INTELLIGENZA ARTIFICIALE E LABORATORIO

Planning e Sistemi a Regole

Roberto Micalizio

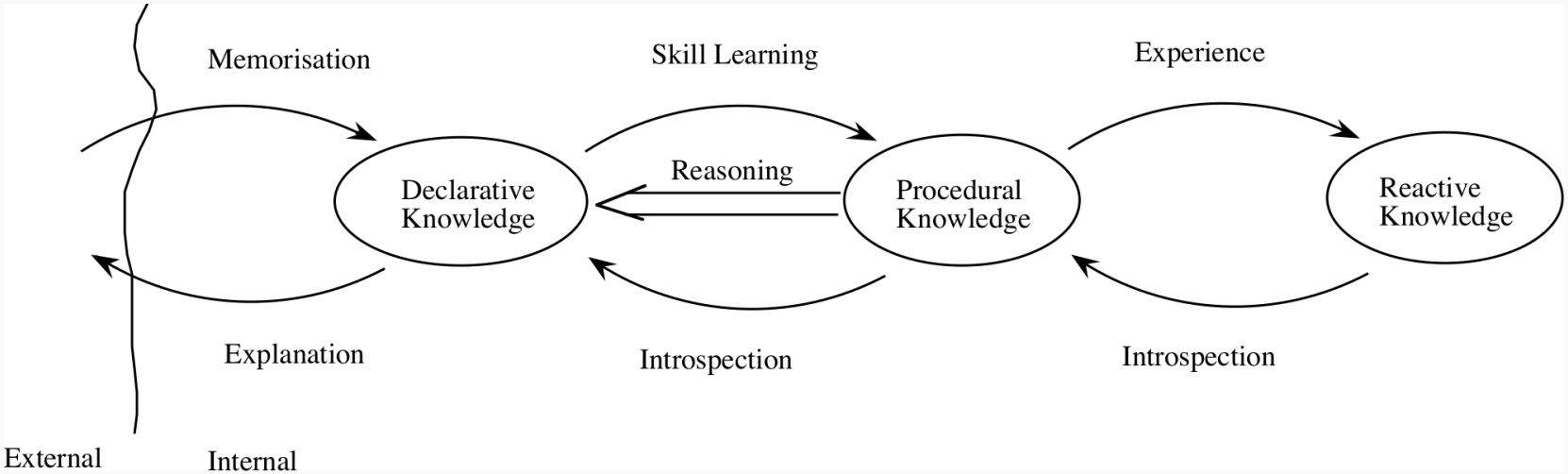
a.a. 2018/2019

Università degli Studi di Torino,
Dipartimento di Informatica

KNOWLEDGE, REASONING, UNDERSTANING

Knowledge

- **Knowledge** $\equiv$ Competenza: qualsiasi cosa che consenta di risolvere problemi
- Diverse forme di conoscenza secondo i cognitivisti



  - *Dichiarativa*: è astratta, è usata per comunicare e ragionare, deve essere interpretata per poter essere usata
  - *Procedurale*: una sequenza di passi per risolvere un problema
  - *Reattiva*: rsposta automatica ad uno stimolo, riflesso condizionato

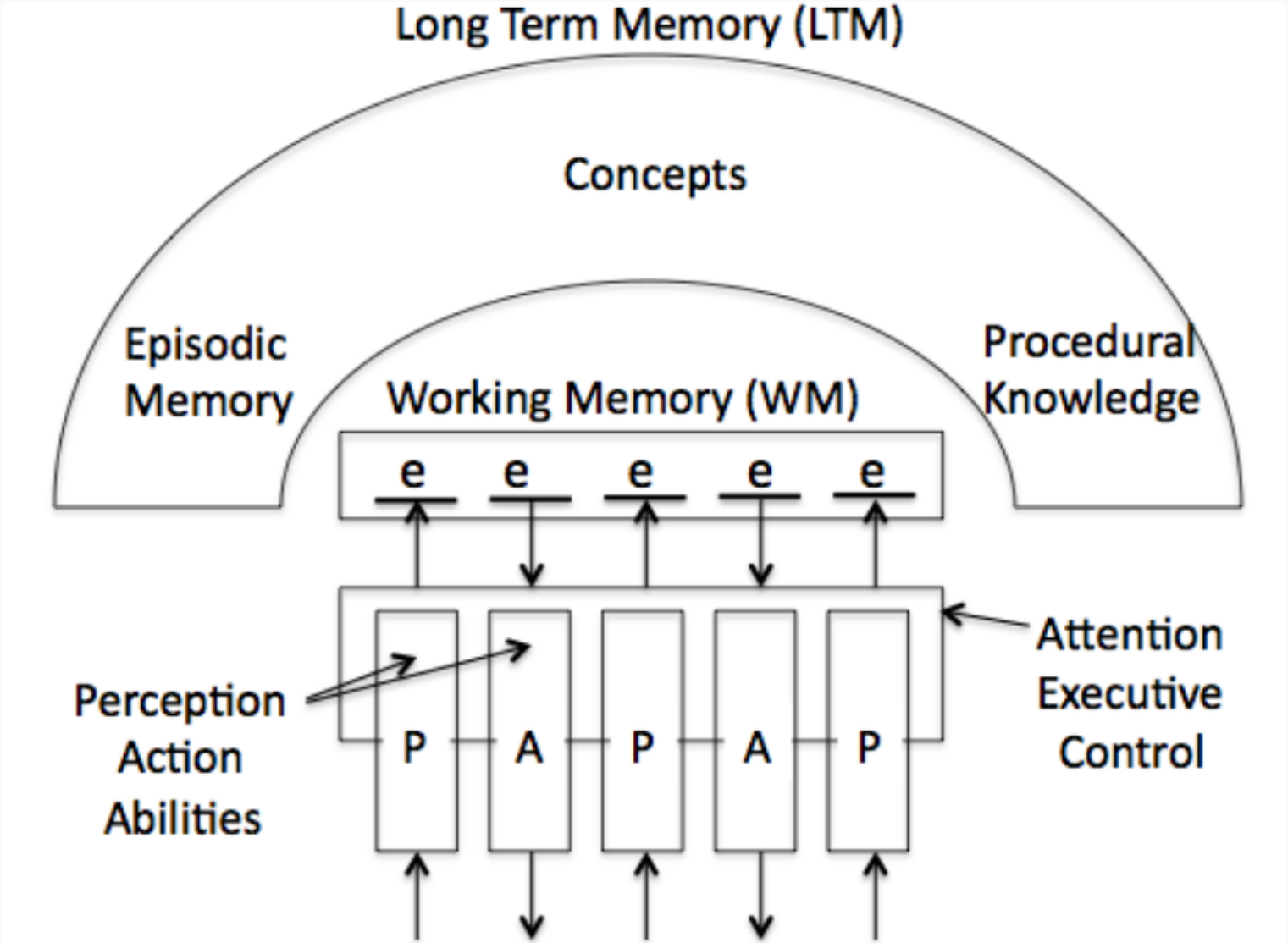
Reasoning

- **Reasoning** è la capacità di attribuire un significato ai fenomeni che venogno percepiti o ricordati.
- Reasoning può essere applicato a diversi tipi di memoria (percettivi, episodica, procedurale, o concettuale)
- Reasoning consente di associare ad un fenomeno una qualche forma di conoscenza rendendo possibile di spiegare, predire o agire... $\Rightarrow$ Reasoning porta alla comprensione (Understanding)

Understanding

- **Understanding** è l'abilità di predire e spiegare
- Tipicamente si affida ad un qualche modello che può essere usato per predire il risultato di un processo o fenomeno.

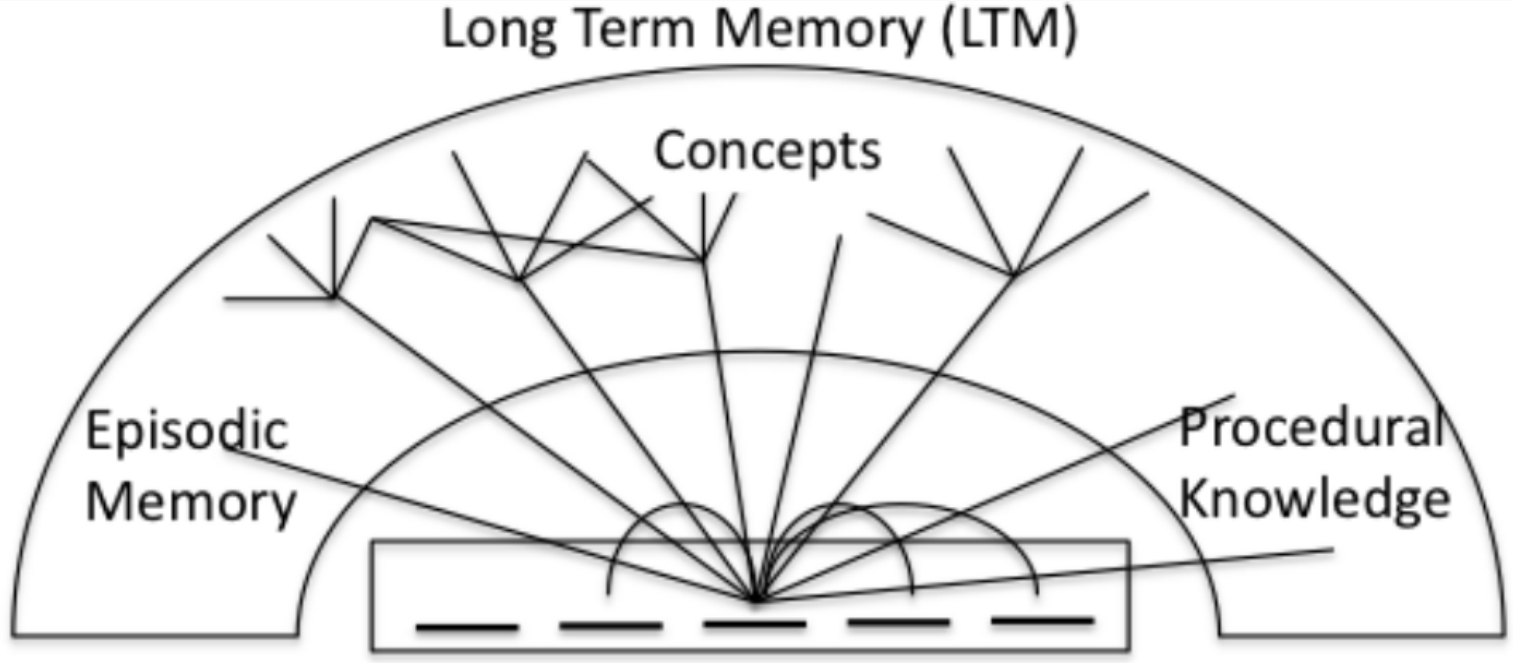
Un possibile modello cognitivo



Un possible modello cognitivo

- **Perception**: trasforma e combina gli stimoli sensoriali in Fenomeni
- **Short Term Perceptual Memory**: memoria temporanea che contiene gli stimoli recenti
- **Action**: attivazione di gruppi di muscoli
- **Working Memory**: 7 +/- 2 elementi (ricordati o percepiti)
- **Long Term Memory**:
  - *Episodic Memory*: Memorie di esperienze sensoriali significative
  - *Semantic Memory*: Rappresentazione astratta di esperienze sensoriali
  - *Procedural Memory*: Sequenze di operazioni per conseguire un risultato
  - *Spatial Memory*: reti di luoghi

Attivazione Distribuita



- L'energia si propaga dalla memoria a breve termine agli altri elementi della memoria a breve e lungo termine
- le parti attivate propagano l'energia ad altre parti ancora fino a rimpiazzare uno degli elementi nella memoria di lavoro
- l'energia decade col tempo

SISTEMI ESPERTI

Sistemi Esperti

Detti anche *Knowledge-Based Systems*

> *"A computer system that emulates the decision-making ability of a human expert in a restricted domain."*
>
> [Giarratano & Riley 1998]

> *"An intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solutions."*
>
> [Giarratano & Riley 1998]

Sistemi Esperti

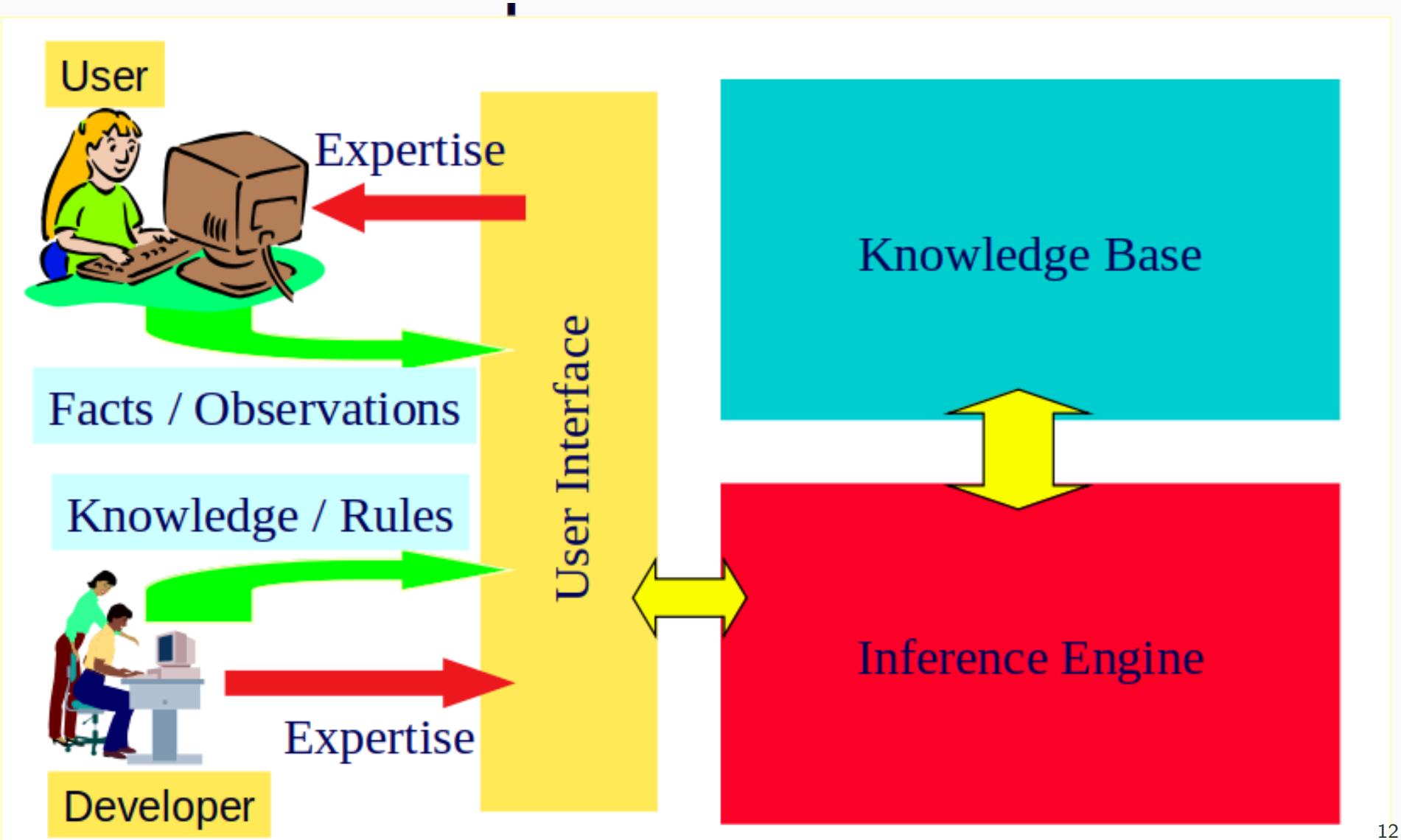
- Un sistema basato sulla conoscenza è un sistema in grado di risolvere problemi in un dominio limitato ma con prestazioni simili a quelle di un esperto umano del dominio stesso.
- Generalmente esamina un largo numero di possibilità e costruisce dinamicamente una soluzione.
- *"La potenza di un programma intelligente nel risolvere un problema dipende primariamente dalla quantità e qualità di conoscenza che possiede su tale problema"*. (Feigenbaum)

Sistemi Esperti

- Il programma non è un insieme di istruzioni immutabili che rappresentano la soluzione del problema, ma un ambiente in cui:
  - rappresentare;
  - utilizzare (*reasoning*);
  - modificare

  una base di conoscenza.
- Caratterizzato dalle seguenti proprietà:
  - Specificità (conoscenza di dominio)
  - Rappresentazione esplicita della conoscenza.
  - Meccanismi di ragionamento.
  - Capacità di spiegazione.
  - Capacità di operare in domini poco strutturati

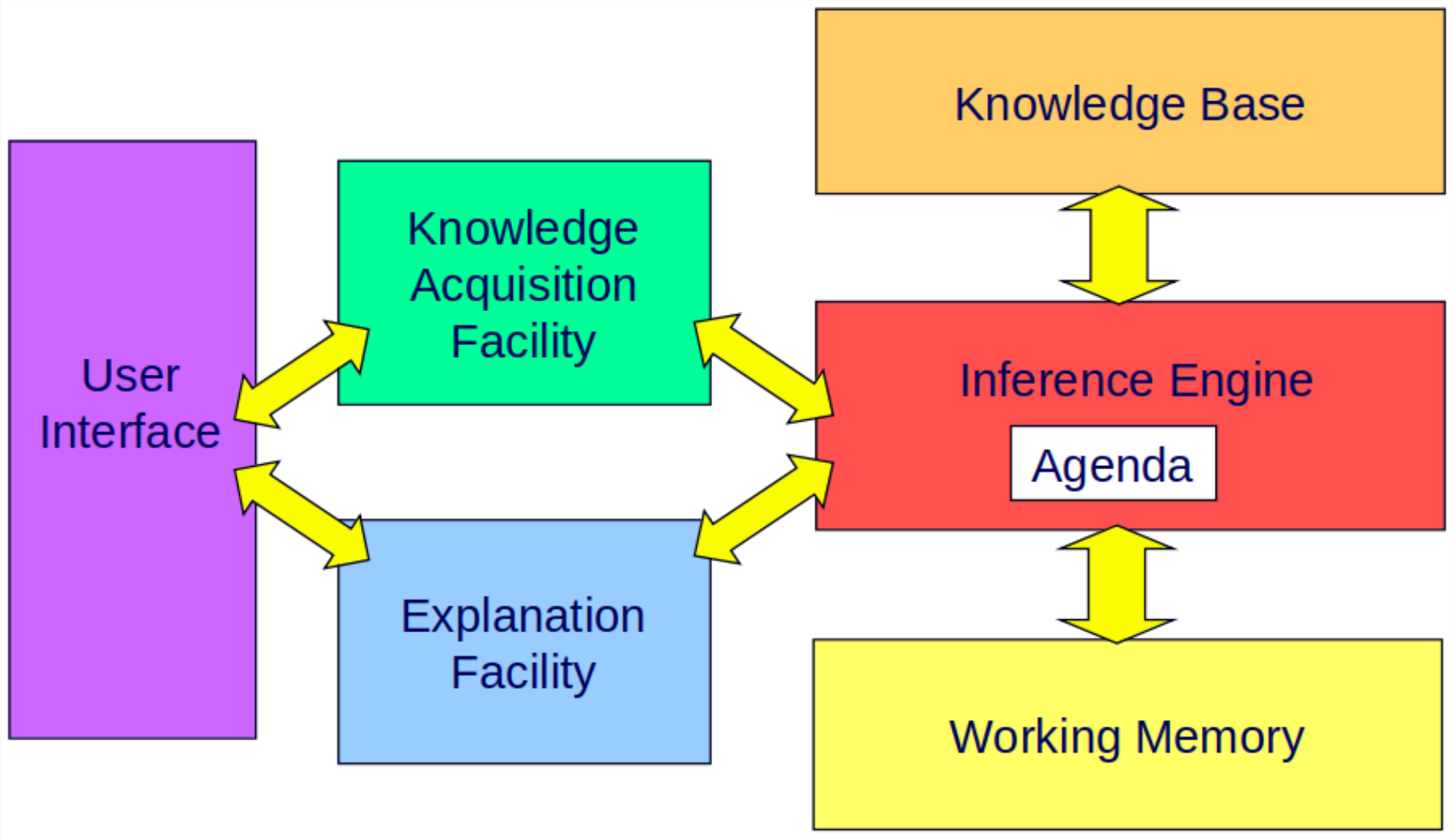
Cos'è un Sistema Esperto



Un sistema ispirato ai modelli cognitivi

- Un sistema esperto mima il modello cognitivo umano:
  - le regole sono memorizzate nella memoria a lungo termine
  - conoscenza temporanea è mantenuta nella memoria a breve termine
  - stimoli sensoriali esterni attivano le regole (comportamenti automatici/riflessi)
  - le regole attivate possono a loro volta attivarne di nuoce (input interno; "pensiero/ragionamento")

Architettura di un Sistema esperto



Architettura di un Sistema esperto

- **Knowledge-Base** [KB] (Rule-Base): mantiene la conoscenza dell'esperto come regole `condizione-azione` (aka: `if-then` or `premise-consequence`)
- **Working Memory** [WM]: mantiene i fatti iniziali e quelli generati dalle inferenze
- **Inference Engine**:
  - *pattern matching*: confronta la parte `if` delle regole rispetto ai fatti della WM
  - regole che hanno la parte `if` soddisfatta sono dette **attivabili** e sono poste nell'Agenda
  - **Agenda**: elenco ordinato di regole attivabili, le regole sono ordinate in base alla loro priorità o in base ad altre strategie di preferenza/conflict resolution
  - **Execution**: la regola in cima all'agenda è selezionata ed eseguita (*firing*)
- **Explanation Facility**: fornisce una giustificazione delle soluzioni (*reasoning chain* $\equiv$ sequenza di regole che sono state attivate)
- **Knowledge Acquisition Facility**: aiuta l'ingegnere della conoscenza ad integrare nuove regole e a mantenerle nel tempo
- **User Interface**: consente all'utente di interagire con il sistema esperto (porre problemi e ricavarne una risposta)

Esempio di Regole di Produzione

```
R1:  IF <la temperatura è maggiore di 100 gradi>,
     AND <la pressione ha valore p1>
     THEN <attiva la valvola .....>
     AND <segnala una situazione di allarme all'operatore>
--- priorità 10

R2:  IF <la pressione ha valore p1>
     THEN <attiva la procedura di funzionamento normale PROC5>
--- priorità 5
```

Due direzioni di ragionamento

- **forward chaining** (**data-driven**)
  - ragionamento da fatti a conclusioni
  - non appena fatti (osservazioni) sono disponibili, questi sono usati per il matching degli antecedenti delle regole
  - spesso usati per real-time monitoring e controllo
  - CLIPS, OPS5
  - le regole possono essere causali o meno:
    - $\langle \text{disease} \rangle \rightarrow \langle \text{symptom} \rangle$ (predict, prescribe)
    - $\langle \text{symptom} \rangle \rightarrow \langle \text{disease} \rangle$ (explain, diagnose)

Due direzioni di ragionamento

- **forward chaining** (**data-driven**)
  - ragionamento da fatti a conclusioni
  - non appena fatti (osservazioni) sono disponibili, questi sono usati per il matching degli antecedenti delle regole
  - spesso usati per real-time monitoring e controllo
  - CLIPS, OPS5
  - le regole possono essere causali o meno:
    - $\langle \text{disease} \rangle \rightarrow \langle \text{symptom} \rangle$ (predict, prescribe)
    - $\langle \text{symptom} \rangle \rightarrow \langle \text{disease} \rangle$ (explain, diagnose)
- **backward chaining** (**goal-driven**)
  - partendo da un'ipotesi (goal), regole e fatti a supporto dell'ipotesi sono ricercati fino a quando tutte le parti dell'ipotesi non sono dimostrate
  - usata in sistemi diagnostici
  - MYCIN
  - le regole possono essere causali o meno:
    - $\langle \text{disease} \rangle \rightarrow \langle \text{symptom} \rangle$ (explain, diagnose)
    - $\langle \text{symptom} \rangle \rightarrow \langle \text{disease} \rangle$ (predict, prescribe)

Conoscenza di dominio e di controllo

- Ogni sistema basato sulla conoscenza deve riuscire ad esprimere due tipi di conoscenza in modo separato e modulare:
  - Conoscenza sul dominio dell'applicazione (**COSA**);
  - Conoscenza su **COME** utilizzare la conoscenza sul dominio per risolvere problemi (**CONTROLLO**).
- Problemi:
  - Come esprimere la conoscenza sul problema?
  - Quale strategia di controllo utilizzare?

Domini applicativi

- **Interpretazione**: Si analizzano dati complessi e potenzialmente affetti da rumore per la determinazione del loro significato (Dendral, Hearsay-II).
- **Diagnosi**: Si analizzano dati (con potenziale rumore) per la determinazione di malattie o errori (Mycin, ...).
- **Monitoring**: I dati si interpretano continuamente per la generazione di allarmi in situazioni critiche. Al sistema è richiesta una risposta in tempo reale soddisfacente (VM).
- **Planning e Scheduling**: Si determina una sequenza intelligente di azioni per raggiungere un determinato obiettivo (Molgen).
- **Previsione** (e.g., economica): Si desidera costruire un sistema in grado di prevedere il futuro in base a un appropriato modello del passato e del presente (Prospector).
- **Progetto e configurazione**: Il Sistema Esperto deve essere in grado di progettare sistemi partendo da ben determinate specifiche (R1/XCON).

Quando non usare un sistema esperto

- Sistemi esperti non sono da preferire quando:
  - esistono algoritmi "tradizionali" efficienti per risolvere il problema considerato
  - l'aspetto principale del problema è la computazione, e non la conoscenza
  - la conoscenza non può essere modellata o elicitata in modo efficiente
  - l'utente finale è riluttante ad applicare un sistema esperto per via della criticità del task (e.g., somministrare medicamenti, pilotare aerei,...)
- Posso risolvere il problema di pianificazione con un sistema esperto?

Quando non usare un sistema esperto

- Sistemi esperti non sono da preferire quando:
  - esistono algoritmi "tradizionali" efficienti per risolvere il problema considerato
  - l'aspetto principale del problema è la computazione, e non la conoscenza
  - la conoscenza non può essere modellata o elicitata in modo efficiente
  - l'utente finale è riluttante ad applicare un sistema esperto per via della criticità del task (e.g., somministrare medicamenti, pilotare aerei,...)
- Posso risolvere il problema di pianificazione con un sistema esperto?

  Sì! (Ma con molta fatica)

  Un sistema a regole di produzione ha la stessa potenza espressiva di una macchina di Turing, quindi può risolvere qualsiasi problema risolvibile con un algoritmo tradizionale, ma non sempre è il formalismo più adeguato.

CLIPS

CLIPS Primitive Data Types

- float
- integer
- **symbol**: e.g. this-is-a-symbol, wrzlbrmft, !?@*+
- string: e.g. "This is a string"
- external address (indirizzi o strutture dati restituite da funzioni user-defined)
- instance name (Cool)
- instance address (Cool)

Avviare/Terminare CLIPS

- Avviare CLIPS da riga di comando (UNIX):

```
$ ./clips
```

Avviare/Terminare CLIPS

- Avviare CLIPS da riga di comando (UNIX):
  `$ ./clips`
  appare il prompt
  `CLIPS>`

Avviare/Terminare CLIPS

- Avviare CLIPS da riga di comando (UNIX):
  `$ ./clips`
  appare il prompt
  `CLIPS>`
- Terminare CLIPS
  `CLIPS> (exit)`

Avviare/Terminare CLIPS

- Avviare CLIPS da riga di comando (UNIX):
  `$ ./clips`
  appare il prompt
  `CLIPS>`
- Terminare CLIPS
  `CLIPS> (exit)`

NB. CLIPS è un interprete di regole, quindi possiamo modificare la WM e la KB in qualsiasi momento aggiungendo/rimuovendo fatti e regole.

Fatti

- "chunk of information" elemento indivisibile di informazione
- sono definiti da un nome di relazione
- possono avere zero o più *slot* (i.e., *facet*)
  (se non hanno slot il nome della relazione li identifica univocamente)
- costrutto `deftemplate` per definire fatti strutturati (detti non ordinati)
- costrutto `deffacts` per definire il set iniziale di fatti

Fatti - esempi

- **fatto ordinato** (no slot)

```
(person-name Franco L Verdi)
(person-name Verdi Franco L)
```

- **fatto non ordinato** (esempio di **template**)

```
(deftemplate person "commento opzionale"
   (slot name)
   (slot age)
   (slot eye-color)
   (slot hair-color))
```

- **fatto non ordinato** (esempio di **istanza** di template)

```
(person (name "Franco L. Verdi")
    (age 46)
    (eye-color brown)
    (hair-color brown))
```

Modellare la conoscenza con i templates

```
(deftemplate student "a student record"
   (slot name (type STRING))
   (slot age (type INTEGER) (default 18) (range 0 ?VARIABLE) )
   (slot gender (type SYMBOL) (allowed-symbols male female))
)

(deftemplate volleyball-team
   (slot name (type STRING))
   (multislot player (type STRING) (cardinality 6 6))
   (multislot alternates (type STRING) (cardinality 0 2)) )
```

Modellare la conoscenza con i templates

```
(deftemplate student "a student record"
   (slot name (type STRING))
   (slot age (type INTEGER) (default 18) (range 0 ?VARIABLE) )
   (slot gender (type SYMBOL) (allowed-symbols male female))
)

(deftemplate volleyball-team
   (slot name (type STRING))
   (multislot player (type STRING) (cardinality 6 6))
   (multislot alternates (type STRING) (cardinality 0 2)) )
```

?VARIABLE indica che non c'è un limite superiore specificato

Modellare la conoscenza con i templates

```
(deftemplate student "a student record"
   (slot name (type STRING))
   (slot age (type INTEGER) (default 18) (range 0 ?VARIABLE) )
   (slot gender (type SYMBOL) (allowed-symbols male female))
)

(deftemplate volleyball-team
   (slot name (type STRING))
   (multislot player (type STRING) (cardinality 6 6))
   (multislot alternates (type STRING) (cardinality 0 2)) )
```

male e female sono gli unici valori ammessi

Modellare la conoscenza con i templates

```
(deftemplate student "a student record"
   (slot name (type STRING))
   (slot age (type INTEGER) (default 18) (range 0 ?VARIABLE) )
   (slot gender (type SYMBOL) (allowed-symbols male female))
)

(deftemplate volleyball-team
   (slot name (type STRING))
   (multislot player (type STRING) (cardinality 6 6))
   (multislot alternates (type STRING) (cardinality 0 2)) )
```

cardinalità minima e massima del multislot: quanti valori deve almeno e può al più contenere

Esempio di `deffacts`

il costrutto `deffacts` è usato unicamente per stabilire quali fatti sono veri all'avvio del motore inferenziale

```
(deffacts famiglia-verdi "alcuni membri della famiglia Verdi"
    (person (name "Luigi") (age 46) (eye-color brown) ( hair-color brown))
    (person (name "Maria") (age 40) (eye-color blue) ( hair-color brown))
    (person (name "Marco") (age 25) (eye-color brown) ( hair-color brown))
    (person (name "Lisa") (age 20) (eye-color blue) ( hair-color blonde))
)
```

- I fatti sono effettivamente asseriti in WM solo dopo aver dato il comando `(reset)`
- Per elencare i fatti presenti in memoria si usa `(facts)`

Manipolazione di fatti

- aggiungere fatti
  `(assert <fact>+)`
- rimuovere fatti
  `(retract <fact-index>+)`
- modificare fatti
  `(modify <fact-index> (<slot-name> <slot-value>)+ )`
  equivale a rimuovere il fatto originale e aggiungere il nuovo fatto modificato con un indice incrementato
- duplicare fatti
  `(duplicate <fact-index> (<slot-name> <slot-value>)+ )`
- ispezionare la working memory
  `(facts)` stampa la lista di fatti
  `(watch facts)` automaticamente mostra i cambiamenti che occorrono nella WM a seguito dell'esecuzione delle regole
- `(unwatch facts)` per disabilitare il watching sui fatti

Regole

```
(defrule <rule name> ["comment"]
   <patterns>* ; left-hand side (LHS)
               ; or antecedent of the rule
=>
   <actions>*) ; right-hand side (RHS)
               ; or consequent of the rule
```

Regole - esempio

```
(defrule birthday-FLV
   (person (name "Luigi")
      (age 46)
      (eye-color brown)
      (hair-color brown))
      (date-today April-13-02)
=>
      (printout t "Happy birthday, Luigi!")
      (modify 1 (age 47))
)
```

NB. Questo è un caso molto particolare di regola perché i pattern dell'antecedente sono completamente istanziati, in generale si ricorre all'uso di **variabili**

Regole - esempio

```
(defrule birthday-FLV
   (person (name "Luigi")
      (age 46)
      (eye-color brown)
      (hair-color brown))
      (date-today April-13-02)
=>
      (printout t "Happy birthday, Luigi!")
      (modify 1 (age 47))
)
```

ATTENZIONE! questo non è il modo corretto per riferirsi ai fatti in WM, servono delle "maniglie", handlers fornite da variabili!

Regole - ispezionare e tracciare

- `(rules)` elenca le regole definite fino a questo momento nella KB
- `(ppdefrule` *`nome-regola`*`)` mostra la definizione della regola
- `(agenda)` mostra l'agenda attuale: elenco ordinato di regole attivabili
  - la regola in cima sarà la prossima ad essere eseguita
  - per ogni regola sono indicati i fatti che l'attivano
- `(watch rules)` mostra, durante l'esecuzione, quali regole sono eseguite
- `(watch activations)` mostra, durante l'esecuzione, quali attivazioni hanno permesso l'esecuzione delle regole
- `(unwatch rules)` e `(unwatch activations)` per disattivare il watching

Debugging... a sort of

- Per evitare di avere le informazioni di watching direttamente nell'interfaccia dell'interprete, si può usare il comando
  `(dribble-on` *`filename`*`)`
  che salva tutto ciò che viene visualizzato sul terminale sul file indicato, compreso gli input dell'utente.
  `(dribble-off)`
  Disabilita il comando.
- Breackpoints:
  `(set-break <rulename>)`
  `(remove-break <rulename>)`
  `(show-breaks)`

Variabili

- Nomi simbolici che cominciano con '?'
- **binding**:
  - variabili in un pattern (LHS) sono legate a valori di fatti in WM
  - ogni occorrenza di una variabile in una regola ha lo stesso valore
  - l'occorrenza più a sinistra (la prima) determina il valore
  - il binding è valido solo localmente ad una regola
  - variabili sono usate anche come maniglie (handlers) per accedere ai fatti della WM
    es. `?age <- (age harry 17)`
- wildcards ($\sim$ variabili senza nome)
  - ? match con qualsiasi valore in un campo singolo (slot) di un fatto
  - $? match con zero o più valori in un multislot di un fatto

Esempio Corretto

```
(defrule birthday-FLV
   ?person <- (person (name "Luigi")
      (age 46)
      (eye-color brown)
      (hair-color brown))
      (date-today April-13-02)
=>
      (printout t "Happy birthday, Luigi!")
      (modify ?person (age 47))
)
```

Altro Esempio

```
(defrule find-blue-eyes
   (person (name ?name)(eye-color blue))
=>
   (printout t ?name " has blue eyes." crlf))
```

La **regola scatta per tutte le persone che hanno gli occhi blu**. I valori degli slot name, age, ecc. non contano, ma con ?name siamo in grado di recuperare il nome della specifica persona che attiva la regola

Però non vorrei entrare in un loop infinito in cui la regola stampa all'infinito *"Lisa has blue eyes"*.

Altro Esempio

```
(defrule find-blue-eyes
   (person (name ?name)(eye-color blue))
=>
   (printout t ?name " has blue eyes." crlf))
```

La **regola scatta per tutte le persone che hanno gli occhi blu**. I valori degli slot name, age, ecc. non contano, ma con ?name siamo in grado di recuperare il nome della specifica persona che attiva la regola

Rifrazione

è un meccanismo alla base del patter-matching che impedisce di attivare due vole una regola sugli stessi fatti

La **regola scatta una sola volta per ogni persona con gli occhi blu**.

"Lisa has blue eyes" viene stampato una volta sola

Avviare il motore inferenziale di CLIPS

today.clp:

```
(defrule start
   (initial-fact)
=>
(printout t "hello"))
```

```
CLIPS> (load today.clp)
CLIPS> (facts)
CLIPS> (reset)
CLIPS> (facts)
f-0 (initial-fact)
```

Avviare il motore inferenziale di CLIPS

today.clp:

```
(defrule start
   (initial-fact)
=>
(printout t "hello"))
```

```
CLIPS> (load today.clp)
CLIPS> (facts)
CLIPS> (reset) (carica i fatti nei costrutti deffacts)
CLIPS> (facts)
f-0 (initial-fact)
```

Avviare il motore inferenziale di CLIPS

today.clp:

```
(defrule start
   (initial-fact)
=>
(printout t "hello"))
```

```
CLIPS> (load today.clp)
CLIPS> (facts)
CLIPS> (reset)
CLIPS> (facts)
```

`f-0 (initial-fact)` (è un fatto di default utilizzato dal motore)

Avviare il motore inferenziale di CLIPS

today.clp:

```
(defrule start
   (initial-fact)
=>
(printout t "hello"))
```

```
CLIPS> (load today.clp)
CLIPS> (facts)
CLIPS> (reset)
CLIPS> (facts)
f-0 (initial-fact)
  CLIPS> (run)
CLIPS>hello
```

Field Constraints

Vincoli definiti per il singolo campo usati per filtrare il pattern matching

- ~ **not**: il campo può prendere qualsiasi valore tranne quello specificato

```
(name ~Simpson)
(age ~40)
```

- | **or**: sono specificati valori alternativi ammissibili per uno stesso campo

```
(name Simpson|Oswald)
(age 30|40|50)
```

- & **and**: il valore del campo deve soddisfare una congiunzione di vincoli

```
(name ?name&~Simpson)
(name ?name&Harvey)
```

- : **expression**: aggiungi l'espressione che segue come vincolo

```
(age ?age&:(> ?age 20))
(name ?name&:(eq  (sub-string 1 1 ?name) "S")
```

Field Constraints - Esempio

```
(defrule silly-eye-hair-match
  (person (name ?name1)
      (eye-color ?eyes1 & blue|green)
      (hair-color ?hair1 & ~black))
  (person (name ?name2 & ~?name1)
      (eye-color ?eyes2 & ~?eyes1)
      (hair-color ?hair2 & red|?hair1))
=>
(printout t ?name1 " has "?eyes1 " eyes and "  ?hair1 " hair." crlf)
(printout t ?name2 " has "?eyes2 " eyes  and " ?hair2 " hair." crlf)
)
```

Field Constraints vs test

- In alternativa ai field constraint è possibile usare la clausola
  `(test` *`boolean-function`*`)`
- Ad esempio

```
(defrule silly-eye-hair-match
  (person (name ?name1)
          (eye-color ?eyes1)
          (hair-color ?hair1))
  (person (name ?name2)
          (eye-color ?eyes2)
          (hair-color ?hair2 & red|?hair1))
  (test (or (eq ?eyes1 blue) (eq ?eyes1 green)  ) )
  (test (neq ?hair1 black))
  (test (neq ?name1 ?name2))
  (test (neq ?eyes1 ?eyes2))
  (test (or (eq ?hair2 red) (eq ?hair2 ?hair1)  ))
=>
(printout t ?name1 " has "?eyes1 " eyes and "  ?hair1 " hair." crlf)
(printout t ?name2 " has "?eyes2 " eyes  and " ?hair2 " hair." crlf))
```

Field Constraints vs test

- In alternativa ai field constraint è possibile usare la clausola
  `(test` *`boolean-function`*`)`
- Ad esempio

```
(defrule silly-eye-hair-match
  (person (name ?name1)
          (eye-color ?eyes1)
          (hair-color ?hair1))
  (person (name ?name2)
          (eye-color ?eyes2)
          (hair-color ?hair2 & red|?hair1))
  (test (or (eq ?eyes1 blue) (eq ?eyes1 green)  ) )
  (test (neq ?hair1 black))
  (test (neq ?name1 ?name2))
  (test (neq ?eyes1 ?eyes2))
  (test (or (eq ?hair2 red) (eq ?hair2 ?hair1)  ))
=>
(printout t ?name1 " has "?eyes1 " eyes and "  ?hair1 " hair." crlf)
(printout t ?name2 " has "?eyes2 " eyes  and " ?hair2 " hair." crlf))
```

C'è un evidente vantaggio ad usare i filed constraint, quale?

ESERCITAZIONE

Usare CLIPS per rappresentare predicati e formule logiche

- Vogliamo scrivere una piccola KB che ci permetta di inferire le parentele tra un gruppo di persone, ad esempio vorremmo dire che
  - se x è un genitore di y, allora è anche un antenato di y

$$\forall x, y, \mathit{parent}(x, y) \rightarrow \mathit{ancestor}(x, y)$$

  - se x è un genitore di un antenato z di y, allora anche x è un antenato di y

$$\forall x, y, z, \mathit{parent}(x, z) \land \mathit{ancestor}(z, y) \rightarrow \mathit{ancestor}(x, y)$$

  - essere padre/madre vuol dire essere un genitore

$$\forall x, y, \mathit{father}(x, y) \rightarrow \mathit{parent}(x, y)$$

$$\forall x, y, \mathit{mother}(x, y) \rightarrow \mathit{parent}(x, y)$$

  - avere figli, essere umani e maschi vuol dire essere padri

$$\forall x, y, \mathit{hasChild}(x, y) \land \mathit{human}(x) \land \mathit{male}(x) \rightarrow \mathit{father}(x)$$

  - analogamente

$$\forall x, y, \mathit{hasChild}(x, y) \land \mathit{human}(x) \land \mathit{female}(x) \rightarrow \mathit{mother}(x)$$

Usare CLIPS per rappresentare formule logiche

- Supponiamo di essere a conoscenza dei seguenti fatti:

```
human(Luigi), male(Luigi)
human(Marta), female(Marta)
human(Luca), male(Luca),
human(Maria), female(Maria),
human(Lucrezia), female(Lucrezia),
human(Ludovico), male(Ludovico),
human(Miriam), female(Miriam),
hasChild(Luigi, Marta)
hasChild(Luigi, Luca)
hasChild(Marta, Maria)
hasChild(Marta, Lucrezia)
hasChild(Maria, Ludovico)
hasChild(Ludovico, Miriam)
```

Esercizio 1

- Scrivere un programma CLIPS che deduca tutte le possibili relazioni di parentela
- **NB** questa è una forma di inferenza monotona senza backtracking
- Modificare poi il codice in modo tale da catturare quest'ulteriore implicazione:

$$\forall x, y, z, \mathit{parent}(x, y) \land \mathit{parent}(x, z) \rightarrow \mathit{sibling}(y, z)$$

Facendo cura che il fatto *sibling* venga aggiunto una sola volta in WM (i.e., $\mathit{siblin}(y, z) \equiv \mathit{sibling}(z, y)$)

Esercizio 2

- Scrivere un programma CLIPS che simuli una ricerca backward
- partendo da un goal della forma
  `(maingoal ancestor Luigi Miriam)`
  restituisca YES se può dimostrare che Luigi è un antenato di Miriam
- in questo caso la ricerca deve essere guidata dal goal
- non tutte le possibili relazioni parentali saranno inferite

CLIPS - ASPETTI AVANZATI

Strategie di conflict resolution

- **agenda**:
  - lista di tutte le regole che hanno la parte sinistra (LHS) soddisfatta (e non sono ancora state eseguite)
  - ogni modulo (v. seguito) ha la sua propria agenda
  - agisce come uno stack
  - la regola al top è la prima ad essere eseguita
- Come sono ordinate le regole nell'agenda?

Strategie di conflict resolution

- La posizione di una **regola attiva** nell'agenda è determinata come:
  a) al di sopra di tutte le regole con *salience* (priorità) più basso ed al di sotto di tutte le regole con salience più alto
  b) fra regole con stesso salience, è la strategia di risoluzione del conflitto (conflict resolution strategy) a determinare la posizione;
  c) se una regola è attivata (insieme ad altre) per la stessa modifica della WM, e i passi a) e b) non sono in grado di specificare l'ordine, allora viene ordinata in base all'ordine con cui le regole sono scritte nel sorgente

Strategie di conflict resolution

- **salience**:
  - L'ingegnere della conoscenza può assegnare una priorità ad una regola tramite la salience rule property.
  - I valori di salience cadono nell'intervallo [-10000, 10000].
  - **NB** In genere non è mai buona norma attribuire salience in modo da forzare un ordine preciso di attivazione delle regole.

```
(defrule test-1  (declare (salience 99))
   (fire test-1)
=>
   (printout t "Rule test-1 firing." crlf))
```

Strategie di conflict resolution

- **Depth**: regole attive più recenti in cima alle altre regole di pari salience
- **Breadth**: regole attive più recenti dopo regole di pari salience
- **Simplicity**: regole attive più recenti sopra a regole con uguale o più alta *specificità*
- **Complexity**: regole attive più recenti sopra a regole con uguale o pià bassa specificità
- **LEX**, **MEA**: da *OPS5*
- **Random**: alle nuove regole attive viene assegnato un valore casuale con cui sono poi ordinate in agenda

Strategie di conflict resolution: Depth

- sono preferite regole attivate dai fatti più recenti
- esempio:
  - il `fatto-a` attiva la `regola-1` e la `regola-2`
  - il `fatto-b` attiva la `regola-3` e la `regola-4`
  - se viene asserito il `fatto-a` prima del `fatto-b`:
    - la `regola-3` e la `regola-4` saranno poste al di sopra della `regola-1` e della `regola-2` nell'agenda
    - la posizione della `regola-1` relativa alla `regola-2` e della `regola-3` relativa alla `regola-4` sarà arbitraria

Strategie di conflict resolution: Breadth

- sono preferite regole attivate dai fatti meno recenti
- esempio:
  - il `fatto-a` attiva la `regola-1` e la `regola-2`
  - il `fatto-b` attiva la `regola-3` e la `regola-4`
  - se viene asserito il `fatto-a` prima del `fatto-b`:
    - la `regola-1` e la `regola-2` saranno poste al di sopra della `regola-3` e della `regola-4` nell'agenda
    - la posizione della `regola-1` relativa alla `regola-2` e della `regola-3` relativa alla `regola-4` sarà arbitraria

Strategie di conflict resolution: Simplicity e Complexity

- **Specificity**: La specificity di una regola è determinata dal numero di confronti da effettuare nella LHS della regola.

- **Simplicity Strategy**
  - Fra tutte le regole con lo stesso salience, le nuove regole attivate sono posizionate al di sopra di tutte le attivazioni di regole con maggiore o uguale specificità.
- **Complexity Strategy**
  - Fra tutte le regole con stesso salience, le nuove regole attivate sono poste al di sopra di tutte le attivazioni di regole con minore o uguale specificità.

Strategie di conflict resolution: LEX

- Fra tutte le regole con la stessa salience, le nuove regole attivate sono ordinate come segue
  - Ogni fatto è etichettata con un "time tag" per indicarne la recentezza rispetto agli altri fatti
  - I pattern entities associati ad ogni attivazione di regola sono ordinati in modo decrescente per determinare la posizione tra le regole con lo stesso salience.
  - Un'attivazione con un pattern entities più recente è sistemata prima di una attivazione con pattern entities meno recenti.

Esempio di attivazioni di regole

pattern nell'ordine con cui sono definiti	come sono ordinati con LEX
rule6: f-1, f-4	rule6: f-4, f-1,
rule5: f-1, f-2, f-3	rule5: f-3, f-2, f-1
rule1: f-1, f-2, f-3	rule1: f-3, f-2, f-1
rule2: f-3, f-1	rule2: f-3, f-1
rule4: f-1, f-2	rule4: f-2, f-1
rule3: f-2. f-1	rule3: f-2. f-1

Strategie di conflict resolution: MEA

- Ogni fatto è etichettato con un "time tag" per indicarne la recentezza rispetto agli altri fatti
- l primo "time tag" del pattern entity associato con il primo pattern è utilizzato per determinare la posizione della regola
- Un'attivazione, il cui primo "time tag" del pattern è più grande di altri primi tag delle attivazioni, è inserita prima delle altre attivazione nell'agenda
- A parità di time tag del primo pattern entity si confrontano i time tag dei pattern successivi

Esempio di attivazioni di regole

come sono scritte le refole	come sono riordinate secondo MEA
rule6: f-1, f-4	rule2: f-3, f-1
rule5: f-1, f-2, f-3	rule3: f-2, f-1
rule1: f-1, f-2, f-3	rule6: f-1, f-4
rule2: f-3, f-1	rule5: f-1, f-2, f-3
rule4: f-1, f-2	rule1: f-1, f-2, f-3
rule3: f-2, f-1	rule4: f-1, f-2

Regole e Connettivi Logici

- Per default i pattern nell'antecedente delle regole sono in and ma non è l'unica possibilità
- `or`:
  `(or (pattern1) (pattern2))`
  soddisfatta se almeno uno dei due pattern unifica con i fatti della WM

```
(defrule celebrate
(or (birthday) (anniversary))
=>
(printout t "Let's have a party!" crlf))
```

Regole e Connettivi Logici

- `not`:
  (not (pattern))
  soddisfatto quando nessun fatto unifica con il pattern

```
(defrule working-day
(not (birthday))
(not (anniversary))
=>
(printout t "Let's work!" crlf))
```

Regole e Connettivi Logici

- `exists`:
  (exists (pattern))
  soddisfatto per un unico fatto che unifica

```
(defrule emergency-report
  (exists
    (or (emergency (emergency-type fire))
        (emergency (emergency-type bomb)))
  )
=>
    (printout t "There is an emergency." crlf )
)
```

  Se anche la WM contenesse entrambe le emergenze, la regola scatterebbe una volta sola

Regole e Connettivi Logici

- `forall`:
  (forall (pattern))
  soddisfatto se il pattern vale per tutti i fatti che unificano

```
(defrule evacuated-all-buildings
  (forall  (emergency (emergency-type fire | bomb)
                      (location ?building) )
           (evacuated (building ?building))
  )
=>
(printout t "All buildings with emergency are evacuated " crlf))
```

  La regola scatta solo se per tutti i `?building` dove è stata riscontrata una emergenza è anche stato dato l'ordine di evacuazione.

Regole e Connettivi Logici

- `forall`:

  Situazione 1: la regola scatta (una volta sola)

  ```
  (emergency (emergency-time fire) (location library))
  (evacuated (building library))
  (emergency (emergency-time bomb) (location main-room))
  (evacuated (building main-room))
  ```

  Situazione 2: la regola NON scatta

  ```
  (emergency (emergency-time fire) (location library))
  (evacuated (building library))
  (emergency (emergency-time bomb) (location main-room))
  ```

Funzioni gensym e gensym*

- `(gensym)`
  restituisce un nuovo simbolo con forma `genX` dove `X` è un intero incrementato automaticamente ad ogni invocazione della funzione. Il primo simbolo generato è gen1 I simboli generati non sono necessariamente univoci
- `(gensym*)`
  Simile a `gensym` ma con la garanzia che il simbolo generato è univoco
- `(setgen X)`
  imposta il valore iniziale del numero in coda al simbolo

Esempi:

```
CLIPS> (setgen 1)
1
CLIPS> (assert (gen1 gen2 gen3))
<Fact-0>
CLIPS> (gensym)
gen1
CLIPS> (gensym*)
gen4
CLIPS>
```

Funzioni gensym e gensym*

- Sono utili per etichettare fatti distinti che richiedono un identificatore univoco
- Possono essere usati anche per tracciare fatti distinti che però fanno riferimento collettivamente alla stessa informazione. Ad esempio, uno stato in A*.

```
(deftemplate car                (deftemplate plane
(slot state-id)                 (slot state-id)
(slot name)                     (slot name)
(slot position))                (slot position))

(car (state-id gen1) (name FIAT2) (position TO))
(car (state-id gen1) (name FIAT1) (position MI))
(plane (state-id gen1) (name AIRBUS2) (position PA))

(car (state-id gen2) (name FIAT2) (position MI))
(car (state-id gen2) (name FIAT1) (position MI))
(plane (state-id gen2) (name AIRBUS2) (position TO))
```

Funzioni gensym e gensym* come default values

```
CLIPS>
(deftemplate foo
  (slot w (default ?NONE)) ;richiede che ci sia sempre un valore
  (slot x (default ?DERIVE))
  (slot y (default (gensym*))) ;il valore è determinato alla prima asserzione di un
                               ;fatto foo
  (slot z (default-dynamic (gensym*)))) ;il valore viene determinato ad
                               ;ogni asserzione del fatto foo

CLIPS> (assert (foo))
[TMPLTRHS1] Slot w requires a value
because of its (default ?NONE) attribute.
CLIPS> (assert (foo (w 3)))
<Fact-0>
CLIPS> (assert (foo (w 4)))
<Fact-1>
CLIPS> (facts)
f-0     (foo (w 3) (x nil) (y gen1) (z gen2))
f-1     (foo (w 4) (x nil) (y gen1) (z gen3))
For a total of 2 facts.
CLIPS>
```

Funzione bind

- Lega esplicitamente un valore ad una variabile, può essere usato nel conseguente delle regole

```
(bind ?distance (+ f g))
(bind ?new (gensym*))
(assert (car (state-id ?new) (name ?name) (position MI)))
```

Input/output

```
(open <file-name> <logical-name> "r")
```

- `<file-name>` è il nome del file su disco
- `<logical-name>` è un nome usato all'interno del codice CLIPS
- "r" indica il permesso di lettura ("w" per scrivere)

```
(open "example.dat" my-file "r")
(read my-file)
```

- Al termine il file deve essere chiuso:

```
(close <logical-name>)
```

Input/output

- `read` legge un solo simbolo
- `readline` legge un'intera riga terminata da CR

Struttura generale:

- `(read <logical-name>)` legge un solo token dal file
- `(read)` legge un solo token dal terminale (il file di default)

Esempi:

- `(bind ?input (read))`
- `(bind ?$input (readline))`

PROGRAMMAZIONE MODULARE

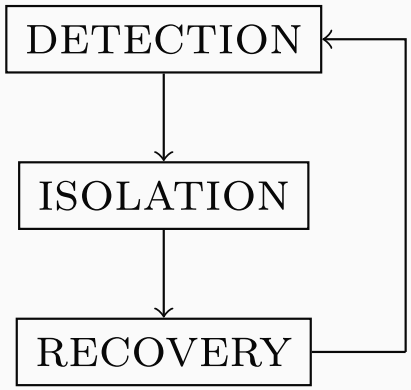
Il problema

- Programmi in domini reali coninvolgono facilmente migliaia di regole
- come abbiamo visto, non tutte le regole codificano *conoscenza di dominio* (cioè che riguardano il problema che si vuole risolvere), alcune regole codificano infatti *conoscenza di controllo*, che regola il comportamento del programma
- L'interleaving tra conoscenza di controllo e di dominio può rappresentare un serio problema nella manutenzione della KB nel tempo.

Modularità

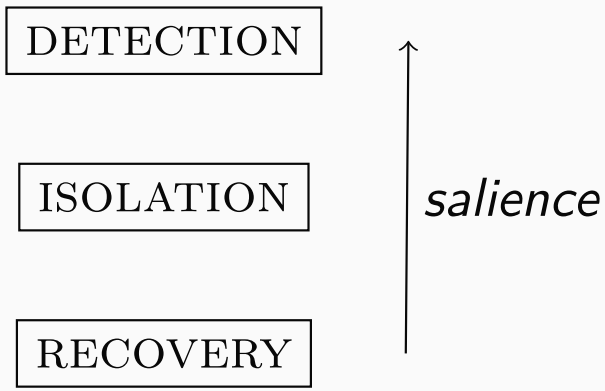
- Immaginiamo di dover risolvere un problema di diagnosi di un dispositivo
- l'analisi diagnostica si svolge in tre fasi successive:
  - Detection: riconoscere che esiste un guasto
  - Isolation: determinare quale componente del dispositivo ha causato il guasto
  - Recovery: determinare quali passi seguire per ripristinare la corretta funzione del dispositivo

  Il processo può essere iterativo:



Prima possibile soluzione: Salience

- Per fare in modo che le regole della fase *recovery* vengano eseguite solo dopo che la fase di *isolation* sia terminata, e abbia seguito la fase di *detection*, si potrebbe ricorrere alla salience



- due possibili svantaggi:
  - la conoscenza di controllo è ancora distribuita tra le regole
  - non garantisce il corretto ordine deelle'esecuzione

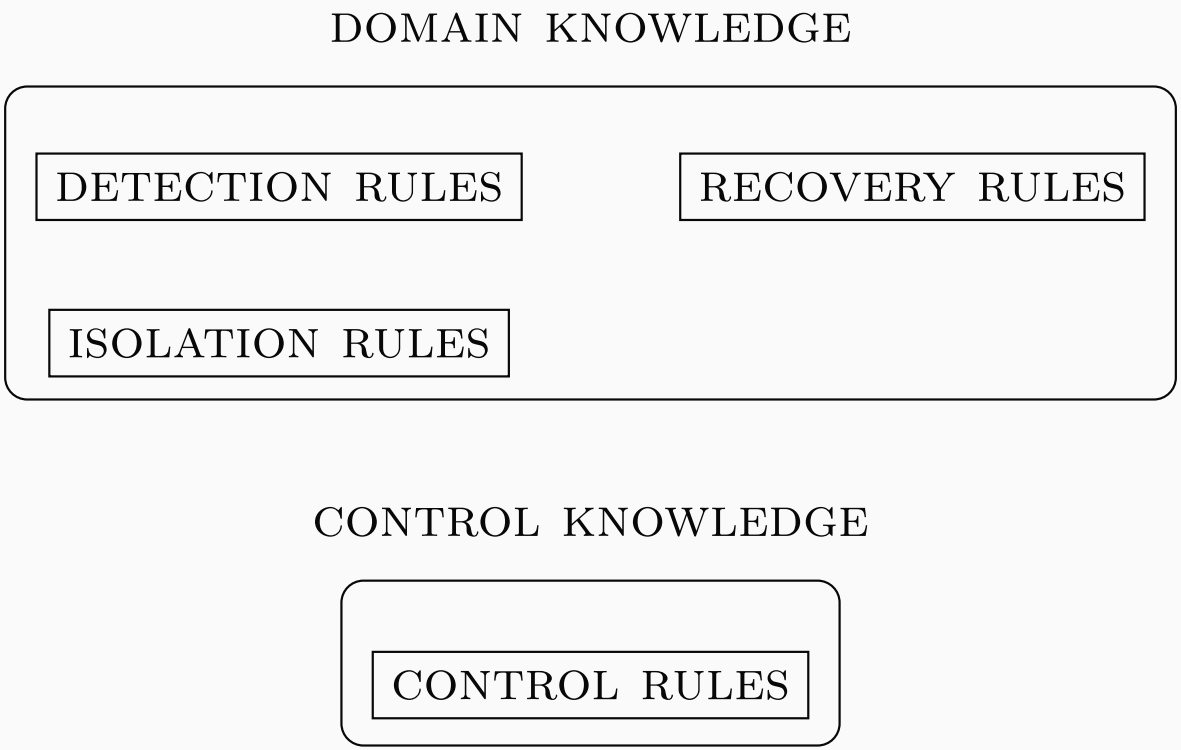
Prima possibile soluzione: Salience - Esempio

```
(defrule fire-first (declare (salience 30))
  (priority first)
=>
  (printout t "Print first" crlf))

(defrule fire-second (declare (salience 20))
  (priority second)
=>
  (printout t "Print second" crlf))

(defrule fire-third (declare (salience 10))
  (priority third)
=>
  (printout t "Print third" crlf))
```

- Se in WM sono presenti i fatti
  `(priority first)`, `(priority second)`, `(priority third)`
  l'agenda ordina le regole come ci si aspetta
- Ma se la WM contenesse solo: `(priority second)`, `(priority third)`
  allora regole della seconda fase sarebbero eseguite prima delle regola della prima fase

Fasi e Fatti di Controllo

- Una soluzione migliore per controllare il flusso dell'esecuzione è quella di separare *control knowledge* da *domain knowledge*
  - ad ogni regola è assegnato un control pattern che indica in quale fase quella regola è applicabile
  - regole di controllo sono poi definite per traferire il controllo da una fase alla successiva



Fasi e Fatti di Conrollo

- Regole di controllo

```
(defrule detection-to-isolation
   (declare (salience -10))
   ?phase <- (phase detection)
=>
   (retract ?phase)
   (assert (phase isolation)))

(defrule isolation-to-recovery
   (declare (salience -10))
   ?phase <- (phase isolation)
=>
   (retract ?phase)
   (assert (phase recovery)))

(defrule recovery-to-detection
   (declare (salience -10))
   ?phase <- (phase recovery)
=>
   (retract ?phase)
   (assert (phase detection)))
```

Fasi e Fatti di Controllo

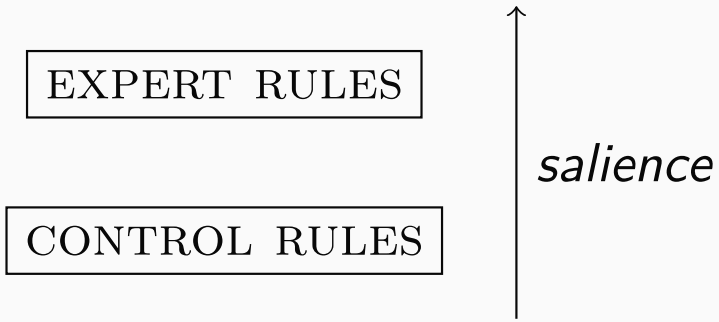
- Ogni regola che appartiene ad una fase specifica ha come primo pattern element un fatto di controllo

```
(defrule find-fault-location-and-recovery
   (phase recovery)
   (recovery-solution switch-device ?replacement on)
=>
   (printout t "Switch device" ?replacement "on" crlf) )
```

- In combinazione con la MEA questo pone tutte le regole attive della fase più recente in cima all'agenda

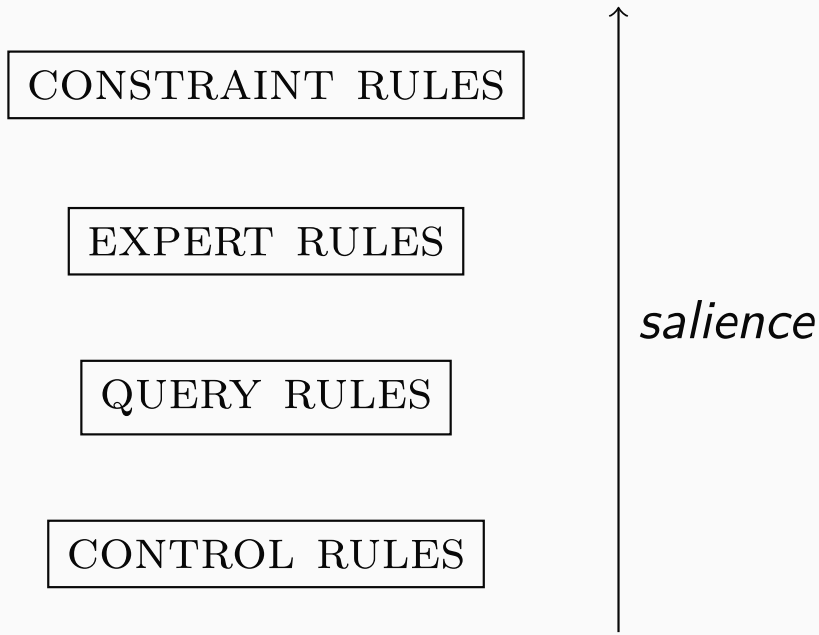
Fasi e Fatti di Controllo

- Osservazione: mentre è attiva la fase di *detection* sarà anche attiva la regola di controllo `detection-to-isolation`
- La salience più bassa di questa regola garantisce che sarà eseguita solo dopo che le tutte le regole della fase di *detection* sono state eseguite



Fasi e Fatti di Controllo

- In genere possiamo evidenziare quattro classi di knowledge che possono essere modellate in un sistema esperto



Fasi e Fatti di Controllo

- Un possibile svantaggio:
  - Fatti di controllo partizionano logicamente la KB, ma la WM rimane unica
  - Regole in una fase $phase_1$ potrebbero accedere e modificare fatti che sono di esclusiva competenza di una fase successiva $phase_i$
  - Il problema è che con il crescere del programma non è sempre facile riconoscere queste situazioni, visto che il programma potrebbe essere scritto da persone diverse anche nell'arco di un periodo lungo (vedi problemi di manutenibilità)

Fasi e Fatti di Controllo

- Uno svantaggio ancora più importante:
  - In certe situazioni vorrei poter passare da una fase all'altra in modo più flessibile
  - ad esempio, vorrei passare da *isolation* a *recover* per sistemare un particolare componente e poi tornare di nuovo alla fase di *isolation* e ritrovare tutti i fatti e le regole in agenda di prima
  - con i fatti di controllo questo non è possibile nel modo corretto perché richiede di rimuovere e riasserire il fatto `(phase isolation)`
  - ri-asserire `(phase isolation)` potrebbe abilitare molte più regole del voluto perché inibisce la rifrazione
- occorre quindi un modo per partizionare WM e KB più strutturato $\Rightarrow$ i **moduli**

Moduli

- CLIPS usa i moduli per partizionare KB e WM

```
(defmodule <module-name> [<comment>])
```

- In assenza di indicazioni, tutti i fatti, template e regole sono definite nel modulo `MAIN`
- volendo allora risolvere il problema della diagnosi di un dispositivo possiamo definire i tre moduli:

```
CLIPS> (defmodule DETECTION)
CLIPS> (defmodule ISOLATION)
CLIPS> (defmodule RECOVERY)
```

- da qui in avanti regole e template sono definite per default nell'ultimo modulo definito (che diventa quello corrente)
- ma si può specificare il modulo di destinazione al momento della definizione:

```
(defrule DETECTION::rule-detection-1 ... => ... )
(deftemplate ISOLATION::component (slot name) .... )
```

Moduli

- NB i Moduli partizionanto non solo la KB, ma anche la WM e di conseguenza anche l'agenda $\Rightarrow$ **ogni modulo ha la propria agenda e WM**
- alcuni comandi utili per consultare WM/KB in presenza di moduli:
  - `(get-current-module)`
  - `(set-current-module` *`module-name`*`)`
  - `(list-defrules)` visualizza il contenuto del modulo corrente
  - `(list-defrules` *`module-name`*`)` visualizza le regole definite in *`module-name`* anche se non è il modulo corrente
  - `(list-defrules *)` visualizza le regole definite in tutti i modi
  - il comportamento per `(facts)`, `(agenda)`, etc. è simile

Moduli - Condivisione

- Ovviamente i moduli devono poter scambiare qualche informazione, altrimenti averli completamente partizionati non sarebbe di alcuna utilità
- i costrutti `(defrule)` e `(deffacts)` sono strettamente privati, non possono cioè essere condivisi
- i costritti `(deftemplate)` possono essere definit in un modulo ed importati in altri
- quando un `(deftemplate)` è condiviso tutti i fatti non ordinati istanza di quel template sono anch'essi condivisi

Moduli - Condivisione

- Esportare:
  - `(defmodule` *`module-name`* `(export ?ALL))`
  - `(defmodule` *`module-name`* `(export ?NONE))`
  - `(defmodule` *`module-name`* `(export deftemplate ?ALL))`
  - `(defmodule` *`module-name`* `(export deftemplate ?NONE))`
  - `(defmodule` *`module-name`* `(export deftemplate` *`<template-name>`*$^{+}$`))`
- Importare:
  - `(defmodule` *`module-name`* `(import ?ALL))`
  - `(defmodule` *`module-name`* `(import ?NONE))`
  - `(defmodule` *`module-name`* `(import deftemplate ?ALL))`
  - `(defmodule` *`module-name`* `(import deftemplate ?NONE))`
  - `(defmodule` *`module-name`* `(import deftemplate` *`<template-name>`*$^{+}$`))`

Moduli - Focus

- le regole che sono prese in considerazione per l'esecuzione sono quelle del modulo che in questo momento ha il *focus*, che non corrisponde all'ultimo modulo definito!
- i comandi di `(reset)` e `(clear)` portano il focus sul modulo MAIN
- L'esecuzione del programma parte quindi sempre dal modulo MAIN
- CLIPS gestisce il passaggio da un modulo ad un altro attraverso uno stack chiamato appunto **focus**

Moduli - Focus

- con il comando (`focus` *`module-name`*) pone in cima allo stack focus il modulo indicato
- di conseguenza, tutte le regole definite in quel modulo saranno prese in esame ed eventualmente eseguite
- un modulo rimane in cima allo stack fino a quando
  - ci sono regole attive da eseguire, quando non ci sono più regole eseguibili, il modulo viene rimosso dalla cima dello stack (pop implicito)
  - una regola esegue il comando esplicito di pop: (`pop focus`) oppure (`return`)
  - una regola aggiunge un ulteriore modulo in cima allo stack (`focus` *`altro-modulo`*)

Moduli - Focus

- è possibile specificare più moduli con il comando focus:
  `(focus DETECTION ISOLATION RECOVER)`
  il top è a sinistra
- il comando `(list-focus-stack)` visualizza il contenuto dello stack,
- in alternativa si può usare anche `(get-focus-stack)`
- anche il focus ha il suo watch: `(watch focus)`

Auto-focus

- E' possibile specificare che alcune regole sono così importanti che, se le loro precondizioni diventano soddisfatte sono poste in cima all'agenda anche quando il modulo che ha il focus non è quello in cui sono definite
- un caso dove questo è molto utile è quando la regola riconosce la violazione di un vincolo quando

```
(defmodule CHECK (import MAIN ?ALL) (import FILM ?ALL))
(defrule checks (declare (auto-focus TRUE))
   ?x <- (ticket (name ?n) (title ?t))
  (film (title ?) (VM YES))
  (person (name ?n) (age ?a&:(< ?a 18)))
=>
  (retract ?x)
  (printout t "No film for " ?n crlf)
)
```

Sostituire i fatti di controllo con i moduli

```
(deffacts MAIN::control-information
    (phase-sequence DETECTION ISOLATION RECOVERY))

(defrule MAIN::change-phase
   ?list <- (phase-sequence ?next-phase  $?other-phases)
=>
   (focus ?next-phase)
   (retract ?list)
   (assert (phase-sequence ?other-phases ?next-phase)))
```

- ogni volta che una fase termina il focus torna al mail dove la regola `change-phase` sposta il focus sulla fase successiva
- confrontando questa soluzione con quella che fa ricorso ai fatti di controllo i vantaggi dovrebbero essere evidenti: molte meno regole, più facili da mantenere (es. invertire o aggiungere fasi)

RICERCA CON BACKTRACKING

Un po' di maniere...

- Consideriamo il problema di far sedere N persone ad un tavolo in modo tale che:
  - ogni persona abbia accanto una persona del sesso opposto
  - ogni persona condivida almeno un hobby con i suoi vicini

Un po' di maniere...

- Consideriamo il problema di far sedere N persone ad un tavolo in modo tale che:
  - ogni persona abbia accanto una persona del sesso opposto
  - ogni persona condivida almeno un hobby con i suoi vicini
- Cominciamo con il modellare gli ospiti:

```
(deftemplate guest(slot name)(slot sex)(slot hobby))
```

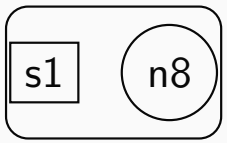
Modellare lo spazio di ricerca

- Ogni nodo dello spazio di ricerca corrisponde ad un assegnaemto di persone a certe sedie
- Più stati saranno aperti durante la ricerca
- Devo poter distinguere tra assegnamenti validi in uno stato da quelli validi in un altro stato
- occorre un identificatore dello stato:

```
(deftemplate path
    (slot id) ;accomuna i fatti appartenenti allo stesso stato
    (slot name)
    (slot seat))
```

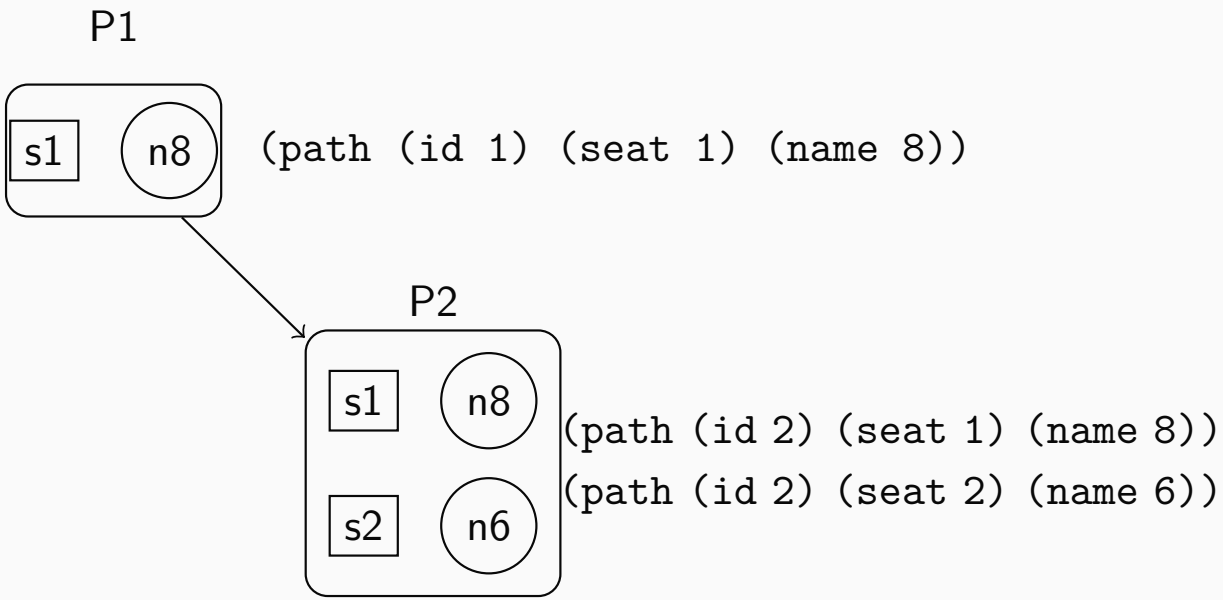
Modellare lo spazio di ricerca - Esempio

P1

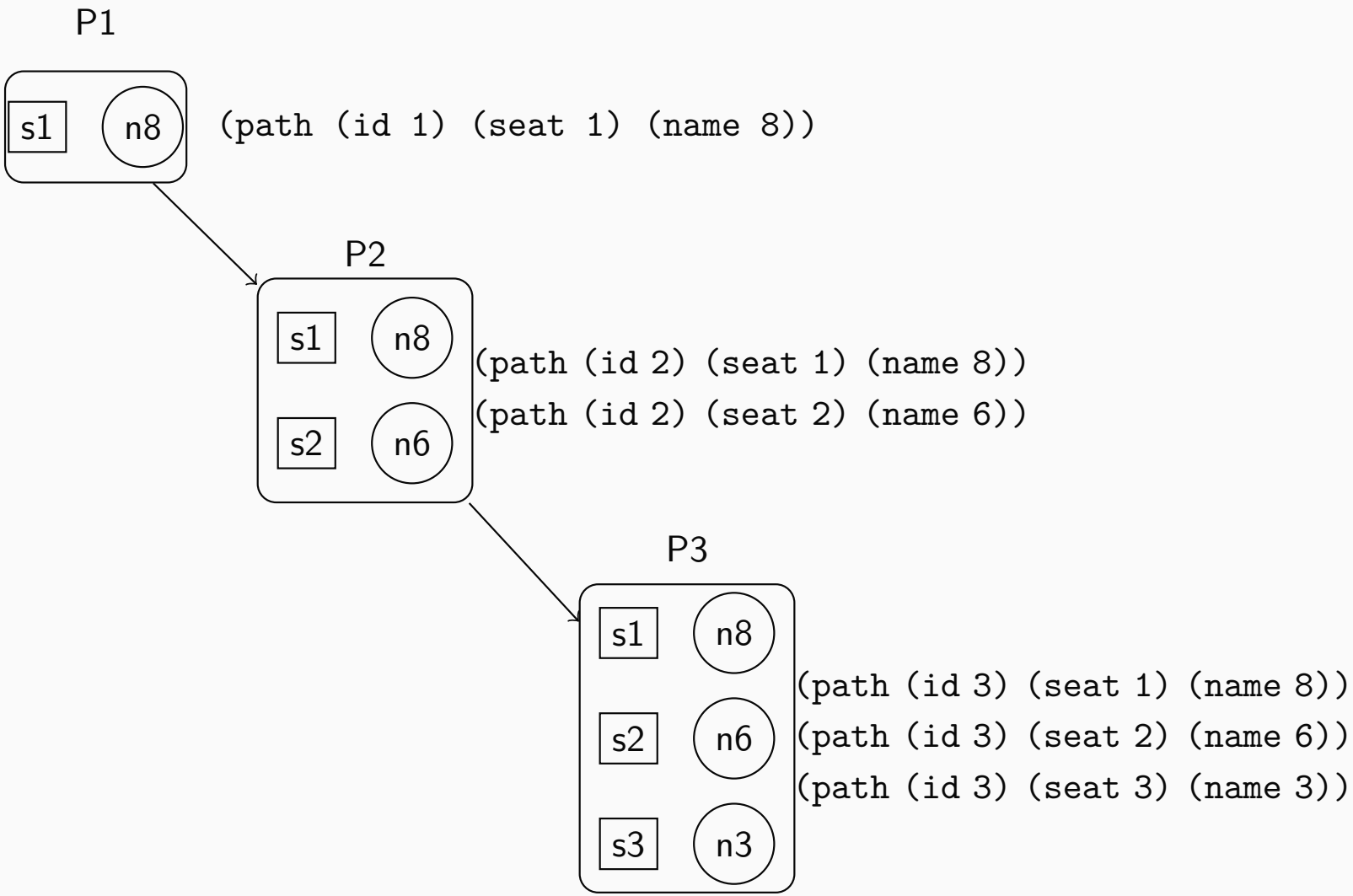



```
(path (id 1) (seat 1) (name 8))
```

Modellare lo spazio di ricerca - Esempio



Modellare lo spazio di ricerca - Esempio



Modellare lo stato

- Altra struttura di supporto per la definizione dello stato corrente
- coppie di assegnamenti consecutivi:

```
(deftemplate seating
   (slot seat1) (slot name1) (slot name2) (slot seat2)
   (slot id) (slot pid) (slot path_done))
```

- ogni assegnamento ha un suo `id`, ma appartiene ad un path (`pid`):

```
seating (seat1 1) (name1 n8) (seat2 2) (name2 n8) (id 1) (pid 0) (path_done yes))
seating (seat1 1) (name1 n8) (seat2 2) (name2 n8) (id 2) (pid 1) (path_done yes))
seating (seat1 1) (name1 n8) (seat2 2) (name2 n8) (id 3) (pid 2) (path_done yes))
```

Input

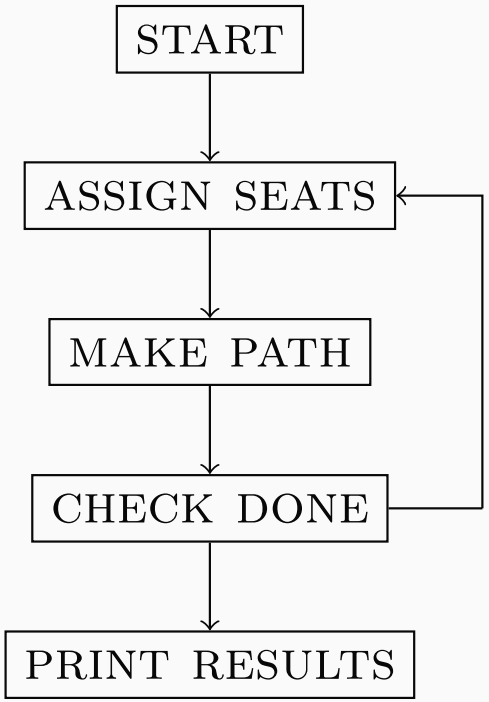
- gli ospiti e i loro hobby

```
(deftemplate guest (slot name) (slot sex) (slot hobby) )
```

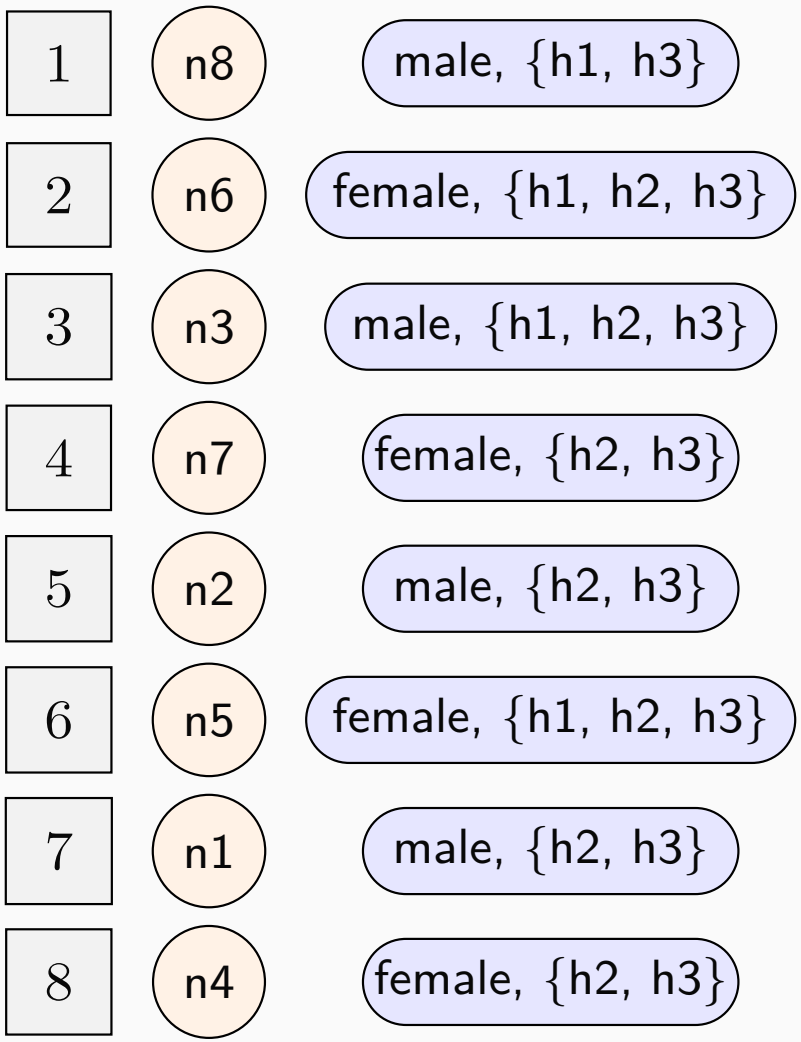
```
(guest (name n1) (sex m) (hobby h3)) (guest (name n1) (sex m) (hobby h2))
(guest (name n2) (sex m) (hobby h2)) (guest (name n2) (sex m) (hobby h3))
(guest (name n3) (sex m) (hobby h1)) (guest (name n3) (sex m) (hobby h2))
(guest (name n3) (sex m) (hobby h3))
(guest (name n4) (sex f) (hobby h3)) (guest (name n4) (sex f) (hobby h2))
(guest (name n5) (sex f) (hobby h1)) (guest (name n5) (sex f) (hobby h2))
(guest (name n5) (sex f) (hobby h3))
(guest (name n6) (sex f) (hobby h3)) (guest (name n6) (sex f) (hobby h1))
(guest (name n6) (sex f) (hobby h2))
(guest (name n7) (sex f) (hobby h3)) (guest (name n7) (sex f) (hobby h2))
(guest (name n8) (sex m) (hobby h3)) (guest (name n8) (sex m) (hobby h1))
```

Ricerca

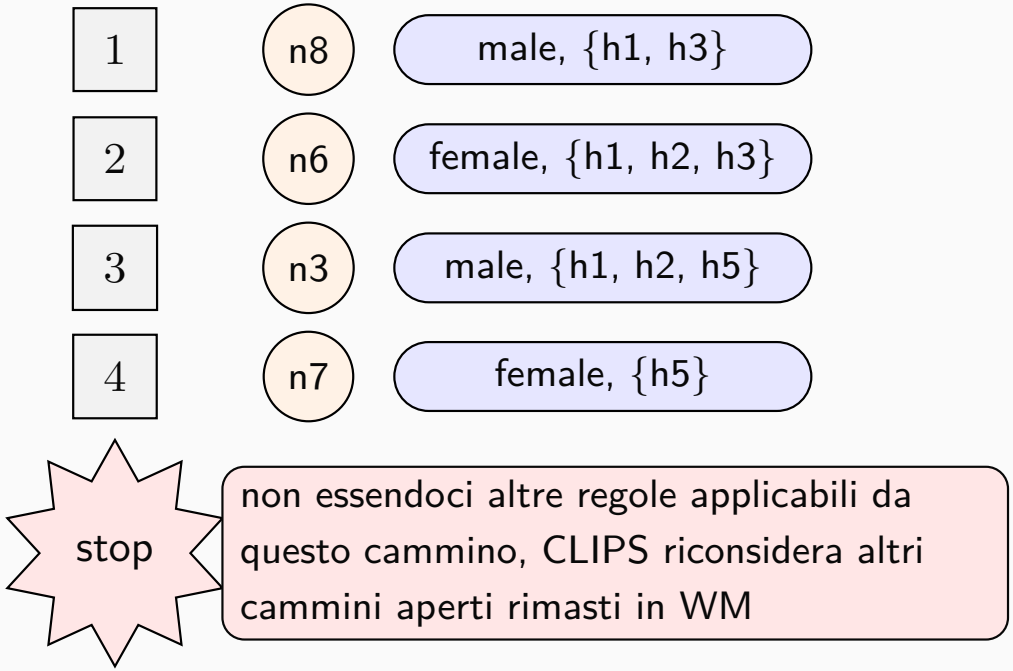
- Organizzata per contesti/fasi



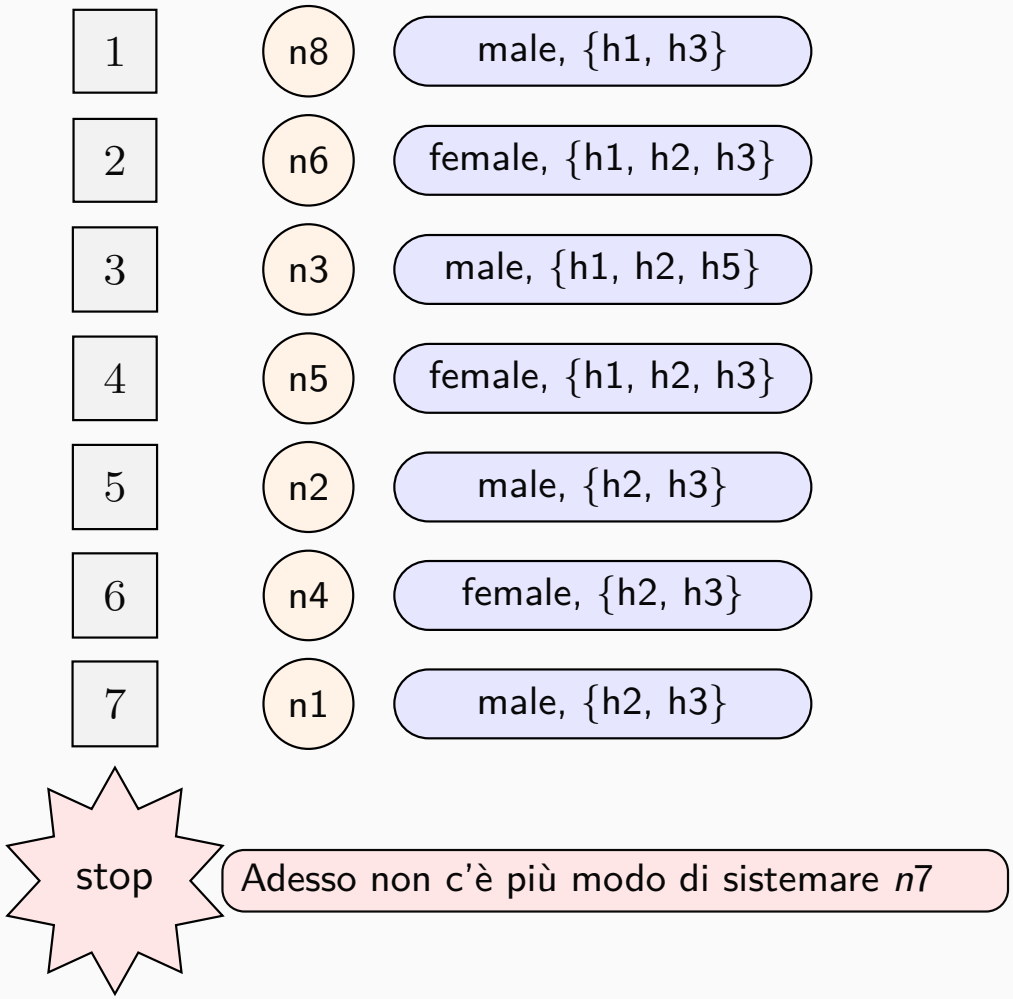
Esempio di esecuzione senza backtracking



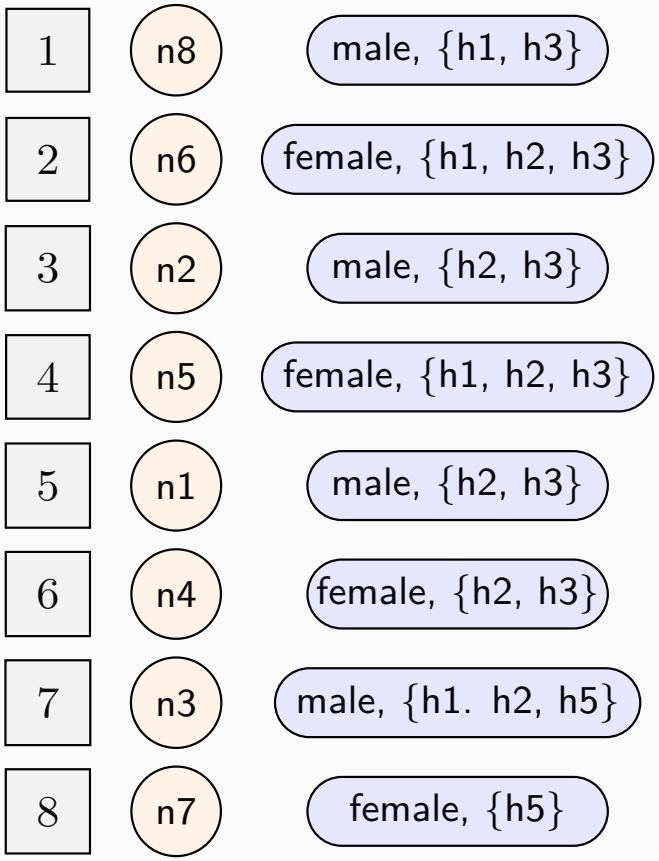
Esempio di esecuzione con backtracking



Esempio di esecuzione con backtracking



Esempio di esecuzione con backtracking



- Dopo molti backtracking, verrà esplorata la soluzione in cui $n3$, l'unico compatibile con $n7$, siederà in penultima posizione.

INCERTEZZA - CERTAINTY FACTORS

Certainty Factors

- CLIPS non ha una funzionalità built-in per la gestione del ragionamento incerto
- E' possibile però incorporare incertezza usando una strategia simile a quella usata in MYCIN

Esempio di regola MYCIN

```
IF (1) The stain of the organism is gramneg and
   (2) The morphology of the organism is rod and
   (3) The patient is a compromised host
THEN There is suggestive evidence (0.6) that the identity of the
organism is pseudoinonas
```

- Intuitivamente la regola equivale ad una probabilità condizionata

$$P(H|E_1 \cap E_2 \cap E_3) = 0.6$$

Certainty Factors $\neq$ Probabilità

- MYCIN opera nel contesto medico
- I medici non ragionano in termini probabilistici, ma per gradi di *belief* e *disbelief*
- questi gradi di belief non sono necessariamente consistenti in termini probailistici!
- cioè il medico non ritiene necessariamente vera l'equazione

$$P(H'|E_1 \cap E_2 \cap E_3) = 1 - 0.6$$

  dove H' è un'ipotesi complementare ad H.
- La ragione è che:

Certainty Factors $\neq$ Probabilità

- MYCIN opera nel contesto medico
- I medici non ragionano in termini probabilistici, ma per gradi di *belief* e *disbelief*
- questi gradi di belief non sono necessariamente consistenti in termini probailistici!
- cioè il medico non ritiene necessariamente vera l'equazione

$$P(H'|E_1 \cap E_2 \cap E_3) = 1 - 0.6$$

  dove H' è un'ipotesi complementare ad H.
- La ragione è che:

  - mentre è vero che E_1, E_2, E_3 hanno una relazione causa-effetto con H, da cui segue $P(H|E_1 \cap E_2 \cap E_3)$
  - non è necessariamente vero che E_1, E_2, E_3 hanno una relazione causa-effetto con H', quindi $P(H'|E_1 \cap E_2 \cap E_3)$ è corretta probabilisticamente, ma non è giustificata nella pratica

Certainty Factors: Un esempio concreto

- Immaginiamo di partecipare ad una gara di salto in alto
- Dopo un certo numero di salti avete un'ultima possibilità per qualificarvi alla sessione di gare successive
- Vi occorre un buon salto per guadagnare abbastanza punti e posizionarvi bene in classifica.
- Calcolate allora che se fate un salto di almeno 1.8m potrete qualificarvi con probabilità 0.7

$$P(qualifica|salto1.8) = 0.7$$

Certainty Factors: Un esempio concreto

- Immaginiamo di partecipare ad una gara di salto in alto
- Dopo un certo numero di salti avete un'ultima possibilità per qualificarvi alla sessione di gare successive
- Vi occorre un buon salto per guadagnare abbastanza punti e posizionarvi bene in classifica.
- Calcolate allora che se fate un salto di almeno 1.8m potrete qualificarvi con probabilità 0.7

$$P(qualifica|salto1.8) = 0.7$$

- Molte cose possono andare diversamente da quanto previsto:
  - Gli altri concorrenti saltano molto meglio di voi
  - Un arbirto mette in discussione il vostro salto
  - ...
- Di conseguenza non ha molto senso dire che

$$P(nonqualifica|salto1.8) = 0.3$$

perché non è solo il salto di per sé che può causare la non qualifica.

Certainty Factors: Definizione

- Certainty Factors (Carnap 1950) come probabilità alternativa per catturare un grado di conferma (*degree of confirmation* o *epistemic probability*)
- Un CF conferma un'ipotesi data una qualche evidenza
- In MYCIN un CF è stata definita come la differenza tra belief e disbelief:

$$CF(H, E) = MB(H, E) - MD(H, E)$$

dove

  - CF è il fattore di certezza nell'ipotesi H data l'evidenza E
  - MB è la misura di crescita della credenza (belief) in H dato E
  - MD è la misura di riduzione della credenza in H data E

Certainty Factors: Definizione

Certainty Factor

rappresenta il grado "netto" di credenza sulla verità di un'ipotesi basata su una qualche evidenza. E' un valore nell'intervallo [-1, 0, +1].

- 1 il fatto è noto essere vero per certo
- 0 non ci sono abbastanza elementi per affermare che il fatto sia vero o falso (incertezza massima)
- -1 il fatto è noto essere falso

- Un CF combina quindi belief e disbelief in un unico valore
- Questo comporta almeno due vantaggi:
  - Ranking delle ipotesi
  - Un CF consente ad un esperto di esprimere un grado di belief senza necessariamente sottoscrivere un valore per il disbelief.

Certainty Factors: Osservazione

- Poiché $CF(H, E)$ è una misura della conferma dell'ipotesi H data l'evidenza E
- e poiché $CF(H', E)$ è una misura della non credenza all'ipotesi complementare H' data E
- Ma allora deve valere che

$$CF(H, E) + CF(H', E) = 0$$

Come simulare MYCIN in CLIPS: OAV

- MYCIN rappresenta fatti come una tripletta *object-attribute-value* (OAV)
- Questa trippletta può essere modellata usando il seguente template

```
(defmodule OAV (export deftemplate oav))
(deftemplate OAV::oav
    (multislot object (type SYMBOL))
    (multislot attribute (type SYMBOL))
    (multislot value)
    (slot CF (type FLOAT) (range -1.0 +1.0)))
```

Modellare Conoscenza da MYCIN a CLIPS (esempio)

```
(oav (object organism)
(attribute stain)
(value gramneg)
(CF 0.3))

(oav (object organism)
(attribute morphology)
(value rod)
(CF 0.7))

(oav (object patient)
(attribute is a)
(value compromised host)
(CF 0.8))
```

Simulare il ragionamento di MYCIN

- MYCIN consente di derivare la stessa tripla OAV da path distinti e quindi combinare i corrispondenti fattori di certezza
- CLIPS per default non consente di avere fatti identici
- due OAV con CF diversi sono ammessi, ma non due OAV con medesimo CF
- E' possibile disabilitare questa restrizione:
  `(set-fact-duplication TRUE)`
- Come combinare i CF?

Simulare il ragionamento di MYCIN

- Dati due OAV identici nei loro primi tre slot
  - oav_1 con CF_1
  - oav_2 con CF_2
- Si ottiene un nuovo *oav* con $NewCertainty = (CF_1 + CF_2) - (CF_1 * CF_2)$
- Esempio

```
(oav (object organism)          (oav (object organism)
(attribute morphology)          (attribute morphology)
(value rod)                     (value rod)
(CF 0.7))                       (CF 0.5))
```

New Certainty = (0.7 + 0.5) - (0.7 * 0.5) = 0.85

Implementativamente

```
(defrule OAV::corabine-certainties-both-positive
        (declare (auto-focus TRUE)
   ?factl <- (oav (object $?o)
        (attribute $?a)
        (value $?v)
        (CF ?C1&:(>= ?C1 0)))
   ?fact2 <- (oav (object $?o)
        (attribute $?a)
        (value $?v)
        (CF ?C2&:(>= ?C2 0)))
   (test (neq ?factl ?fact2))
=>
  (retract ?factl)
  (bind ?C3 (- (+ ?C1 ?C2) (* ?C1 ?C2)))
  (modify ?fact2 (CF ?C3)))
```

- L'auto-focus garantisce che due oav identici saranno uniti prima di essere utilizzati in altre regole

E quando i CF non sono entrambi positivi...

- Oltre alla precedente regola vanno anche aggiunte altre due regole che considerano tutte le possibilità
- Se CF1 e CF2 sono entrambi negativi

$$New\ Certainty = (CF_1 + CF_2) + (CF_1 * CF_2)$$

- Se hanno segno opposto e quindi $-1 \leq CF_1 * CF_2 < 0$

$$New\ Certainty = \frac{CF_1 + CF_2}{1 - min\{|CF_1|, |CF_2|\}}$$

Non basta! Come combino i CF dei fatti nelle attivazioni?

- Una regola è attivata da fatti (LHS), ciascuno con il proprio CF, come combino questi CF?
- allo stesso tempo, la regola attivata produrrà nuovi fatti (RHS), con quale CF?

Combinare i CF delle attivazioni (LHS)

- A seconda del conditional element usato nell'antecedente possiamo avere:
  - $CF(P1 \textit{ or } P2) = max\{CF(P1),\ CF(P2)\}$
  - $CF(P1 \textit{ and } P2) = min\{CF(P1),\ CF(P2)\}$
  - $CF(\textit{not } P) = -CF(P)$

  Dove $P1$ e $P2$ sono pattern che sono soddisfatti da fatti
- MYCIN aggiunge un ulteriore vincolo: se il CF risultante è inferiore a 0.2, la regola viene considerata inapplicabile (il razionale è che la regola è troppo poco probabile)

Ottenere i CF dei fatti asseriti (RHS)

- Il CF dei fatti prodotti è determinato moltiplicando il CF dell'asserzione con il CF dell'antecedente

Esempio di regola MYCIN

```
IF (1) The stain of the organism is gramneg and
   (2) The morphology of the organism is rod and
   (3) The patient is a compromised host
THEN There is suggestive evidence (0.6) that the identity of the
organism is pseudoinonas
```

- Qui 0.6 è il CF dell'asserzione, ma è espressa in termini assoluti, senza cioè considerare il grado di certezza sui fatti che attivano la regola.

In CLIPS...

```
(defmodule IDENTIFY (import OAV deftemplate oav))
  (defrule IDENTIFY::MYCIN-to-CLIPS-translation
    (oav (object organism)
      (attribute stain)
      (value gramneg)
      (CF ?C1))
    (oav (object organism)
      (attribute morphology)
      (value rod)
      (CF ?C2))
    (oav (object patient)
      (attribute is a)
      (value compromised host)
      (CF ?C3))
     (test (> (min ?C1 ?C2 ?C3) 0.2))
   =>
    (bind ?C4 (* (min ?C1 ?C2 ?C3> 0.6))
    (assert (oav (object organism)
       (attribute identity)
       (value pseudomonas)
       (CF ?C4))))
```

UN ESEMPIO DI USO DEI CF: WINE

Wine: Come scegliere il vino

- Obiettivo: determinare con un certo CF quale vino scegliere sulla base dei gusti dell'utente e del cibo che andrà servito
- il programma è organizzato nei seguenti moduli
  - QUESTIONS –> RULES
  - CHOOSE-QUALITIES
  - WINES
  - PRINT-RESULTS

Modellare l'expertise

- Le regole di dominio sono del tipo:

  if `tastiness is average`
  then `best-body is light with certainty 30 and`
  `best-body is medium with certainty 60 and`
  `best-body is full with certainty 30`

- i CF sono nell'intervallo [0,100]
- notare che

  la clausola *if* ha sempre il formato
  if <*attribute*> is <*value*>

  questo semplifica la parsificazione della regola (vedi seguito)

Modellare l'expertise

- Le regole di dominio sono nettamente separate dalle regole di controllo
- Sono definite da fatti non ordinati

```
(deftemplate RULES::rule
(slot certainty (default 100.0))
(multislot if)
(multislot then))
```

Ad esempio

```
(rule (if tastiness is average)
      (then best-body is light with certainty 30 and
            best-body is medium with certainty 60 and
            best-body is full with certainty 30))
```

- notare che la regola (asserzione) ha CF 100, mentre le singole conseguenze hanno diversi CF, la cui somma non è 100 pur essendo mutuamente esclusivi!

Il modulo QUESTIONS

- Definisce le (meta-)regole per porre ed interpretare domande
- le domande, come le regole, sono definite come fatti non ordinati

```
(question (attribute main-component)
    (the-question "Is the main component of the meal meat,
        fish, or poultry? ")
    (valid-answers meat fish poultry unknown))
```

Regole per domande

```
(defrule QUESTIONS::ask-a-question
  ?f <- (question (already-asked FALSE)
  (precursors)
  (the-question ?the-question)
  (attribute ?the-attribute)
  (valid-answers $?valid-answers))
=>
  (modify ?f (already-asked TRUE))
  (assert (attribute (name ?the-attribute)
      (value (ask-question ?the-question ?valid-answers)))))
```

- `(ask-question)` è una funzione che visualizza la domanda, legge la risposta (verificandone la correttezza) e la assegna a `value`.

il modulo RULES

- Definisce le (meta-)regole per interpretare (e semplificare) le regole di dominio

```
(defrule RULES::remove-is-condition-when-satisfied
   ?f <- (rule (certainty ?c1)
               (if ?attribute is ?value $?rest))
         (attribute (name ?attribute)
                    (value ?value)
                    (certainty ?c2))
   =>
      (modify ?f (certainty (min ?c1 ?c2)) (if ?rest)))
```

- In passaggi successivi il CF dell'antecedente accumula la certezza data dalla combinazione di input dell'utente

il modulo WINES

- il database dei vini con le loro caratteristiche:

```
(wine (name Gamay) (color red) (body medium)
      (sweetness medium sweet))
(wine (name Chablis) (color white) (body light)
      (sweetness dry))
(wine (name Sauvignon-Blanc) (color white) (body medium)
      (sweetness dry))
(wine (name Chardonnay) (color white) (body medium full)
      (sweetness medium dry))
 ...
```

Notare che possono esistere più varianti dello stesso vino (es. Chardonnay può variare nel corpo e nella dolcezza)

il modulo WINES

- la regola per generare le proposte:

```
(defrule WINES::generate-wines
  (wine (name ?name)
        (color $? ?c $?)
        (body $? ?b $?)
        (sweetness $? ?s $?))
  (attribute (name best-color) (value ?c) (certainty ?certainty-1))
  (attribute (name best-body) (value ?b) (certainty ?certainty-2))
  (attribute (name best-sweetness) (value ?s) (certainty ?certainty-
=>
  (assert (attribute (name wine) (value ?name)
          (certainty (min ?certainty-1 ?certainty-2 ?certainty-3))))
```

Se esiste un vino che soddisfa le caratteristiche (`attribute`) determinate dalle preferenze dell'utente, quel vino è suggerito con CF pari al minimo dei CF degli attributi

Regola generale per la combinazione dei CF

```
(defrule MAIN::combine-certainties ""
                    (declare (salience 100)
                    (auto-focus TRUE))
  ?rem1 <- (attribute (name ?rel) (value ?val) (certainty ?per1))
  ?rem2 <- (attribute (name ?rel) (value ?val) (certainty ?per2))
  (test (neq ?rem1 ?rem2))
 =>
      (retract ?rem1)
      (modify ?rem2
            (certainty (/ (- (* 100 (+ ?per1 ?per2))
                             (* ?per1 ?per2)) 100))))
```