Intelligenza Artificiale

Sistemi a regole
Sistemi esperti

Marco Piastra

Sistemi a regole – Sistemi esperti

1. Sistemi a regole

2. Jess

3. Fox, Goat and Cabbage (esercitazione Jess)

1

Sistemi a regole
(*Production Systems*)

Logica, Prolog e sistemi a regole

- La logica simbolica:
  - è un sistema per la rappresentazione formale del ragionamento
  - si basa su un formalismo di rappresentazione e su regole di derivazione sintattica (regole di inferenza)
- Il Prolog:
  - si basa sui principi della logica simbolica (calcolo dei predicati del primo ordine)
  - impiega una regola di inferenza speciale (tipicamente il principio di risoluzione)
  - rappresenta un 'linguaggio di programmazione' di uso generale
- I sistemi a regole
  - adottano una forma semplificata di programmazione logica
  - sono stati concepiti per una classe di applicazioni particolari (*sistemi esperti* o *expert systems*)

Logica - Principi generali e fatti specifici

- In una rappresentazione logico-simbolica
  (p.es. un programma Prolog)
  è tipicamente possibile distinguere:
  - la rappresentazione di *principi generali*
  - la rappresentazione di *fatti specifici*
- Esempio:

  principi generali :

  $\forall x \forall y (((\text{madre}(x) = \text{madre}(y)) \wedge (\text{padre}(x) = \text{padre}(y)) \leftrightarrow \text{StessiGenitori}(x, y))$

  $\forall x \forall y ((\text{Maschio}(x) \wedge \text{StessiGenitori}(x, y)) \leftrightarrow \text{Fratello}(x, y))$

  $\forall x \forall y ((\text{Femmina}(x) \wedge \text{StessiGenitori}(x, y)) \leftrightarrow \text{Sorella}(x, y))$

  fatti specifici :

  Femmina(Amelia); Femmina(Alba); Femmina(Paola); Maschio(Mario)

  (madre(Amelia) = Paola); (madre(Alba) = Paola)

  (padre(Amelia) = Mario); (padre(Alba) = Mario)

Regole di produzione

- Un sistema a regole contiene un insieme di regole di produzione
- Ciascuna regola ha la forma
  - <condizioni> $\Rightarrow$ <azioni>
  - talvolta anche descritte come
    <RHS - Right Hand Side> $\Rightarrow$ <LHS - Left Hand Side>
  - condizioni e azioni sono in forma normale congiuntiva (CNF)

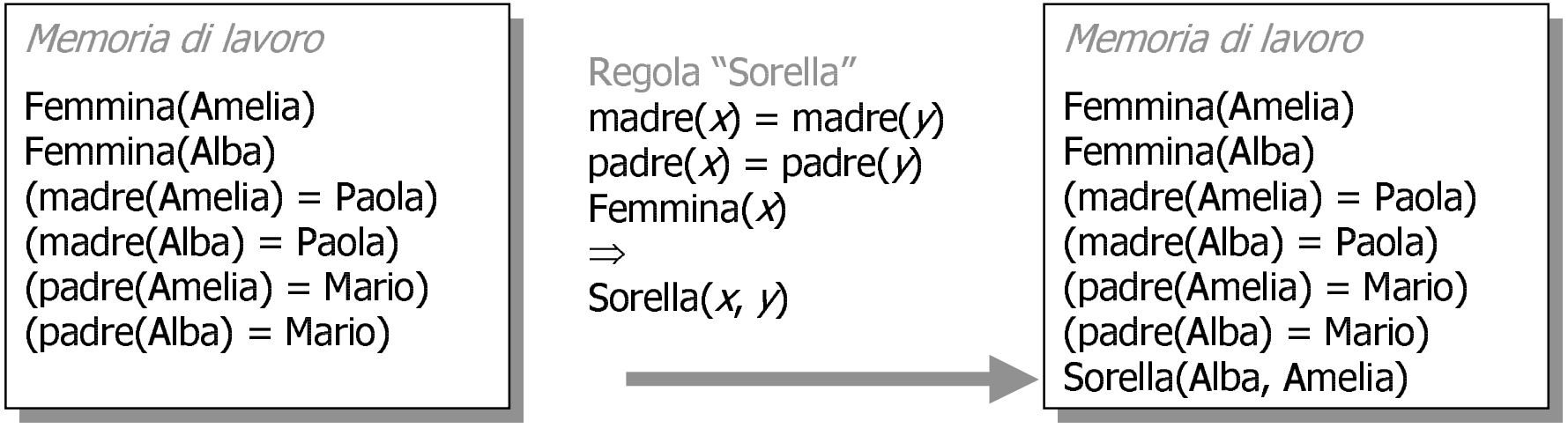
- Esempio:

Regola "Fratello"
madre(x) = madre(y)
padre(x) = padre(y)
Maschio(x)
$\Rightarrow$
Fratello(x, y)

Regola "Sorella"
madre(x) = madre(y)
padre(x) = padre(y)
Femmina(x)
$\Rightarrow$
Sorella(x, y)

Memoria di lavoro

- Un sistema a regole include anche una *memoria di lavoro* (anche *working memory* o WM)
  - la *memoria di lavoro* contiene la rappresentazione dei *fatti specifici*
  - le *regole* operano sulla *memoria di lavoro*
  - le *condizioni* sono istanziate sulla base dei *fatti specifici*
  - le *azioni* tipicamente comportano l'asserzione o la ritrattazione di *fatti specifici* (ma non solo)



Agenda, attivazione

- In ogni istante, un sistema a regole mantiene un *agenda* che contiene le *regole istanziate*
- Il sistema sceglie le *regole istanziate* da attivare
- L'*attivazione* (*firing*) delle regole avviene in modo sequenziale

Memoria di lavoro

Femmina(Amelia)
Femmina(Alba)
(madre(Amelia) = Paola)
(madre(Alba) = Paola)
(padre(Amelia) = Mario)
(padre(Alba) = Mario)

Agenda

Regola "Sorella" (istanza 1)

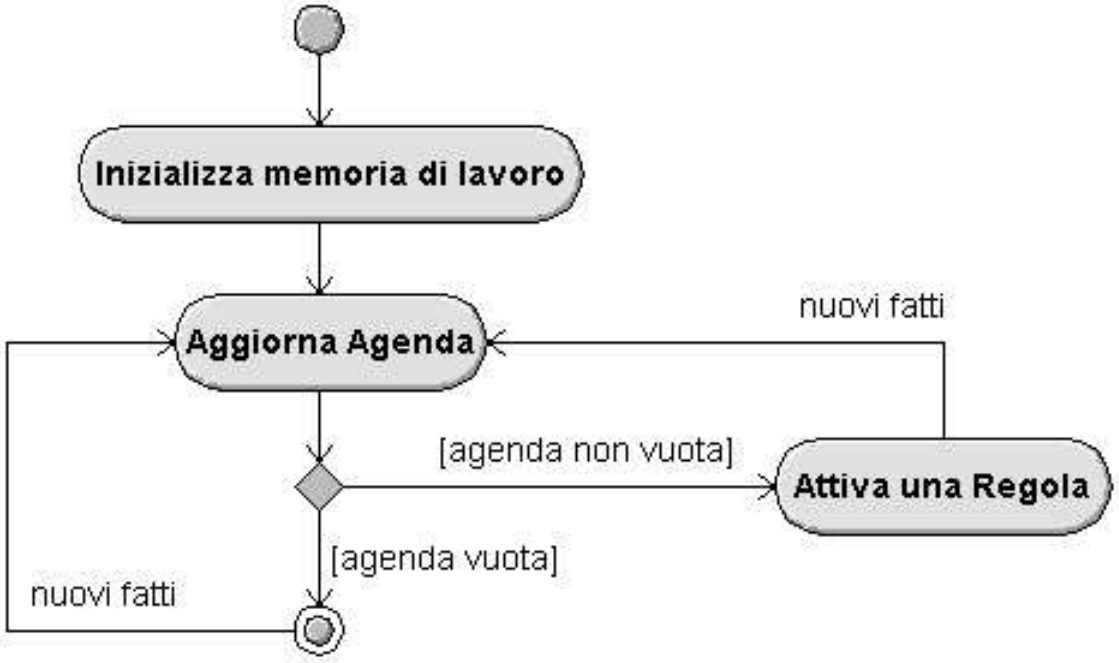
madre(Amelia) = madre(Alba)
padre(Amelia) = padre(Alba)
Femmina(Amelia)
⇒
Sorella(Amelia, Alba)

Regola "Sorella" (istanza 2)

madre(Alba) = madre(Amelia)
padre(Alba) = padre(Amelia)
Femmina(Alba)
⇒
Sorella(Alba, Amelia)

Ciclo di esecuzione

- Il sistema a regole procede ciclicamente:
  - aggiorna l'agenda
  - sceglie ed attiva una regola
  - aggiorna la memoria di lavoro

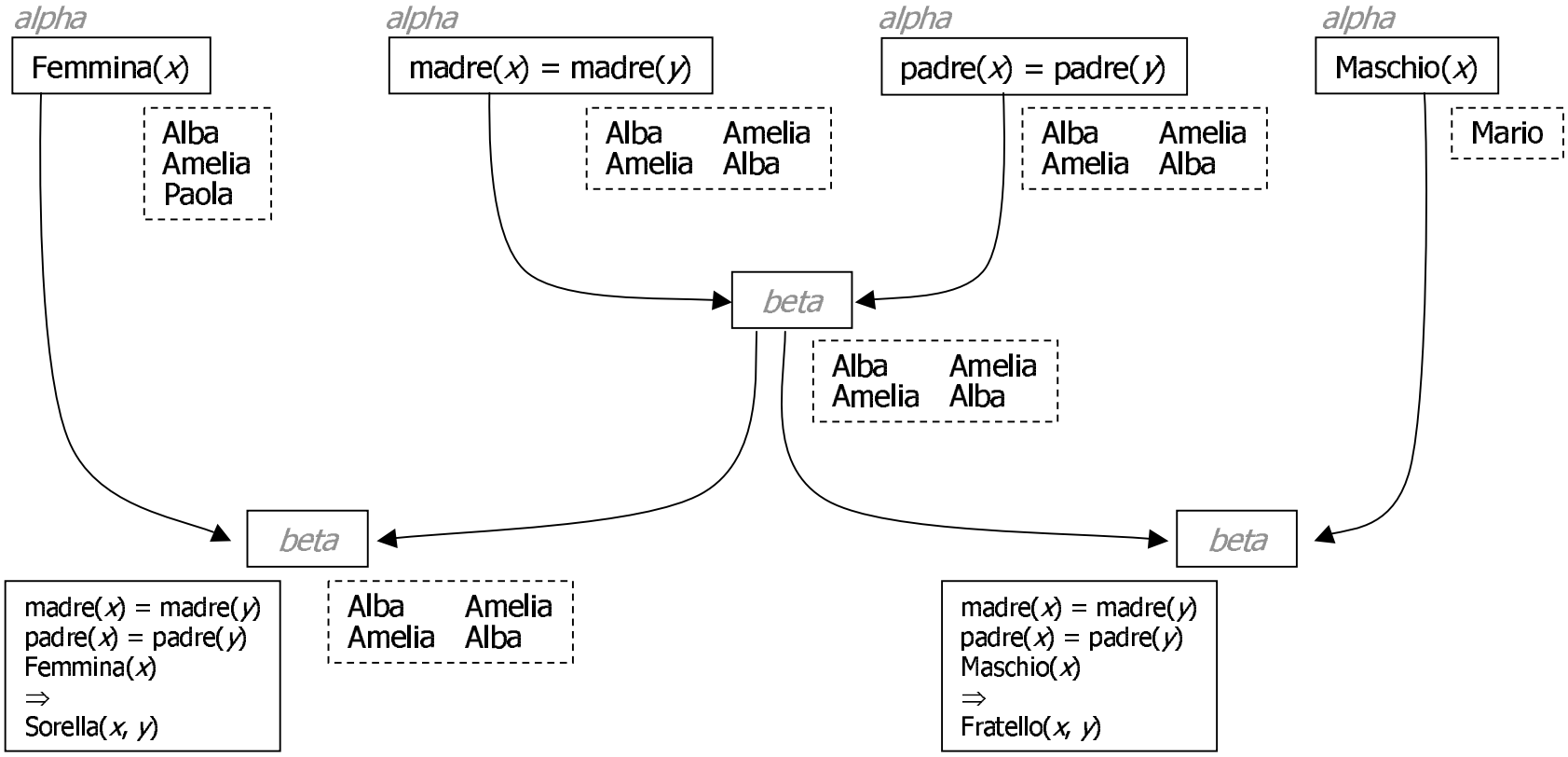


Sistema a regole - come funzionano?

- Il punto critico è l'aggiornamento dell'*agenda*
  - occorre identificare tutte le *istanziazioni* delle regole
  - evitando i cicli infiniti
    - le regole vanno inserite nell'agenda solo in presenza di *fatti nuovi*
    - diversamente, la loro attivazione è inutile
    - si veda l'esempio precedente
- Il confronto diretto è dispendioso
  - sarebbe necessario confrontare tutte le *regole* con tutti i *fatti*
    - distinguendo i fatti nuovi da quelli già noti

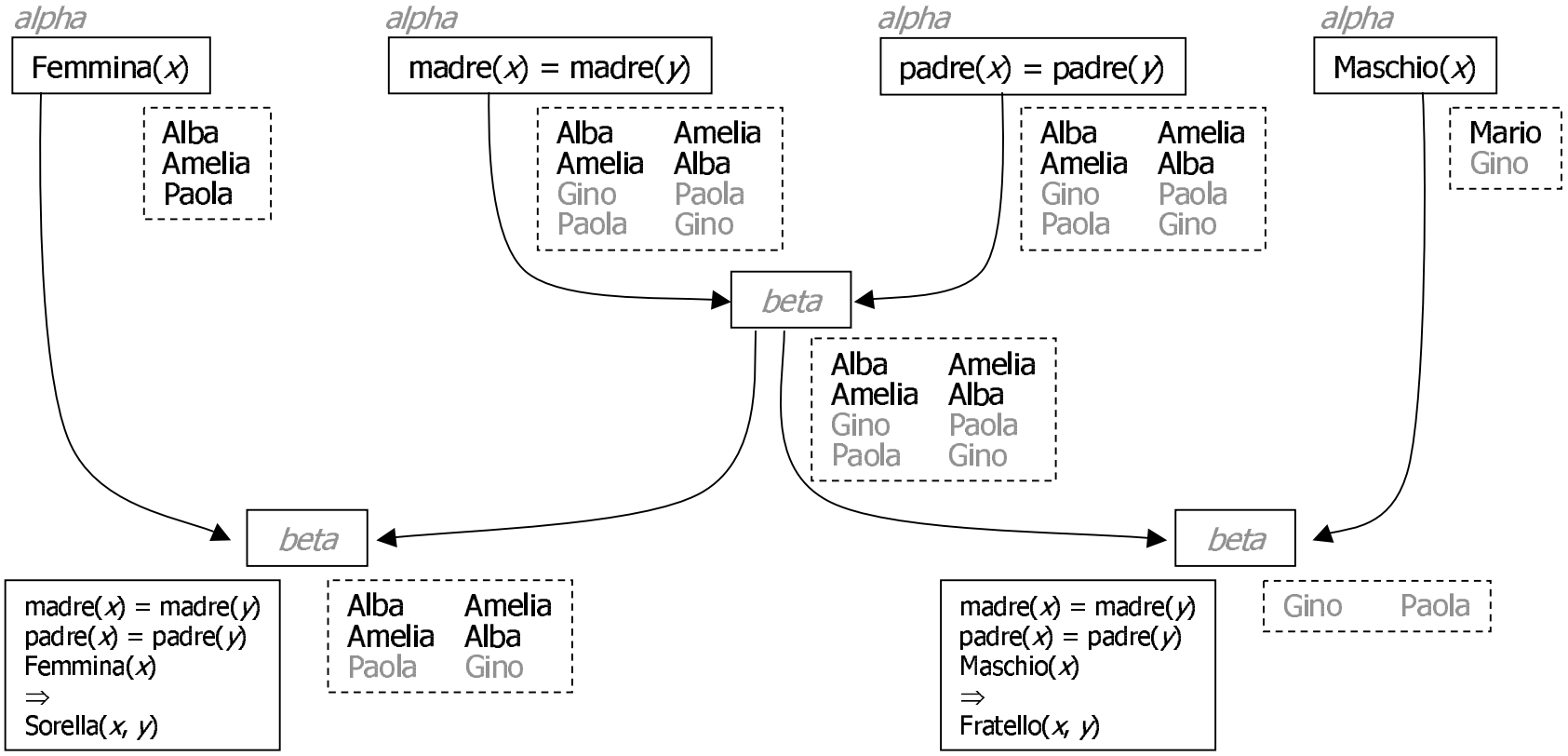
Algoritmo Rete (C. Forgy, 1980)

- Le *condizioni* di un *insieme di regole* vengono rappresentate in forma di *grafo aciclico*
  - a cui viene 'agganciata' la rappresentazione della *memoria di lavoro*



Aggiornamento della Rete

- I nuovi fatti vengono 'agganciati' ai nodi di pertinenza
  - "Gino ha gli stessi genitori di Paola"



2

Sistema Jess
(*Java Expert System Shell*)

Jess - Introduzione

- Una implementazione dell'algoritmo Rete
- incorporato in un ambiente di esecuzione Lisp (minimale)
- il tutto implementato in Java

- Realizzato da
  Ernest J. Friedman-Hill
  Distributed Computing Systems
  Sandia National Laboratories
  Livermore, CA - USA
- Disponibile presso:
  http://herzberg.ca.sandia.gov/jess
- Licenza gratuita (incluso il sorgente) per usi non-commerciali

Jess - Sintassi

- I fatti vengono espressi come tuple

```
(maschio Mario)
(femmina Paola)
(padre Alba Mario)
(padre Amelia Mario)
(madre Alba Paola)
(madre Amelia Paola)
```

- Le regole vengono espresse con una sintassi particolare

```
(defrule sorella
   (padre ?x ?p)
   (padre ?y ?p)
   (madre ?x ?m)
   (madre ?y ?m)
   (femmina ?x)
  =>
   (assert (sorella ?x ?y)))
```

Jess - Funzioni Lisp

- Il sistema a regole Jess è incorporato in un ambiente Lisp
- Di conseguenza le condizioni e le azioni possono includere chiamate a funzioni Lisp

- Due funzioni molto utili
  (tipicamente per definire le *azioni* in una regola)
  - **`(assert`** <fact>**`)`**
    inserimento di <fact> nella *memoria di lavoro*
    (com aggiornamento della struttura Rete)
  - **`(retract`** <fact>**`)`**
    rimozione di <fact> dalla *memoria di lavoro*
    (con aggiornamento della struttura Rete)

Jess – Funzioni di attivazione

- Il sistema Jess si attiva da linea di comando

```
$ java jess.Main
Jess>
```

  - La prima linea attiva il sistema,
    nella seconda compare il prompt del *Lisp Listener*
- Tipicamente le regole sono scritte su un file
  ma possono anche essere inserite direttamente

```
Jess> (batch "regole.clp")
```

  - Caricamento del file `regole.clp`
- Attivazione

```
Jess> (reset)
Jess> (run)
```

  - inizializzazione della memoria di lavoro
    con azzeramento dei fatti
  - attivazione del ciclo principale
    il sistema rimane attivo finchè vi sono regole da eseguire

Jess – Funzioni utili

- Ispezione

  ```
  Jess> (agenda)
  ```

  – mostra le regole istanziate presenti nell'agenda

  ```
  Jess> (facts)
  ```

  – mostra i fatti memorizzati nella memoria di lavoro
- Tracciamento

  ```
  Jess> (watch all)
  ```

  – tracciamento dell'esecuzione

  ```
  Jess> (unwatch all)
  ```

  – elimina il tracciamento dell'esecuzione

  ```
  Jess> (run 1)
  ```

  – attivazione di una sola regola
- Azzeramento

  ```
  Jess> (clear)
  ```

  – azzera regole e fatti

3

Fox, Goat & Cabbage
(*esercitazione con Jess*)

Un dilemma

- Partecipanti:
  - un agricoltore (*farmer*)
  - una volpe (*fox*)
  - una capra (*goat*)
  - un cavolo (*cabbage*)
- Scenario:
  - una barca
  - due rive (*shore-1*, *shore-2*)
- Obiettivo
  - tutti i partecipanti sono su una riva (*shore-1*)
  - l'agricoltore deve traghettare tutti sulla riva opposta (*shore-2*)
- Vincoli:
  - se lasciate sola con la capra, la volpe mangia la capra
  - se lasciata sola con il cavolo, la capra mangia il cavolo

Funzioni Lisp particolari

- Definizione contenuto iniziale della *memoria di lavoro*
  - **(deffact** <fact>**)**
    inserimento iniziale di <fact> nella *memoria di lavoro*
    (e aggiornamento della struttura Rete)
    al momento della esecuzione di **(reset)**
- Uso di templates per strutture dati composite
  - **(deftemplate** <structure>**)**
  - Esempio di definizione:

```
(deftemplate status
    (slot farmer-location)
    (slot fox-location)
    (slot goat-location)
    (slot cabbage-location))
```

  - Esempio d'uso (un *fatto specifico*):

```
(status (farmer-location shore-1)
        (fox-location shore-1)
        (goat-location shore-1)
        (cabbage-location shore-1))
```

Priorità tra regole

- Priorità tra regole regole (*salience*)
  - salvo diversa indicazione, ogni regola ha *salience* 0
  - indicazione esplicita della *salience:*

```
(defrule fox-eats-goat
   (declare (salience 100))
   ...
```

- La priorità tra regole è relativa
  (il valore assoluto della *salience* non conta)
- La priorità influenza la gestione dell'*agenda*
  - nella scelta per l'attivazione
  - le regole a priorità più alta vengono privilegiate
  - rispetto alle regole a priorità più bassa

Condizioni speciali

- Nella soluzione vengono usate alcune condizioni speciali
  - Identità (già vista nell'esempio precedente)

```
(defrule farmer-with-goat-and-fox
    (farmer-location ?x)
    (fox-location ?x)
    (goat-location ?x)
    ...
```

  - Differenza

```
(defrule fox-eats-goat
    (farmer-location ?x)
    (fox-location ?y&~?x)
    (goat-location ?y)
    ...
```

  - Confronto

```
(defrule y-larger-than-x
    (value ?x)
    (value ?y&:(< ?x ?y))
    ...
```

Azioni speciali

- Nella soluzione vengono usate alcune azioni speciali
  - Assegnazione di valori a variabili

```
(bind ?x (+ ?x 1))
```

  - Modifica di *fatti strutturati*

```
(modify ?fact
    (farmer-location ?x)
    (fox-location ?y))
```

  - Duplicazione e modifica di *fatti strutturati*

```
(duplicate ?fact
    (farmer-location ?x)
    (fox-location ?y))
```

Attivazione dell'esempio

- Impostazione della variabile **`CLASSPATH`**
  - meglio se in **`.login`**
- Copia del file delle regole
  - Copiare **`examples/dilemma.clp`**
- Attivazione del sistema Jess

```
$ java jess.Main
Jess>
```

- Caricamento del file **`dilemma.clp`**

```
Jess> (batch "dilemma.clp")
```

- Attivazione dell'esempio

```
Jess> (reset)
Jess> (run)
```

Domande

a) Qual'è l'algoritmo utilizzato per risolvere il dilemma?
   1) fornire una spiegazione informale
   2) spiegare il significato della struttura `status`

b) Qual'è il ruolo della priorità tra regole?
   1) provare a togliere le indicazioni di *salience*
   2) spiegare la differenza di comportamento

- Trascurare invece le regole di presentazione del risultato