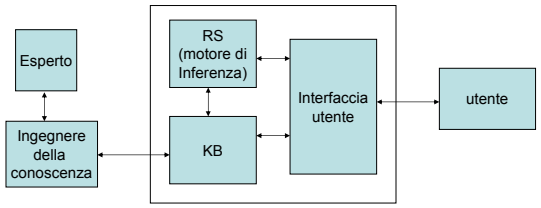
# Lezione 17

Ingegneria della conoscenza



## Ingegneria della conoscenza

- Un settore in cui la rappresentazione della conoscenza e la sua acquisizione sono centrali è quello dei sistemi esperti.
- La situazione è illustrata in figura.






## Gli attori

- Ingegneri del Software:
  - Conoscenza degli strumenti software, non del sistema esperto
- Ingegneri della conoscenza
  - Conoscenza del sistema esperto (diretta) e del dominio di problema (tramite l'esperto)
- Esperti del dominio:
  - Conoscenza del dominio di problema
- Utenti
  - Applicano il sistema per risolvere problemi specifici
- Ingegneri della conoscenza ed esperti costruiscono la conoscenza **generale,** gli utenti introducono la conoscenza relativa ai **casi specifici** (ricordare i livelli di conoscenza individuati nell'esempio dell'impianto elettrico)



## Gli strumenti per l'ingegneria della conoscenza

- Vi sono diversi RRS utilizzati dall'ingegnere della conoscenza
- Tuttavia si può individuare un nucleo di strumenti comuni.
- Ne vedremo alcuni in termini astratti, indipendenti dal particolare RRS, utilizzando DCL come strumento base.



## Meta-interpreti

- Per definire in DCL gli strumenti che considereremo, dobbiamo "implementare" in esso motori di inferenza con caratteristiche diverse.
- Uno strumento per far ciò è l'uso di un meta-interprete man mano arricchibile e adattabile.
- Partiremo da un meta-interprete di base e proseguiremo con vari arricchimenti.



## Vanilla Meta-Interpreter

- E' il meta-interprete di base

LINGUAGGIO OGGETTO: è Prolog. Usando:
op(150, xfy, &) come simbolo di funzione;
atomi a livello oggetto come termini nel meta-livello;
op(200, xfx, '<-') come simbolo di predicato binario;
le clausole diventano fatti del meta-livello.

ESEMPIO:

La clausola oggetto p(f(X),Y) :- h(X), g(Y).
diventa il meta-fatto p(f(X),Y) <- h(X)&h(Y).

- Vanilla è

```
prove(true).
prove(A&B) :- prove(A),
              prove(B).
prove(H) :- H <- B,
            prove(B).
```

**ESERCIZI.**
Aggiungere la or nel corpo;
Aggiungere la and nella testa.



## Vediamo alcuni usi

- Si tratta di modificare vanilla per
  - Modificare la strategia di ricerca
    - Vedremo l'aggiunta di una profondità massima di ricerca;
    - Il testo discute anche il modo di modificare la strategia per applicare le note strategie di ricerca
  - Ritardare i goal
    - Consente l'abduzione
  - Interagire con l'utente
    - Gestire il modo di interazione
  - Spiegare le risposte
    - Costruire gli alberi di prova
  - Debugging
    - inconsistenze nei dati, dati mancanti, risposte mancanti, individuazione dei loop



## Introdurre la profondità di ricerca

- Potare i rami che eccedono una profondità fissata in chiamata

```
prove(true,_).
prove(A&B,D) :-prove(A,D),
               prove(B,D).
prove(H,D) :-  D >= 0,
               D1 is D-1,
               H <- B,
               prove(B,D1).
```

ESERCIZIO: Dare l'iterative deepening.



## Ritardo dei goals

- Alcuni goal sono dichiarati ritardabili. I goal ritardabili non vengono dimostrati, ma messi in una lista.
- Per trattare efficientemente la lista, si possono usare le differenece lists.
- prove(G,D1-D2) : G conseguenza logica di KB e di D1
- In particolare ci chiederemo prove(G,D-[])



```
prove(true,D-D).
prove(A&B,D1-D3) :-prove(A,D1-D2),
                   prove(B,D2-D3).
prove(H,[H|D]-D) :-  delay(H).
prove(H,D1-D2) :- H <- B,
                  prove(B,D1-D2).
```



## Applicazioni

- Ritardare determinati goal aperti (che possono ad esempio entrare in loop se non ground) sino a quando non diventano ground.
- Fare abduzione; ad esempio, nell'impianto elettrico, ritardiamo le risposte ok(X) e possiamo individuare quali fusibili debbono essere OK perché si accenda una data lampadina:
  - ? accesa(l1, FusibileOk – [])
    FusibileOk = [ok(fus1)].
- Ecc.

# Interazione con l'utente

- Il modo di rispondere ad una domanda (query) da parte del programma e dell'utente sono identici:
  - Possiamo avere risposte SI/NO a domande ground
  - Sostituzioni di risposta a domande non ground.
- Interagendo con un utente, si tratta di:
  - Prevedere il tipo di domande e di risposte
    - SI/NO, funzionali, relazionali
  - Porre le domande in un ordine logico
  - Ricordare quali domande sono già state poste, in modo da non ripetere inutilmente una stessa domanda
    - o una domanda che $D\sigma$ sia un caso particolare di una domanda D già posta



```
prove(true).
prove(A&B) :-prove(A),
             prove(B).
prove(H) :-  askable(H),
             answered(H,[Ans,yes]).
prove(H) :-  askable(H),
             unanswered(H),
             ask(H,Ans),
             record(answered(H,Ans)),
             Ans=[_,yes].
prove(H) :- H <- B,
            prove(B).
```



# Dare spiegazioni

- Si tratta di costruire l'albero di prova, come già visto in un esercizio.
- L'albero di prova spiega come una risposta è stata ottenuta; ciò può essere utile per capire la risposta.
  - Esempio: voto negativo ad uno scritto, oppure voto negativo ad uno scritto con tutti le spiegazioni dei motivi che hanno portato a quel voto
  - Dall'albero di prova si possono estrarre informazioni in chiaro:
    - COME
    - PERCHE'
- E' possibile coniugare abduzione e spiegazione consentendo alberi di prova con assunzioni non dimostrate (ritardate).

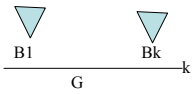


```
how(true,true).
how(A&B,P1&P2) :-how(A,P1),
                 how(B,P2).
how(H,if(H,T)) :- H <- B,
                  how(B,T).
```



# Uso di how nel debugging: risultato errato

- Debug(G) G falso ma dimostrato da KB:
  - Considera l'albero




Se k non è vera nell'interpretazione intesa, trovato errore.
Se qualche Bi non è vero nella interpretazione intesa, allora debug(Bi).



# Debugging: regole mancanti

- G fallisce e dovrebbe avere successo
- Base: se nessuna testa unifica con G, considera i casi seguenti:
  - Una delle teste dovrebbe unificare ma non unifica: correggi la clausola relativa
  - Mancano delle regole: aggiungile
- Passo: almeno una clausola unifica. Considera il corpo di tale clausola, avendo applicato l'unificatore:
  - Vi è un atomo del corpo che ha successo e dovrebbe fallire; procedi come nel caso di risultato errato
  - Qualche atomo fallisce e non dovrebbe: procedi ricorsivamente;
  - Mancano delle regole: aggiungile

# Debugging: Loop

- Indecidibile (halting macchine Turing)
- Comunque uno ci prova.
- Trovare alberi in cui un goal si ripete.
  - eventualmente procedendo per profondità limitate
- Un modo per garantire la terminazione è quello di fornire una misura intera tale che gli atomi dell'albero di prova a livelli più alti abbiano misura minore di quelli sottostanti
  - La misura può prevedere ad esempio termini ground; vedere se vi sono nell'albero nodi che non rendono ground termini richiesti tali
  - … altri motivi



# Altro approccio: Trasformazione

- Anziché usare un meta-interprete, risolvere in Prolog e poi trasformare il programma.



**KKB. Regole generali.**
Le nostre conoscenze generali sono codificate da questo piccolo "sistema esperto", che chiameremo

**elettricista:**

```
accesa(X) :-     lampadina(X), connesso(X,F), tensione(F).
corrente(X) :-   presa(X), connesso(X,F), tensione(F).

tensione(F) :-   filo(F), valvola(V), connesso(F,V), ok(V).
tensione(F) :-   filo(F), filo(G), contatto(F,G), tensione(G).

contatto(F,G) :- filo(F), filo(G), da(G,F).
contatto(F,G) :- filo(F), filo(G), interruttore(I), collega(I,G,F).
```



Idea della trasformazione: aggiungere le spiegazioni al programma

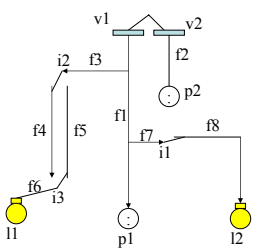
```
accesa(X, perche(connesso(X,F,SC),tensione(F,ST))) :-
        lampadina(X), connesso(X,F,SC), tensione(F,ST).
```



- Terminazione
  contatto(F,G): il filo F è in contatto con il filo G e l'alimentazione proviene da G
  ***ORIENTAMENTO: G è più vicino ad una valvola di F***

- Rappresentazione di un particolare impianto.
  **Bisogna rispettare la condizione di orientamento.**






**impianto:**

```
lampadina(l1).
connesso(l1,f6).
lampadina(l2).
connesso(l2,f8).
valvola(v1).
valvola(v2).
presa(p1).
connesso(p1,f1).
presa(p2).
connesso(p2,f2).

filo(f1).
connesso(f1,v1).
filo(f2).
connesso(f2,v2).
filo(f3).
da(f1,f3).
filo(f4).
filo(f5).
filo(f6).
filo(f7).
da(f1,f7).
filo(f8).

interruttore(i1).
collega(i1,f7,f8) :- on(i1).
interruttore(i2).
collega(i2,f3,f4) :- sw1(i2).
collega(i2,f3,f5) :- sw2(i2).
interruttore(i3).
collega(i3,f4,f6) :- sw1(i3).
collega(i3,f5,f6) :- sw2(i3).
```

- L'impianto può avere diversi stati

**stato1:**

ok(v1).
ok(v2).
on(i1).
sw1(i2).
sw2(i3).



- Se si sbaglia a dare la base dati dell'impianto, il programma può dare risultati errati o entrare in loop.
- Come fare per trovare l'errore?
  - Porre un limite superiore alla profondità delle spiegazioni ed esaminare le connessioni generate; se si incontra un ciclo, si trova l'errore; oppure si esamina la vicinanza nel circuito e si vede se la condizione di orientamento è rispettata, passo passo



# Repetita iuvant

- ***Nell'esempio abbiamo 3 livelli di conoscenza:***
  - **generale:** regole di ragionamento valide in un'intera classe di modelli, che corrispondono i ***mondi*** del dominio di problema; rappresentano la conoscenza di un "esperto";
    - KB **elettricista**
    - **correttezza:** Le regole sono vere in tutti i mondi?
    - **copertura:** abbiamo dimenticato qualche ragionamento che porta un elettricista a risolvere i problemi in esame?



  - **particolare statica:** rappresenta le proprietà statiche rilevanti di un particolare modello, corrispondente ad un particolare mondo:
    - KB **impianto**
    - **correttezza:** le proprietà rappresentate sono vere nel modello (mondo) considerato?
    - **copertura:** abbiamo dimenticato qualche proprietà statica che è necessario conoscere per applicare le regole generali?
  - **particolare contingente:** ha le stesse caratteristiche di quella statica, ma rappresenta proprietà dinamiche
    - KB **stato1, stato2, ….**