based on SQL and PMML

Datatypes:

- number
- string
- Boolean
- days and time duration
- years and months duration
- time
- date and time

DMN – FEEL and S-FEEL

Classic operators (e.g., arithmetic) are defined for all the types: e.g., sum is defined for time durations as the sum of the intervals.

There is also the possibility of defining ranges:

Grammar Rule	FEEL Syntax	Equivalent FEEL Syntax	applicability
51.b	e_1 between e_2 and e_3	$e_1 \geq e_2$ and $e_1 \leq e_3$	
51.c	e_1 in $[e_2, e_3, \dots]$	$e_1 = e_2$ or $e_1 = e_3$ or ...	e_2 and e_3 are endpoints
51.c	e_1 in $[e_2, e_3, \dots]$	e_1 in e_2 or e_1 in e_3 or ...	e_2 and e_3 are ranges
51.c	e_1 in $\leq e_2$	$e_1 \leq e_2$	
51.c	e_1 in $\leq e_2$	$e_1 < e_2$	
51.c	e_1 in $\geq e_2$	$e_1 \geq e_2$	
51.c	e_1 in $\leq e_2$	$e_1 < e_2$	
51.c	e_1 in $(e_2..e_3)$	$e_1 > e_2$ and $e_1 \leq e_3$	
51.c	e_1 in $(e_2..e_3]$	$e_1 > e_2$ and $e_1 \leq e_3$	
51.c	e_1 in $[e_2..e_3)$	$e_1 \geq e_2$ and $e_1 < e_3$	
51.c	e_1 in $[e_2..e_3]$	$e_1 \geq e_2$ and $e_1 \leq e_3$	



DMN – FEEL and S-FEEL

Example of range comparison:

FEEL Expression	Value
5 in (<u><=5</u>)	true
5 in (<u>5..10]</u>)	false
5 in (<u>[5..10]</u>)	true
5 in (<u>4, 5, 6</u>)	true
5 in (<u><5, >5</u>)	false
2012-12-31 in (<u>2012-12-25..2013-02-14</u>)	true



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Federico Chesani

DISI – Department of Computer Science and Engineering
University of Bologna

federico.chesani@unibo.it

www.unibo.it



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Complex Event Processing and Event Calculus

Federico Chesani

Department of
Computer Science and Engineering
University of Bologna

The Internet of Thing paradigm, Big Data, and other...

We are assisting to an exponential increase of:

- “simple” devices that have sensors (e.g., traffic lights presence sensors)
- “simple” devices that have a net connection, and whose data can be accessed continuously at run-time (e.g., environmental and weather stations are queried through the net at runtime)
- “complex devices” that continuously provides a flow of information (security cams, webcams, etc.)

All these devices are producing a huge quantity of data, thus giving us the “Big Data” challenge.

The Internet of Thing paradigm, Big Data, and other...

The Big Data 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 users privacy and rights about this huge amount of information

Applications can be the most various and disparate... my personal feeling is that

- marketing needs are driving and pushing the research, in this moment...
- ... strictly followed by business ICT frameworks (big ICT players), for supporting industrial applications (e.g., industry 4.0)

The Complex Event Processing paradigm...

Complex Event Processing (CEP) is a paradigm (older than IoT and Big Data), for dealing with such huge amount of information

- 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** (philosophical/ontological issue, but with a number of practical consequences)

It is a very natural, human-like way of describing the information collected by sensors...

- Underneath, it is adopting the human reasoning way of abstracting from correlation to causality, with the flow of the time

The Complex Event Processing paradigm...

Complex Event Processing (CEP) is a paradigm (older than IoT and Big Data), for dealing with such huge amount of information

Problem: events alone usually do not provide enough information

Example: A cross road has been added a (buried in the ground) sensor that is able to detect if a car is passing over the sensor itself.

The outcome is something like:

1528038740909 something detected

1528038779384 something detected

1528038790093 something detected

1528038805045 something detected

...

So, what?

The Complex Event Processing paradigm...

Problem: events alone usually do not provide enough information

Example: A cross road has been added a (buried in the ground) sensor that is able to detect if a car is passing over the sensor itself.

The outcome is something like:

1528038740909 something detected

1528038779384 something detected

1528038790093 something detected

1528038805045 something detected

...

In the cross road, we are interested to understand if the road is jammed or traffic or not...

... we would need to extract from the single (simple) events such information...

... for example, by simply **averaging** the number of events within a **sliding temporal window**...

The Complex Event Processing paradigm...

Problem: Maritime Surveillance (from works of Artikis and colleagues)

For each ship in the world, events are collected representing:

- id of the vessel
- GPS location
- time of the detection

Security questions:

- Fast approaching: a vessel is moving at high speed towards other vessels
- Suspicious delay: a vessel fails to report its position, and the estimated speed during the information gap is estimated as low
- Possible rendez-vous: two vessels are suspiciously delayed, at the same time, in the same location

The Complex Event Processing paradigm...

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

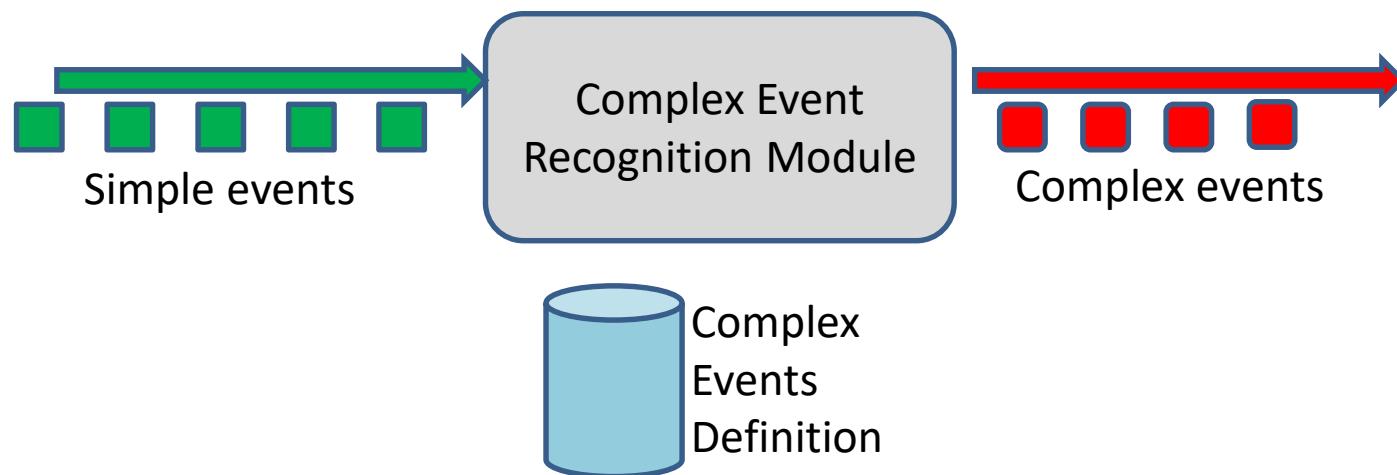
- **Simple events:** events detected by the system underneath, their information payload alone does not provide any immediate answer to our questions
- **Complex events:** events generated by the system itself (they do not necessarily map on real events), that provide higher informative payload, and that can partially/totally answer our application questions

Simple and complex do not refer to the information payload, but rather to their ability to provide answers to the application questions.

The Complex Event Processing paradigm...

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

The term CEP capture the whole framework needed to recognize events. Usually, authors refer also to **Complex Event Recognition Languages**.



The Complex Event Processing paradigm...

According to Artikis et al., CEP/CER can be characterized by:

INPUT:

- Instantaneous events.
- Durative events.
- Context information.

OUTPUT: durative events

- The interval may be open.
- Relational & non-relational events.
- Limits or No limits on the temporal distance between the events comprising the composite activity (support to sliding windows of possibly infinite length).
- Sequence constraints.
- Concurrency constraints.
- Spatial reasoning.
- Event hierarchies.
- Different sensors involved (different meaning) (Sensor Fusion)

The Complex Event Processing paradigm...

A number of different approaches have been proposed for applying CEP.

A classification, from the viewpoint of the adopted framework:

- Automata-based (e.g., Regular Expression)
- Tree-based
- Logic-based

In the following we will focus on Logic-based approaches, and in particular, on the Event Calculus logical framework.

CEP and Drools

Drools provide a support for doing CEP: **Drools Fusion**.

- Events are particular facts
- Events have a timestamp (but support for interval-time events as well)
- Support to Allen algebra for querying instantaneous events with respect to time interval
- On the base of the defined rules, it automatically decides the discard of events:
 - It avoids working memory cluttering and all related problems
 - 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
 - Sensor Fusion is not supported as operators, but can be achieved through the writing of chained rules and precise domain semantics
 - It supports the negation operator over time windows
 - It support fixed as well as sliding windows

CEP and Drools

Few basic assumptions (from the Drools manual):

- Usually required to process **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 state change.
- Usually the rules and queries on events must run in **reactive** modes, i.e., 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.

CEP and Drools – negation: different behavior

Drools Fusion adopts a view of the flow of events based on Stream: events come in with a timestamp...

Example:

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

When the not is evaluated? Different behavior between Cloud mode and Stream mode...

CEP and Drools – negation: different behavior

What about **not observing events**?

Example:

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

When the not is evaluated? Different behavior between Cloud mode and Stream mode...

Note: reasoning over temporal intervals requires the notion of a **clock**...

CEP and Drools – Sliding Windows

What about focusing on the events that happened only the last 24 hours?
How to focus on the last 100 events only?

“**Sliding Windows** are a way to scope the events of interest by defining a window that is constantly moving.”

Two types of sliding windows are supported:

- a) time-based windows
- b) length-based windows

CEP and Drools – Time-based sliding windows

“**Sliding Windows** are a way to scope the events of interest by defining a window that is constantly moving.”

Two types of sliding windows are supported:

a) **time-based windows**

b) 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:time(10m),
        average( $temp )
    )
then
    // sound the alarm
end
```

CEP and Drools – Sliding Windows

“**Sliding Windows** are a way to scope the events of interest by defining a window that is constantly moving.”

Two types of sliding windows are supported:

- a) time-based windows
- b) 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
```

CEP and Drools – Events Expiration

Due to the huge amount of events that should be treated, events will be discarded from the working memory ASAP...

Two ways for determining the expiration:

Explicit expiration:

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

Implicit expiration:

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

CEP and Drools – Temporal Reasoning

A number of Allen temporal operators are supported:

After and Before

```
$eventA : EventA( this after[ 3m30s, 4m ] $eventB )
```

meaning:

```
3m30s <= $eventA.startTimestamp - $eventB.endTimeStamp <= 4m
```

```
$eventA : EventA( this before[ 3m30s, 4m ] $eventB )
```

meaning:

```
3m30s <= $eventB.startTimestamp - $eventA.endTimeStamp <= 4m
```

CEP and Drools – Temporal Reasoning

A number of Allen temporal operators are supported:

Coincides:

```
$eventA : EventA( this coincides $eventB )
```

with threshold:

```
$eventA : EventA( this coincides[15s, 10s] $eventB )
```

During:

```
$eventA : EventA( this during $eventB )
```

meaning:

```
$eventB.startTimestamp < $eventA.startTimestamp <=  
$eventA.endTimestamp < $eventB.endTimestamp
```

CEP and Drools – Temporal Reasoning

A number of Allen temporal operators are supported:

- **finishes**
- **finishedby**
- **includes**
- **meets**
- **metby**
- **overlaps**
- **overlappedby**
- **starts**
- **startedby**

CEP and Drools – Allen operators brief recall

X before Y



Y after X

X meets Y



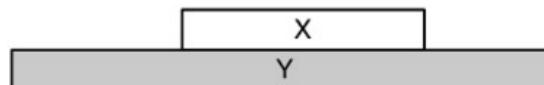
Y met-by X

X overlaps Y



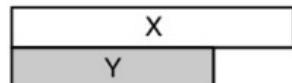
Y overlapped-by X

X during Y



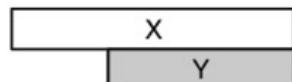
Y contains X

X starts Y



Y started-by X

X finishes Y



Y finished-by X

X equals Y



Taken from: Colonius, Immo. (2015). Qualitative Process Analysis : Theoretical Requirements and Practical Implementation in Naval Domain.



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CEP and Drools – Reasoning about the state of the system

CEP is a very nice way for reasoning about events... but...
...more complex systems require to reason about the state of a system.

How to, in Drools?

Let us use facts (and not events), to represent the state of the system

- Fast and easy solution for simple state representation
- Ok if only a type of event affects a state property

Adopt the **Event Calculus Framework** (Kowalski and Sergot, 1986)

- More complex solution, but cleaner
- Allows to link multiple different events to the same state property
- State property changes can depend also from other state
- Allows to reason on meta-event of state property change (clip and de-clip meta-events)

Event Calculus

- Proposed by Marek Sergot and Robert Kowalski, 1986
- Based on *points of time*
- Reifies both fluents and events

Fluents are properties whose truthness value changes over time

An ontology and two distinct set of axioms:

1. Event calculus "ontology" (fixed)
2. Domain-independent axioms (fixed)
3. Domain-dependent axioms (application dependent)

Event Calculus – EC Ontology

- **HoldsAt(F, T)**: The fluent F holds at time T
- **Happens(E, T)**: event E (i.e., the fact that an action has been executed) happened at time T
-
- **Initiates(E, F, T)**: 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(T₁, F, T)**: Fluent F has been made false between T₁ and T (used in domain-independent axioms)
- **Initially(F)** : fluent F holds at time 0

Event Calculus – Domain-independent Axioms

- $\text{HoldsAt}(F, T) \Leftarrow \text{Happens}(E, T_1) \wedge \text{Initiates}(E, F, T_1) \wedge (T_1 < T) \wedge \neg \text{Clipped}(T_1, F, T)$
- $\text{HoldsAt}(F, T) \Leftarrow \text{Initially}(F) \wedge \neg \text{Clipped}(0, F, T)$
- $\text{Clipped}(T_1, F, T_2) \Leftarrow \text{Happens}(E, T) \wedge (T_1 < T < T_2) \wedge \text{Terminates}(E, F, T)$

Event Calculus – Domain-dependent Axioms

A collection of axioms of type Initiates(...)/Terminates(...), and Initially(...)

EXAMPLE:

- 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).

Event Calculus – Events...

Given a set of events:

- Happens(push_button, 3)
- Happens(push_button, 5)
- Happens(push_button, 6)
- Happens(push_button, 8)
- Happens(push_button, 9)

Is the light on?

Event Calculus allows to answer the queries about HoldsAt predicates very easily

Event Calculus – few things...

- Easily implemented in Prolog...
- ...but (roughly) not safe if fluents/events contain variables, due to the negation in front of the clipping test, that is implemented in Prolog through NAF

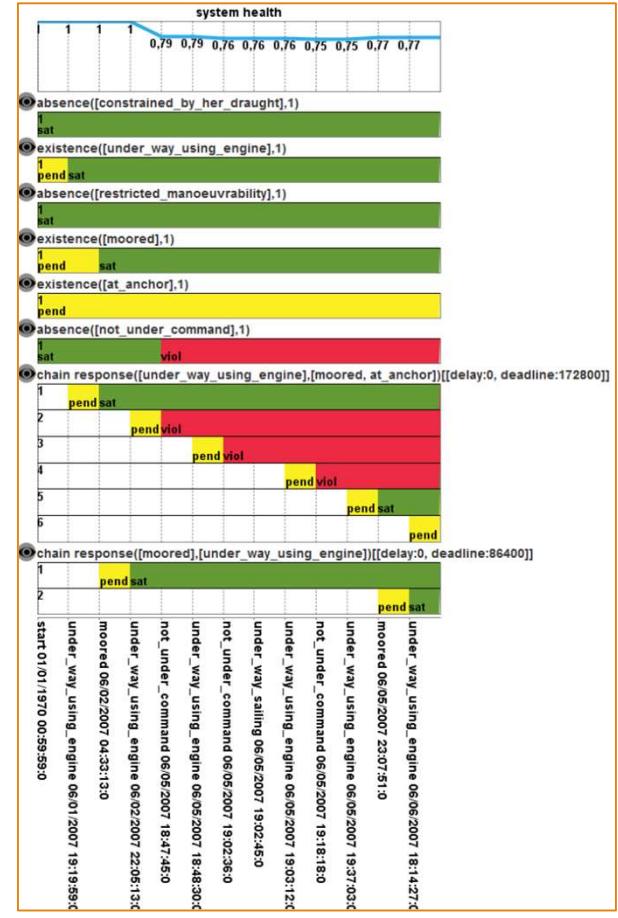
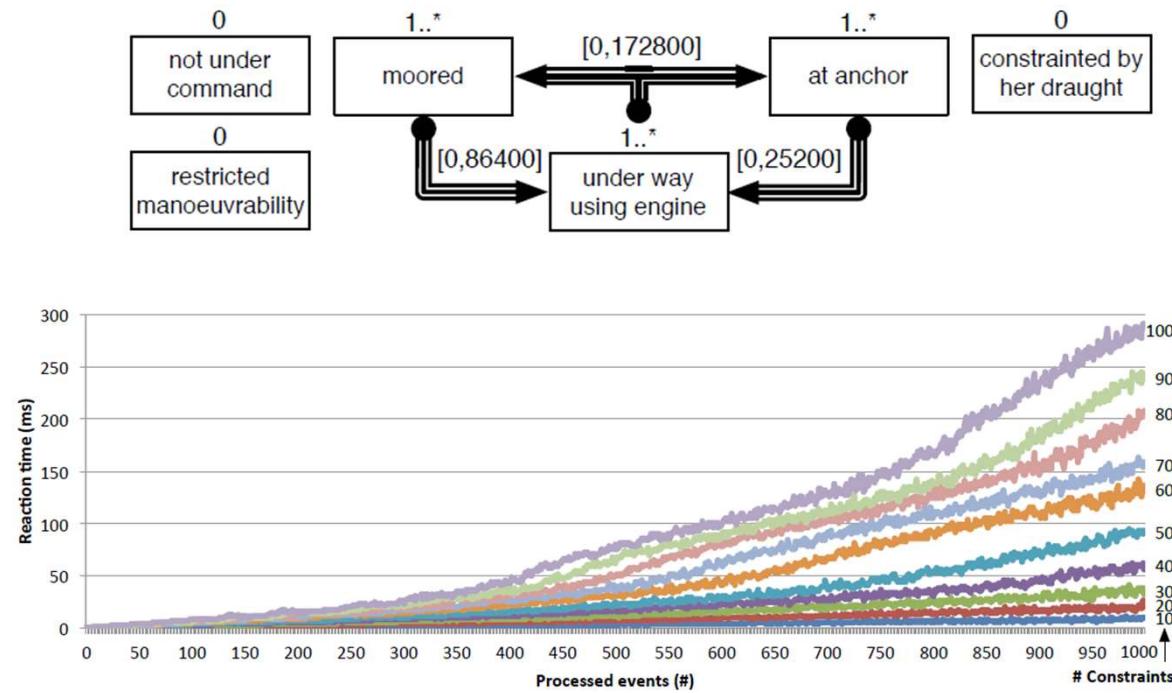
Allows deductive reasoning only

- Takes as input the domain-dependent axioms and the set of happened events...
- 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: Re-computes from scratch the results... computationally very costly !!!!

Reactive Event Calculus

- Luca Chittaro, Angelo Montanari: **Efficient Temporal Reasoning in the Cached Event Calculus.** Computational Intelligence 12: 359-382 (1996)
- Federico Chesani, Paola Mello, Marco Montali, Paolo Torroni: **A Logic-Based, Reactive Calculus of Events.** Fundam. Inform. 105(1-2): 135-161 (2010)
- Overcome the deductive nature of the original formulation given by Sergot & Kowalski
- New happened events can be added dynamically, i.e. the result is **updated** (and not re-computed from scratch)
- Allows events in a wrong order
- More efficient
- Can be implemented in backward reasoning as well as in forward reasoning

Event Calculus – Application: Monitoring



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Reactive Event Calculus and Drools

- Drools already overcomes a number of issues due to logic approaches
- EC provides any way a structured and clean way for addressing the representation of the state of a system

Take Home message: When needed...

- Cleanly separate Events from state properties (fluents)
- Determine which are the events that affect a fluent (i.e., they change its truth)
- Define a rule for any of these events, the consequence of the rule will be a special UpEvent (one for each fluent)
 - These rules can combine a number of conditions
- Define a rule that as a consequence of the UpEvent, will trigger the fluent to true
- Same for DownEvents
- Use UpEvents and DownEvents for meta-reasoning
- If possible, remove fluents from the WM, so as to avoid memory cluttering

Reactive Event Calculus and Drools – Case Study

The HABITAT Project

- Financed by the Region Emilia-Romagna for the technology transfer to middle size industries, companies, firms
- Two year project, still running, ends the 31 of July 2018
- Main goal: develop an eco-system of IoT with sensors, so as to implement smart homes that will ensure people a better living
- Target: Elderly, but still independent; Elderly, already assisted by caregivers
- Radar for indoor positioning
- Sensorized armchair, with posture evaluation
- Sensorized belt, for physical activity recording and analysis
- A Radio, as an output console
- Apps running on smartphones/smartwatch

Reactive Event Calculus and Drools – Case Study

Scenario

- Some parts of your flat are to be intended as red zones. For example, the area around the exit door.
- If the elder is alone, and he is trying to get out, send him a reminder of the things she/he should do before exiting (remember the keys, switch off the stove, close the windows)
- If the elder is attended by a caregiver, alarm the caregiver that she/he is trying to get out
- Implicit: elderly living alone are autonomous, elderly with a caregiver are not autonomous anymore

The following solutions have been developed in collaboration with Dr. Daniela Loretì.

Reactive Event Calculus and Drools – Case Study

Scenario

- Some parts of your flat are to be intended as red zones. For example, the area around the exit door.
- If the elder is alone, and he is trying to get out, send him a reminder of the things she/he should do before exiting (remember the keys, switch off the stove, close the windows)
- If the elder is attended by a caregiver, alarm the caregiver that she/he is trying to get out
- Keep track if the caregiver answer the alarm, or not...
- Which objects? Which fluents? What if the caregiver does not answer?

Reactive Event Calculus and Drools – Case Study

Scenario – Internal events (meta events)

```
/*
 * A user has been detected to be in red zone
 */
declare UserRedZoneUpEvent
    @propertyReactive
    @role(event)
    @timestamp(timestamp)
    user : User
    caregiver : User
    room : String
    since : Long
    timestamp : Long
end

/*
 * The red zone situation has been solved
 */
declare UserRedZoneDownEvent
    @propertyReactive
    @role(event)
    @timestamp(timestamp)
    user : User
    timestamp : Long
end
```



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Reactive Event Calculus and Drools – Case Study

Scenario – Fluents

```
/*
 * The fluent represent the state of the Notification
 */
declare UserRedZoneNotificationFl
    notification : NotificationMessage
    user : User
    room : String
    since : Long
    whenRaised : Long
    whenAcknowledged : Long
    whenIgnored : Long
end
```

Reactive Event Calculus and Drools – Case Study

Scenario – Rule detecting the RedZone

```
rule "Detect the user red zone condition upon LocationEvents - caregiver not  
any information"  
  
when  
  
    $posEv : PositionEvent ( $user : user, $room : locationURI,  
    $timestamp : timestamp ) from entry-point "EventStream"  
  
        $posFl : UserInRoomFluent (user == $user ,  
        $user.getRole() == User.Role.SUBJECT , room == $room, $room memberOf  
        $user.getRedZones() , $since : this.getSince().getTime() , $timestamp - $since  
        > $user.getTriggerTimeForRedZone($room) )  
  
        not ( $f1 : UserRedZoneNotificationFl ( $user == user , whenRaised <  
        $timestamp , (whenAcknowledged==null || whenAcknowledged>$timestamp),  
        (whenIgnored==null || whenIgnored>$timestamp) ))  
  
        not ( $evDown : UserRedZoneDownEvent ($user == user, this before  
        [0s,20s] $posEv) )  
  
        not ($cg : CaregiverPresenceFl( ) )  
  
then  
  
    UserRedZoneUpEvent userRedZoneUpEvent = new UserRedZoneUpEvent(  
    $user , null, $room, $since, System.currentTimeMillis() );  
    insert(userRedZoneUpEvent);  
  
end
```

Reactive Event Calculus and Drools – Case Study

Scenario – Rule detecting the RedZone

```
rule "Detect the user red zone condition upon LocationEvents - caregiver absent"
when
    $posEv : PositionEvent ( $user : user, $room : locationURI,
    $timestamp : timestamp ) from entry-point "EventStream"
        $posFl : UserInRoomFluent (user == $user ,
    $user.getRole() == User.Role.SUBJECT , room == $room, $room memberOf
    $user.getRedZones() , $since : this.getSince().getTime() , $timestamp - $since
    > $user.getTriggerTimeForRedZone($room) )
            not ( $f1 : UserRedZoneNotificationFl ( $user == user , whenRaised <
    $timestamp , (whenAcknowledged==null || whenAcknowledged>$timestamp),
    (whenIgnored==null || whenIgnored>$timestamp) ))
            not ( $evDown : UserRedZoneDownEvent ($user == user, this before
    [0s,20s] $posEv) )
                $cg : CaregiverPresenceFl( presence == false , since < $timestamp )
then
    UserRedZoneUpEvent userRedZoneUpEvent = new UserRedZoneUpEvent(
    $user , null, $room, $since, System.currentTimeMillis() );
        insert(userRedZoneUpEvent);
end
```

Reactive Event Calculus and Drools – Case Study

Scenario – Rule detecting the RedZone

```
rule "Detect the user red zone condition upon LocationEvents - caregiver present"
when
    $posEv : PositionEvent ( $user : user, $room : locationURI,
    $timestamp : timestamp ) from entry-point "EventStream"
        $posFl : UserInRoomFluent (user == $user ,
    $user.getRole() == User.Role.SUBJECT , room == $room, $room memberOf
    $user.getRedZones() , $since : this.getSince().getTime() , $timestamp - $since
    > $user.getTriggerTimeForRedZone($room) )
        not ( $f1 : UserRedZoneNotificationFl ( $user == user , whenRaised <
    $timestamp , (whenAcknowledged==null || whenAcknowledged>$timestamp),
    (whenIgnored==null || whenIgnored>$timestamp) ))
        not ( $evDown : UserRedZoneDownEvent ($user == user, this before
    [0s,20s] $posEv) )
        $cg : CaregiverPresenceFl( presence == true , since < $timestamp )
then
    UserRedZoneUpEvent userRedZoneUpEvent = new UserRedZoneUpEvent(
    $user , $cg.getCaregiver() , $room, $since, System.currentTimeMillis() );
    insert(userRedZoneUpEvent);
end
```

Reactive Event Calculus and Drools – Case Study

Scenario – Generating the notification

```
rule "generate the USER_RED_ZONE Notification"
when
    $ev: UserRedZoneUpEvent ( $user : user, $caregiver : caregiver,
$room : room, $since : since)
then
{
    if ($caregiver == null) {
        // send custom message to the user
    }
    else {
        // notify the caregiver, and then
        insert ( new UserRedZoneNotificationFl ( $notification,
$user, $room, $since, System.currentTimeMillis(), null, null) );
    }
}
end
```

Reactive Event Calculus and Drools – Case Study

Scenario – Managing the Ok answer

```
rule "UserRedZone Answer: OK"
when
    $ans : AnswerMessage ( answer == AnswerMessage.ANSWER.YES,
    $idRefMessage : idRefMessage, $timestamp : timestamp ) from entry-point
    "EventStream"

    $f1 : UserRedZoneNotificationF1 ( notification.getIdSEPA() ==
    $idRefMessage , whenRaised < $timestamp , whenAcknowledged==null ,
    whenIgnored == null , $user : user)

then
    $f1.setWhenAcknowledged(System.currentTimeMillis());
    UserRedZoneDownEvent de = new UserRedZoneDownEvent( $user,
    System.currentTimeMillis());
    insert (de);
end
```

Reactive Event Calculus and Drools – Case Study

Scenario – Managing the “not any” answer

```
rule "UserRedZone Answer: NOT ANY"
no-loop
when
    $evUp : UserRedZoneUpEvent ( $user : user, $room : room, $since : since)
    $f1 : UserRedZoneNotificationF1 ( $notification: notification,
whenAcknowledged==null , whenIgnored == null , $user == user, $room == room)
    not $ans : AnswerMessage ( idRefMessage == $notification.getIdSEPA() ,
$timestamp : timestamp, this after[0s, 30s] $evUp )   from entry-point
"EventStream"
then
    $f1.setWhenIgnored(System.currentTimeMillis());
    UserRedZoneDownEvent de = new UserRedZoneDownEvent( $user,
System.currentTimeMillis());
    insert (de);
end
```

Reactive Event Calculus and Drools – Case Study

Scenario – Closing the notification fluent

```
rule "UserRedZone Closing the advice fluent"
no-loop
when
    $ev : UserRedZoneDownEvent( $user : user, $timestamp : timestamp)
    $f1 : UserRedZoneNotificationF1 ( user == $user , whenRaised <
$timestamp)
then
    //delete ( $ev );
end
```



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Federico Chesani

DISI – Department of Computer Science and Engineering

federico.chesani@unibo.it

www.unibo.it



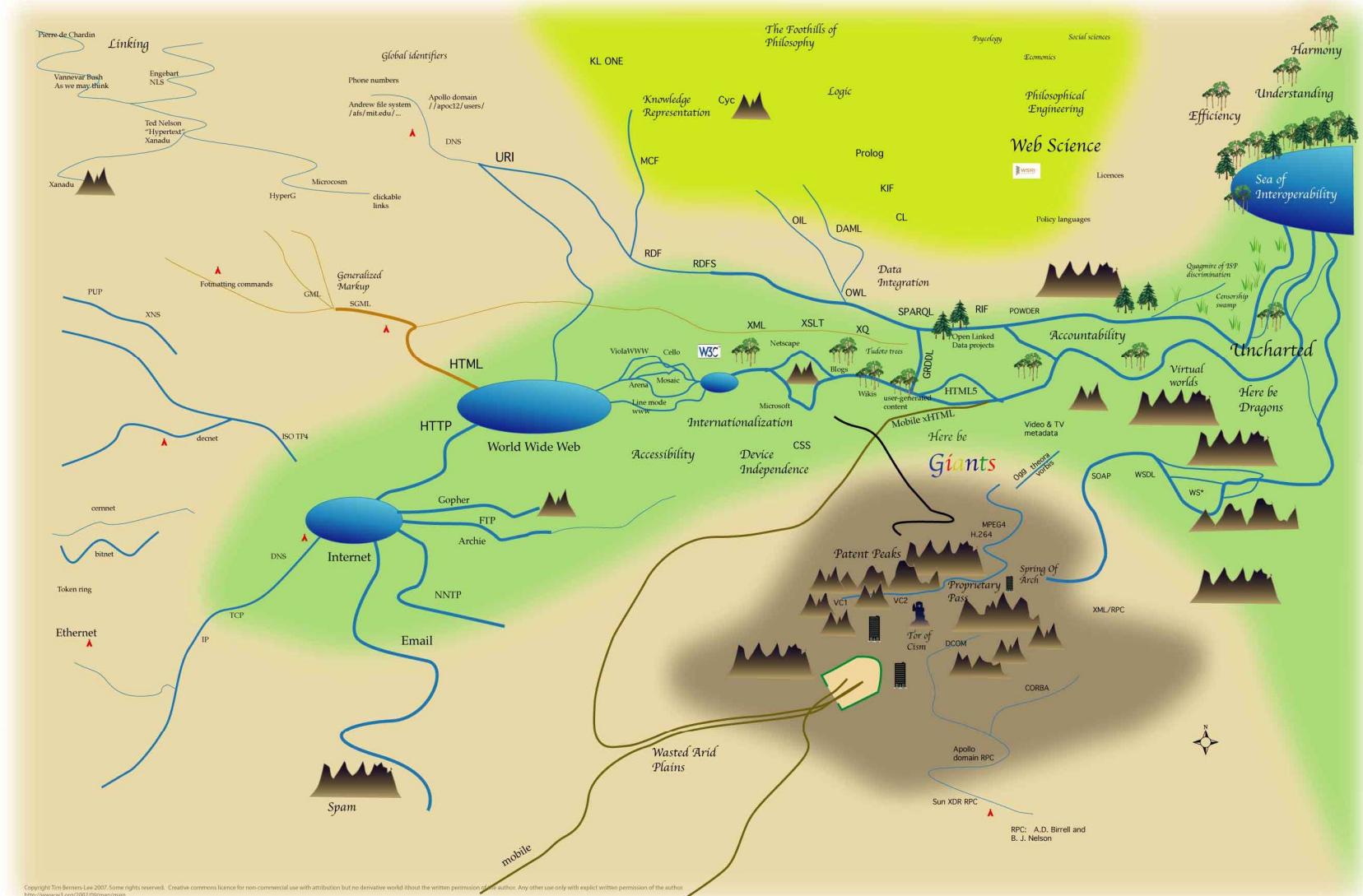
ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Semantic Web and Knowledge Graphs

Federico Chesani

Semantic Web

The Web as depicted by Berners-Lee in 2007

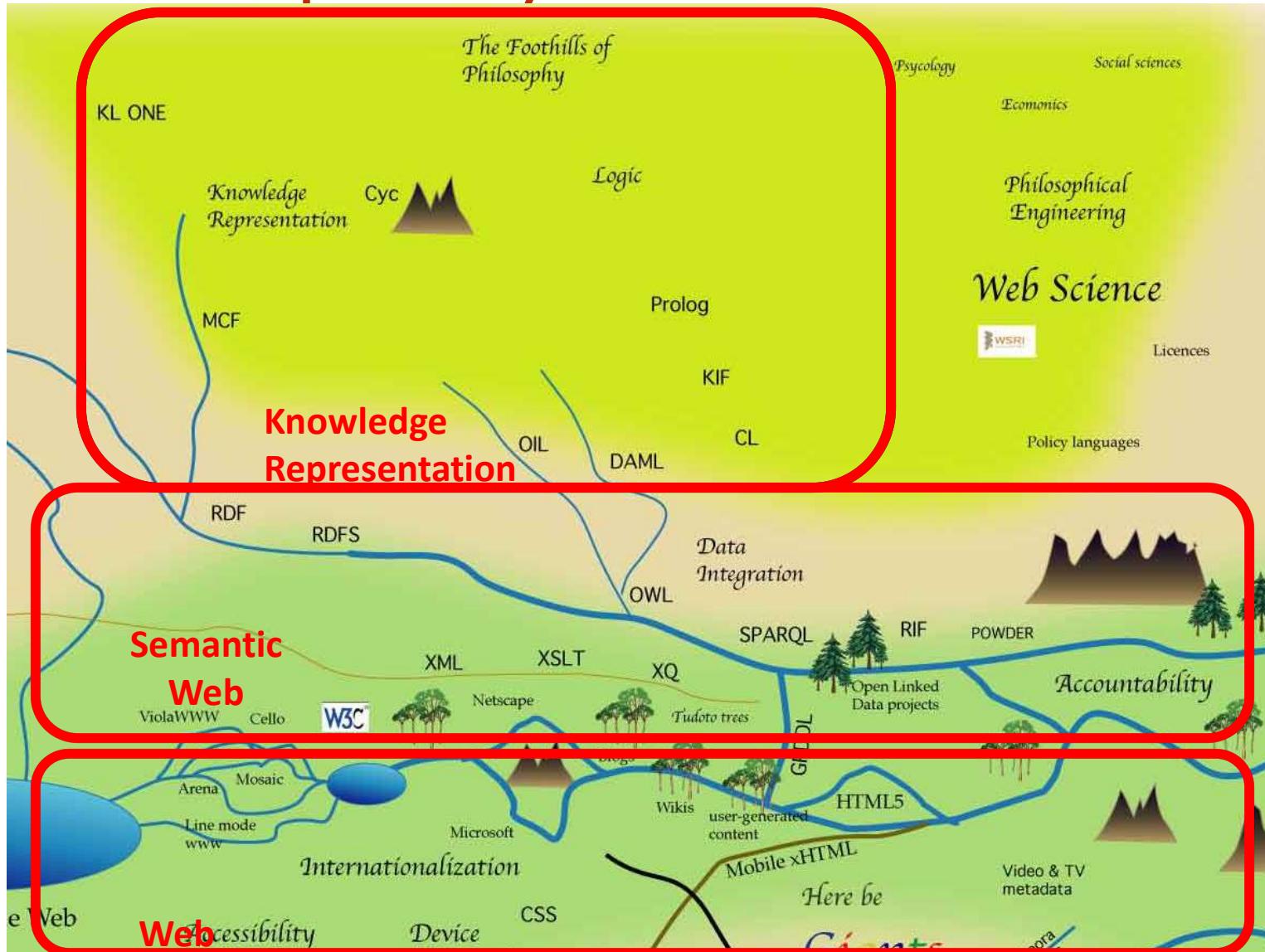


©Tim Berners-Lee, <http://www.w3.org/2007/09/map/main.jpg>



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The Web as depicted by Berners-Lee in 2007



©Tim Berners-Lee, <http://www.w3.org/2007/09/map/main.jpg>



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The Web 1.0

Information represented by means of:

- Natural language
- Images, multimedia, graphic rendering/aspect

Human Users easily exploit all this means for:

- Deducting facts from partial information
- Creating mental associations (between the facts and, e.g., the images)
- They use different communication channels at the same time (contemporary use of many primitive senses)



The Web 1.0

The content is published on the web with the principal aim of being “human-readable”

- Standard HTML is focused on *how* to represent the content
- There is no notion of *what* is represented
- Few tags (e.g. <title>) provide an implicit semantics but ...
 - ... their content is not structured
 - ... their use is not really standardized



The Web 1.0

Web pages contain also links to other pages, but ...

- No information on the link itself ...
 - ... what does a link represent?
 - ... what does the linked page/resource represent?
- E.g.: in my home page there are links to other home pages ...
 - Which ones link to colleagues?
 - Which ones link to friends?



The Web 1.0

Actual Web = Layout + Routing

The problem: it is not possible to
automatically reason about the data

But the web is indeed an immense knowledge base... it's there, it's free...



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The Web 1.0

The web is *global*

- Any page can link to anything
- Approximatively, anyone can publish anything on the web, about any topic
 - *Distribution* of the information
 - *Inconsistency* of the information
 - *Incompleteness* of the information
- Some recent attempts to limit such freedom (with mixed results)



Semantic Web

Goal: “use” and “reason upon” all the available data on the internet automatically

How? By extending the current web with knowledge about the content (semantic information)



Semantic Web

“The Semantic Web is about two things. It is about common formats for integration and combination of data drawn from diverse sources, where on the original Web mainly concentrated on the interchange of documents. It is also about language for recording how the data relates to real world objects. That allows a person, or a machine, to start off in one database, and then move through an unending set of databases which are connected not by wires but by being about the same thing.”

Source: W3C Semantic Web Initiative



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Semantic Web

Principles SW would like to preserve:

- Globality
- Information distribution
- Information inconsistency
 - Content inconsistency
 - Link inconsistency
- Information incompleteness
 - ... of contents
 - ... of routing information (links)



Adding information about the content...

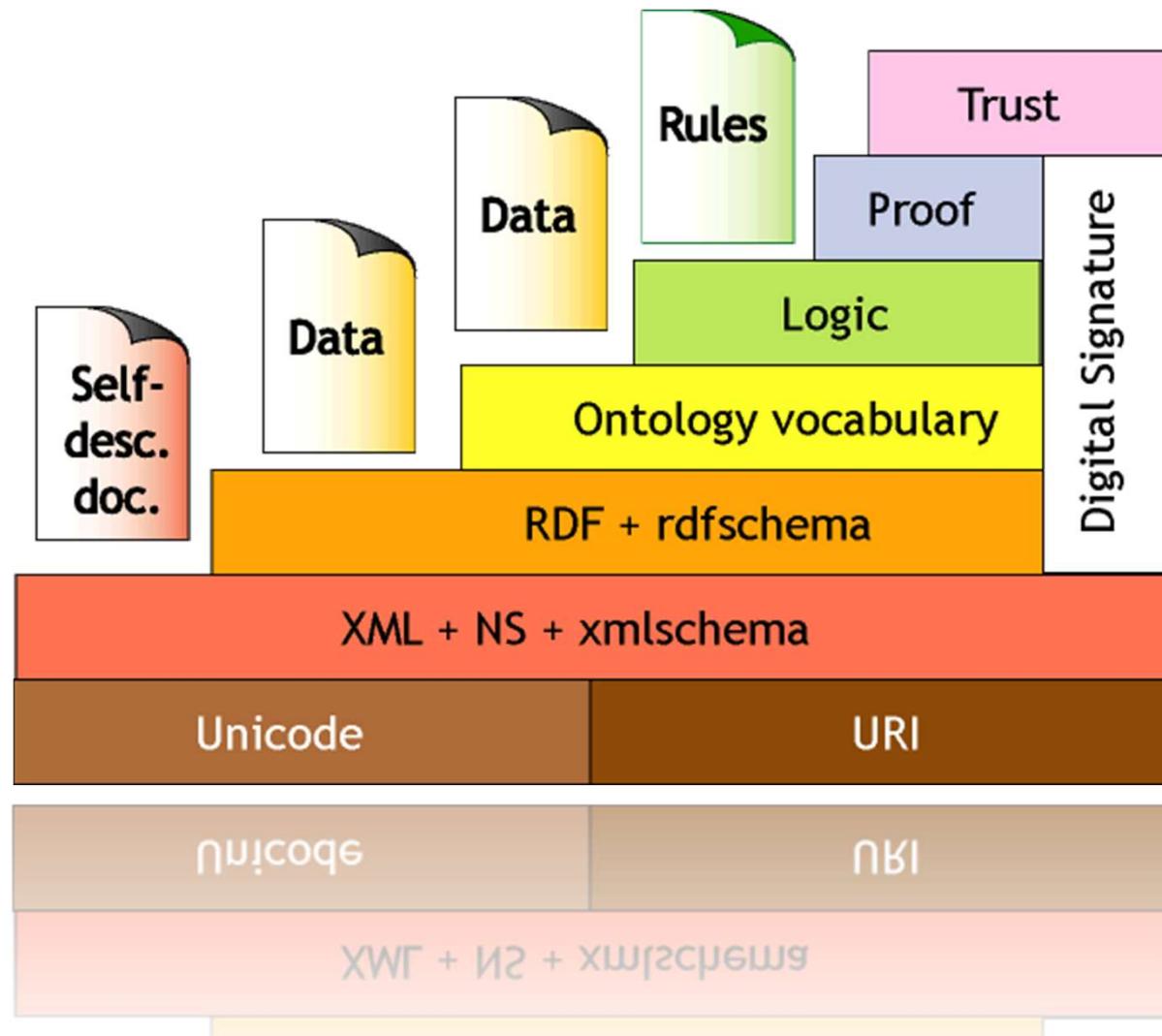
Adding information is not enough

- Information should be structured (e.g., Linneo classification for the living world)
 - *Ontologies!!!!*
- There is the need of some inference mechanism (e.g., syllogism, FOL, DL)
 - *Logic!!!! (and DL in particular)*
- We should be able to infer new knowledge
 - We need the *proofs* that originated such new knowledge



Semantic Web Tools

Recalling the Semantic Web Cake



A unique way for identifying concepts

- How to uniquely identify concepts?
 - > by means of a name system ...
- SW exploits an already available name systems, URIs (*Uniform Resource Identifier*)
 - By definition, URI guarantees unicity of the names
 - To each URI corresponds *one and only* one concept ...
 - ... but more URI can refer to the *same* concept!
 - NOTE: differently from the web, it is not necessary that to each URI corresponds some content!

Examples:

<http://www.repubblica.it>
federico.chesani@unibo.it
ISBN 88-7750-483-8

eXtensible Markup Language - XML

- Created for supporting data exchange between heterogeneous systems (hardware and software)
 - No presentation information
 - Human readable and machine readable
- Extensible, so that anyone can represent any type of data
- Hierarchically structured by means of *tags*
- An XML document can contain, in a preamble, a description of the grammar used in such document (optional) (self-describing document!!!)
- Very mature technology!

Resource Description Framework (RDF/RDFS)

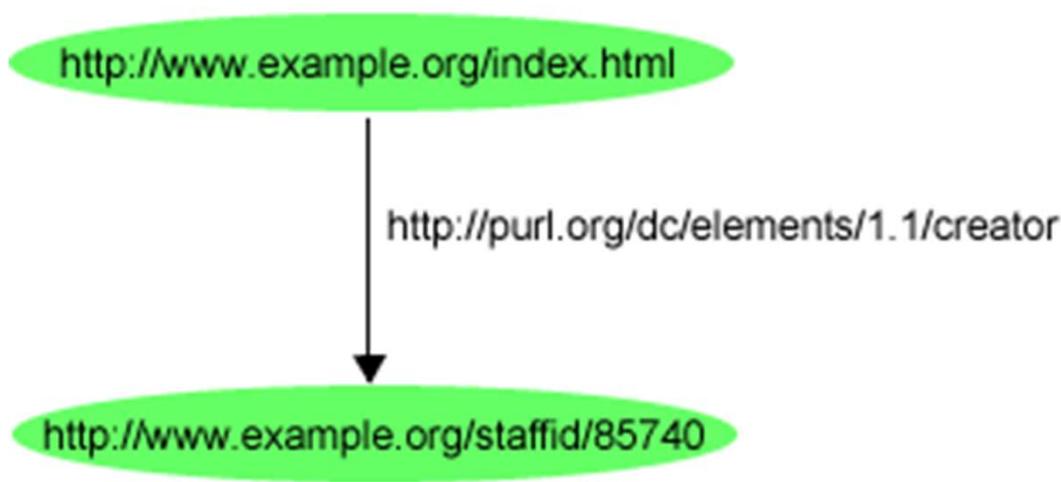
- Standard W3C
- XML-based language for representing “knowledge”
- A design criteria: provide a “minimalist” tool
- Based on the concept of triple:

< subject, predicate, object >

< resource, attribute, value >
- Some different representations (N3, Graph, RDF/XML)

RDF – Graph Representation

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



`http://www.example.org/index.html` has a `creator`
whose value is `John Smith`

RDF – Graph Representation

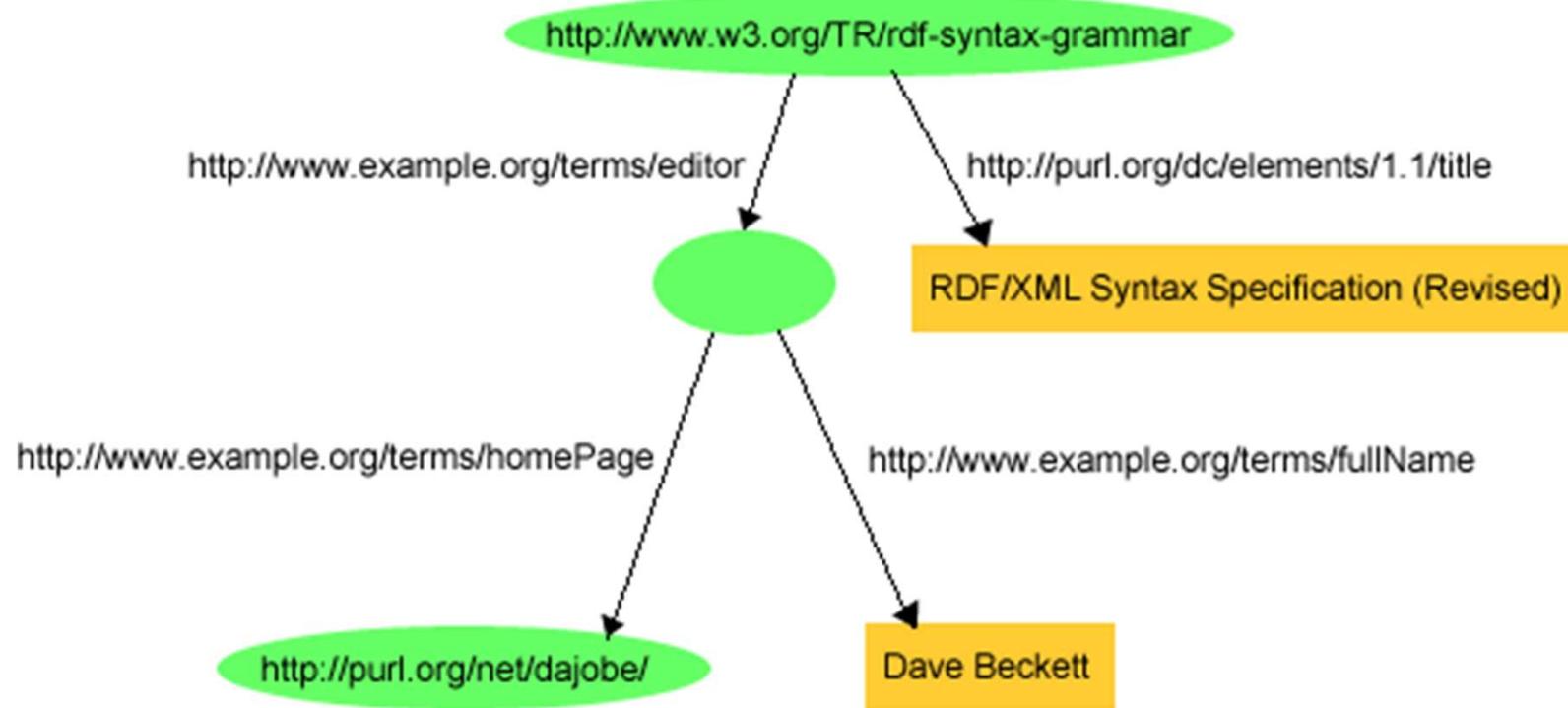


RDF – XML Representation

```
<rdf:RDF  
    xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#  
    xmlns:contact=http://www.w3.org/2000/10/swap/pim/contact#  
>  
  
<contact:Person    rdf:about="http://www.w3.org/People/EM/contact#me">  
    <contact:fullName>Eric Miller</contact:fullName>  
    <contact:mailbox rdf:resource="mailto:em@w3.org"/>  
    <contact:personalTitle>Dr.</contact:personalTitle>  
</contact:Person>  
</rdf:RDF>
```

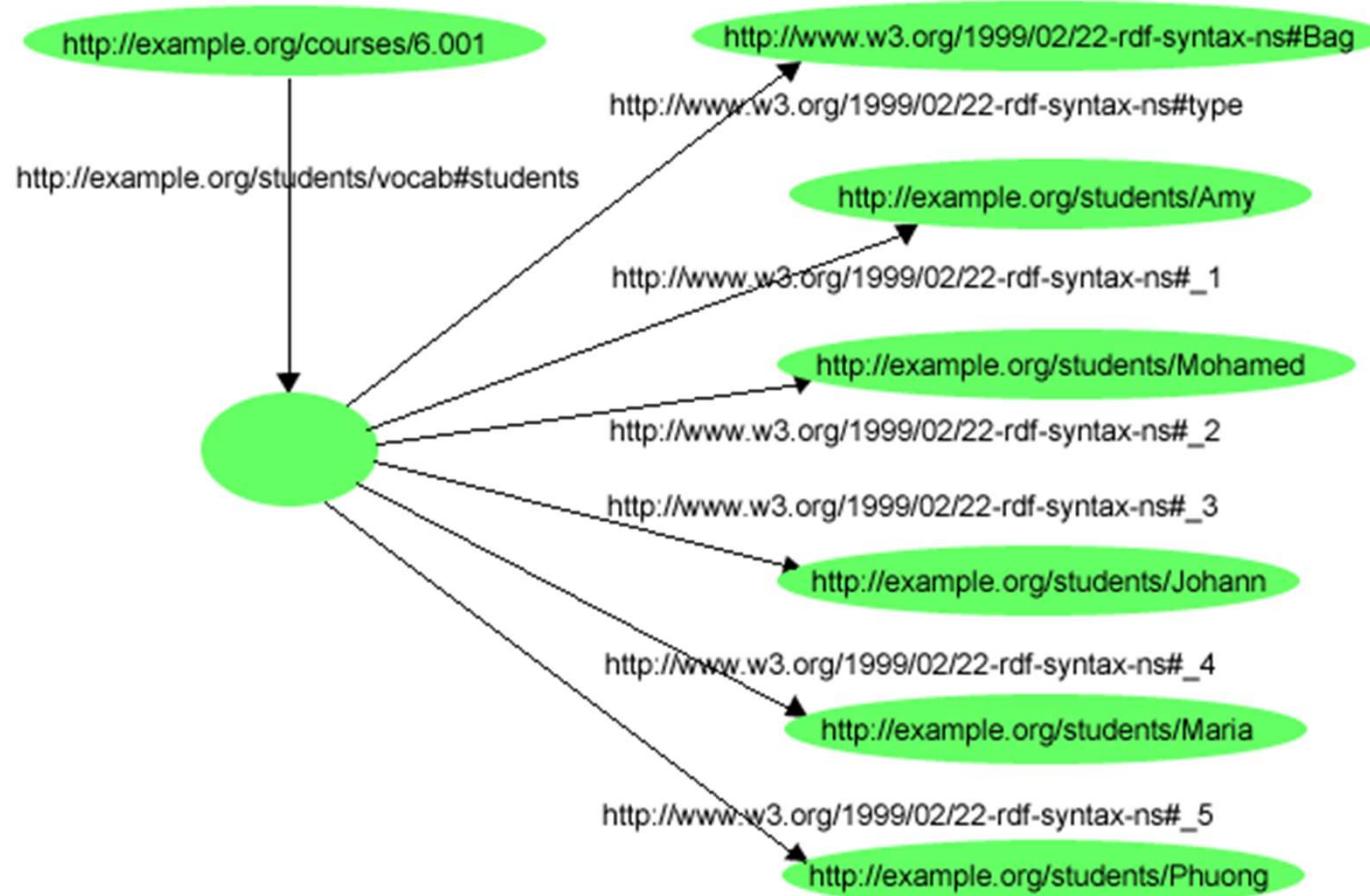
I can query for the mailbox of Eric Miller, without knowing a priori if he uses mailbox or email ...
... if Eric Miller will change mailbox, serach result will be coherent!

RDF - Examples



Empty Nodes

RDF – Examples



Bags/Sets

RDF – Expressive Power

RDF supports:

- **Types** (classes) by means of the attribute **type** (that assume as value an URI)
- Subject/object of a sentence can be also **collections** (bag, sequence, alternative)
- ***Meta-sentences***, through *reification* of teh sentences (“Marco says that Federico is the author of web page xy”)

RDF Schema

- RDF can be intended also as a description of resource attributes and of the values of such attributes
- RDFS allows to describe classes and relations with other classes/resources
 - *type*
 - *subClassOf*
 - *subPropertyOf*
 - *range*
 - *domain*

RDF and E/R Models

- Many similarities with E/R models ...
 - ... RDF is more expressive
- RDF to be intended as the “E/R” for the web
- Relations in RDF are “first class entities”
- In RDF the list of properties of an entity is not:
 - A priori determined by the developer
 - Centralized (DB)
 - Consequence of the fact that any one can assert anything about any one else

RDF and Relational Databases

There is a direct mapping with relational db

- A record is viewed as a RDF node
- The name of a table column is viewed as `rdf:propertyType`
- The corresponding field value is intended as the value of the property
- RDF aims to integrate different databases with different underlying model
 - Traditional DBMS are optimized for creating new data models within the same db or within a restricted set of dbs

RDF Tools

Many tools already available ...

Only in the W3C wiki there are citations for:

- 38 Frameworks/reasoners
- 47 RDF Triple Stores

Have a look to

<http://www.w3.org/2001/sw/wiki/Tools>

RDFa

- RDFa is a specification for attributes to express structured data in XHTML.
- The rendered, hypertext content of XHTML is reused by the RDFa markup
 - publishers don't need to repeat significant data in the document.

Source: RDFa Primer

<http://www.w3.org/TR/2008/NOTE-xhtml-rdfa-primer-20081014/>

RDFa

...

All content on this site is licensed under

 a Creative Commons License
.

...

All content on this site is licensed under

 a Creative Commons License
.

This page has a **relation** of type **license** with the page at creative commons...

Source: RDFa Primer

<http://www.w3.org/TR/2008/NOTE-xhtml-rdfa-primer-20081014/>

RDFa

```
...
<div>
    <h2> The trouble with Bob </h2>
    <h3> Alice </h3>
    ...
</div>
```

```
<div xmlns:dc="http://purl.org/dc/elements/1.1/">
    <h2 property="dc:title"> The trouble with Bob </h2>
    <h3 property="dc:creator"> Alice </h3>
    ...
</div>
```

Note the reference to the DC namespace, i.e. the Dublin Core initiative
<http://dublincore.org/>

Source: RDFa Primer

<http://www.w3.org/TR/2008/NOTE-xhtml-rdfa-primer-20081014/>

SPARQL

- SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as [RDF](#) or viewed as [RDF](#) via middleware.
- SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions.
- Supports extensible value testing and constraining queries by source [RDF](#) graph.
- The results of SPARQL queries can be results sets or [RDF](#) graphs.

Source: SPARQL W3C Working group

<http://www.w3.org/2001/sw/wiki/SPARQL>

<http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>

SPARQL

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 . }
```

Source: SPARQL W3C Working group

<http://www.w3.org/2001/sw/wiki/SPARQL>

<http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>

Ontology Web Language (OWL 1.0)

- Standard W3C
- Based upon/extend RDF/RDFS
- Formal Semantics (*Description Logic Fragments*)
- Three level of expressivity/complexity
 - OWL Lite
 - OWL DL
 - OWL Full
- OWL 2.0 è già standard affermato

OWL – Features

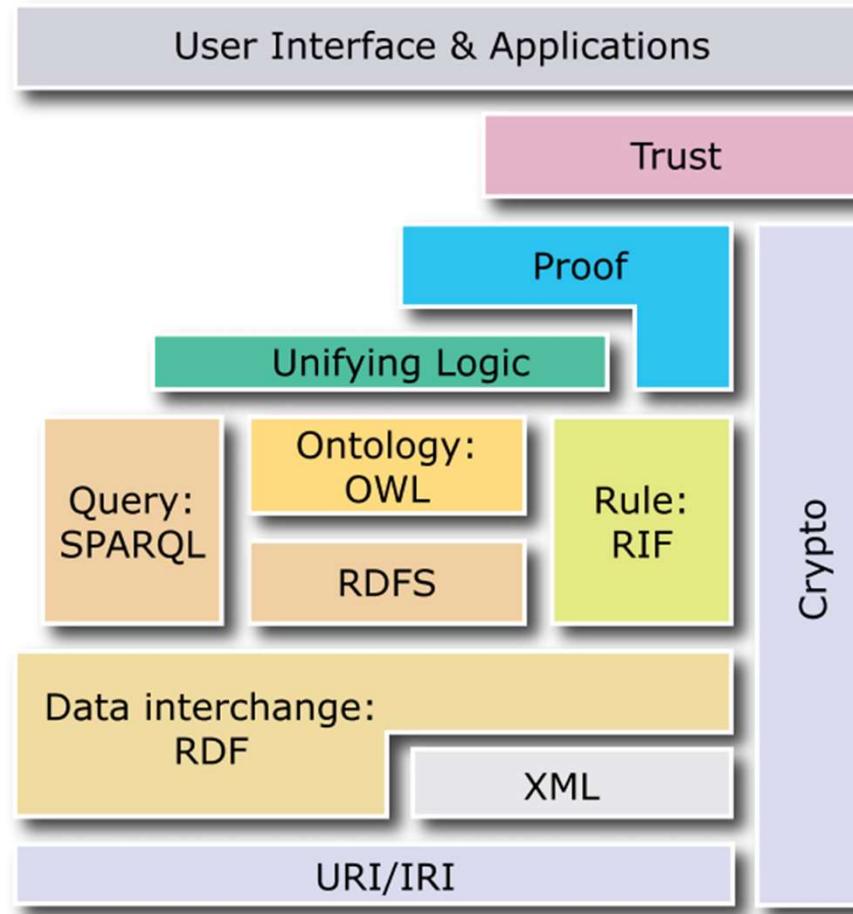
- **Classes (categories)**: subClassOf, intersectionOf, unionOf, complementOf, enumeration, equivalence, disjoint
- **Properties (Roles, Relations)**: symmetric, transitive, functional, inverse Functional, range, domain, subPropertyOf, inverseOf, equivalentProperty
- **Instances (Individuals)**: sameIndividualAs, differentFrom, allDifferent

OWL Tools

- Many tools for OWL
 - Editors (37 listed at <http://www.w3.org/2001/sw/wiki/Category:Editor>)
 - Reasoners (39 listed at <http://www.w3.org/2001/sw/wiki/Category:Reasoner>)
- Quite often integrated in a comprehensive framework

A well known (but not necessarily the best one) ontology editor:
Protégé <http://protege.stanford.edu/>

The Semantic Web Cake



Knowledge Graphs

Knowledge Graphs

Suggested Readings:

1. Natasha Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, and Jamie Taylor. 2019. **Industry-scale knowledge graphs: lessons and challenges.** Commun. ACM 62, 8 (July 2019), 36–43. DOI: <https://doi.org/10.1145/3331166>
2. Dieter Fensel, Umutcan Simsek, Kevin Angele, Elwin Huaman, Elias Kärle, Oleksandra Panasiuk, Ioan Toma, Jürgen Umbrich, Alexander Wahler: **Knowledge Graphs - Methodology, Tools and Selected Use Cases.** Springer 2020, ISBN 978-3-030-37438-9, pp. 1-147



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Knowledge Graphs – technological push

- 2012: At Google they understand the need of overcoming search approaches based on statistical data retrieval only...
- ... they need something able to represent and reason upon knowledge...
- The Semantic Web stack seems too complex, and not so fast for the Google needs
 - Reasoning on the T-Box is computationally expensive, and might even not terminate
- They mentioned the term "Knowledge Graphs" in 2012



Knowledge Graphs – business push

- Early 2000 have shown the win of statistical methods over KB methods in accessing the web
- Google business model:
 - People make a search
 - Google replies with urls and **ads**
 - People click on urls, and leave
- Limit of this model: users arrive and leave quite early



Knowledge Graphs – business push

- Transformation of Google, since 2011: from a **search engine**, to a **query-answering engine**
- Provide the users with the answer, and keep them in google pages....

The screenshot shows a Google search results page. The search bar contains the query "where does Federico Chesani work". Below the search bar, there are navigation links for All, Images, News, Maps, Videos, More, Settings, and Tools. It indicates that there are about 23,300 results found in 0.63 seconds. A featured snippet box highlights information about Federico Chesani, stating he is an assistant professor of Computer Science at DISI — University of Bologna since April 2012. Below this, a blue link provides the source: "www.sciencedirect.com › science › article › abs › pii A distributed approach to compliance monitoring of business ...". At the bottom of the page, there are links for "About Featured Snippets" and "Feedback", along with the University of Bologna logo and its name.

Google where does Federico Chesani work

All Images News Maps Videos More Settings Tools

About 23,300 results (0.63 seconds)

Federico Chesani, Ph. D. in Computer Science, is assistant professor of Computer Science at DISI — University of Bologna since April 2012.

www.sciencedirect.com › science › article › abs › pii
A distributed approach to compliance monitoring of business ...

About Featured Snippets Feedback

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Knowledge Graphs – the Google approach

How Google answer the two previous questions?

- Create a common, simple vocabulary... schema.org
- Create a **simple**, but **robust** corpus of types, properties, etc.
- Push the web to adopts these standards (well, after all it is Google!)

Based on annotations, Google KG is reported to contains 100B facts about 1B entities



Knowledge Graphs

Basic idea: just store the information in terms of nodes and arcs connecting the nodes.

Logical formulas are missing...

...T-Box and A-Box are stored at the same "level".

- Billions of factual knowledge, in form of "triplettes"
- No conceptual schema: data from various sources, with different semantics can be mixed freely (avoids the knowledge acquisition bottleneck)
- **No reasoning**
- Graph algorithms used to fast traverse the graph, looking for the solution
- "Embedding" allows to represent such graphs in terms of n-grams, and then apply ML techniques



Knowledge Graphs

Definition of Fensel and colleagues:

"Knowledge Graphs are very large semantic nets that integrate various and heterogeneous information sources to represent knowledge about certain domains of discourse."



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Knowledge Graphs – "open" examples

- **DBpedia** (<http://dbpedia.org/>), defined as the "de facto central dataset on the Semantic Web"
 - October 2016 release counts 13B RDF triples
- **Freebase** (<http://www.freebase.com/>) started as a collaborative KB, then it was acquired by Google, and dismissed in 2016. Its knowledge was used to improve/extend Google's KG
- **YAGO** (<https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/downloads/>), also based on Wikipedia
- **NELL** (<http://rtw.ml.cmu.edu/rtw/resources>)
- **Wikidata** (https://www.wikidata.org/wiki/Wikidata:Main_Page), at August 2019 it counts 7B triples
- **KBpedia** (<http://www.kbpedia.org/>), offers a bridge towards any other KB (open and commercial)
- **Datacommons.org**, launched by Google in 2018, and integrates knowledge about geographic and administrative areas, demographics, and other public available. Accessible through a browser interface. Each fact brings along the provenance.



Knowledge Graphs – commercial examples

- Cyc (<http://www.cyc.com/>) "one of the longest-living AI projects", provides a common sense knowledge base.
- Facebook's Entities Graphs (<http://www.facebook.com/notes/facebook-engineering/under-the-hood-the-entities-graph/10151490531588920>), used internally by Facebook, stores data about users, interests, and connections. Available through Facebook Graph API
- Google's Knowledge Graph (<https://developers.google.com/knowledge-graph/>), in 2016 Google claimed it holds 70B facts. It is not known how information is stored, but it uses schema.org for terms. Accessible through the Google knowledge Graph API.



Examples of (fairly large!!!) Industry-scale Knowledge Graphs (from [1])

Common characteristics of the knowledge graphs.

	Data model	Size of the graph	Development stage
Microsoft	The types of entities, relations, and attributes in the graph are defined in an ontology.	~2 billion primary entities, ~55 billion facts	Actively used in products
Google	Strongly typed entities, relations with domain and range inference	1 billion entities, 70 billion assertions	Actively used in products
Facebook	All of the attributes and relations are structured and strongly typed, and optionally indexed to enable efficient retrieval, search, and traversal.	~50 million primary entities, ~500 million assertions	Actively used in products
eBay	Entities and relation, well-structured and strongly typed	Expect around 100 million products, >1 billion triples	Early stages of development and deployment
IBM	Entities and relations with evidence information associated with them.	Various sizes. Proven on scales documents >100 million, relationships >5 billion, entities >100 million	Actively used in products and by clients



Google's Knowledge Graph – an example

Google Search More

All Maps Images News Videos More Settings Tools

About 153,000 results (0.65 seconds)

www.trattoriaboni.it ▾ [Translate this page](#)
Trattoria Boni Bologna: Home
Una delle più vecchie trattorie bolognesi dove poter riscoprire sapori e profumi del bel tempo che fu. Un tempo nel quale la "nouvelle cuisine" non era ancora ...

www.trattoriaboni.it/index.php/menu ▾ [Translate this page](#)
Menu - Trattoria Boni Bologna
Una delle più vecchie trattorie bolognesi dove poter riscoprire sapori e profumi del bel ...
Piatto Rustico di **Boni** (tigelle e crescentine con affettati) SOLO A CENA.

www.tripadvisor.com/.../Bologna/Bologna-Restaurants ▾
Trattoria Boni, Bologna - Menu, Prices & Restaurant Reviews ...
★★★★★ Rating: 4 - 857 reviews - Price range: \$\$ - \$\$\$
Trattoria Boni, Bologna: See 857 unbiased reviews of **Trattoria Boni**, rated 4 of 5 on Tripadvisor and ranked #210 of 1830 restaurants in Bologna.
You visited this page on 2/18/20.

ristadvisor.it/Trattoria-Boni_Bologna ▾ [Translate this page](#)
Trattoria Boni Bologna
Trattoria Boni - Bologna cucina tipica bolognese. in questo locale si va sul sicuro si mangia alla grande con piatti tipici bolognesi tortellini tagliatelle bollito e ...

[it-it.facebook.com/Luoghi/Bologna/Ristorante italiano](http://it-it.facebook.com/Luoghi/Bologna/Ristorante-italiano)
Trattoria Boni - Bologna - Menù, prezzi, recensioni dei ...



See photos See outside

Trattoria Boni

[Sito web](#) [Indicazioni stradali](#) [Salva](#)

4.3 ★★★★★ 1,129 recensioni Google

€€ · Ristorante di cucina tradizionale

Indirizzo: Via Don Luigi Sturzo, 22, 40135 Bologna BO

Orari: **Closed** · Opens 12PM ▾

Orari suggeriti da un utente

Mon Closed

Tue-Sat 12-2:45PM and 7:30-10:30PM

Sun 12-2:45PM

Telefono: 051 615 4337

[Suggerisci una modifica](#) · Sei il proprietario di quest'attività?

Where does this information come from?



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Some key questions when evaluating the *quality* of a Knowledge Graph

- **Coverage**

Does the graph have all the required information? If not, can we estimate how much of it?

Usually the answer to the former question is NO...

- **Correctness**

Is the information correct?

Correctness... it depends on the type of information (it can be objective or subjective), and it depends on the *intended use* of the information

- **Freshness**

Is the content up-to-date?

Keep in mind the change rate of the domain: it is different to represent stock quotes, and parenthood



Google KG API

An example from <https://developers.google.com/knowledge-graph>:

```
"""Example of Python client calling Knowledge Graph Search API."""
from __future__ import print_function
import json
import urllib

api_key = open('.api_key').read()
query = 'Taylor Swift'
service_url = 'https://kgsearch.googleapis.com/v1/entities:search'
params = {
    'query': query,
    'limit': 10,
    'indent': True,
    'key': api_key,
}
url = service_url + '?' + urllib.urlencode(params)
response = json.loads(urllib.urlopen(url).read())
for element in response['itemListElement']:
    print(element['result']['name'] + ' (' + str(element['resultScore']) + ')')
```



Google KG API

An example from <https://developers.google.com/knowledge-graph>:

```
{  
  "@context": {  
    "@vocab": "http://schema.org/",  
    "goog": "http://schema.googleapis.com/",  
    "resultScore": "goog:resultScore",  
    "detailedDescription": "goog:detailedDescription",  
    "EntitySearchResult": "goog:EntitySearchResult",  
    "kg": "http://g.co/kg"  
  },  
  "@type": "ItemList",  
  "itemListElement": [  
    {  
      "@type": "EntitySearchResult",  
      "result": {  
        "@id": "kg:/m/0dl567",  
        "name": "Taylor Swift",  
        "@type": [  
          "Thing",  
          "Person"  
        ],  
        "description": "Singer-songwriter",  
        "image": {  
          "contentUrl": "https://t1.gstatic.com/images?q=tbn:ANd9GcQmVDAhjhWnN2OWys2ZMO3PGAhupp5tN2LwF_BJmiHgi19hf8Ku",  
          "url": "https://en.wikipedia.org/wiki/Taylor_Swift",  
          "license": "http://creativecommons.org/licenses/by-sa/2.0"  
        },  
        "detailedDescription": {  
          "articleBody": "Taylor Alison Swift is an American singer-songwriter and actress. Raised in Wyomissing, Pennsylvania, she moved to Nashville, Tennessee, at the age of 14 to pursue a career in country music.",  
          "url": "http://en.wikipedia.org/wiki/Taylor_Swift",  
          "license": "https://en.wikipedia.org/wiki/Wikipedia:Text_of_Creative_Commons_Attribution-  
ShareAlike_3.0_Unported_License"  
        },  
        "url": "http://taylorswift.com/"  
      },  
      "resultScore": 4850  
    }  
  ]  
}
```

