PROLOG E SISTEMI ESPERTI

- Prolog può essere utilizzato in almeno due modi differenti nella costruzione di sistemi esperti:
  - come semplice linguaggio di realizzazione di un sistema basato su conoscenza.
  - sfruttando le caratteristiche del Prolog e definendo e costruendo dei sistemi integrati nel Prolog.
- Il Prolog può essere visto come un sistema a regole di produzione in cui:
  - clausole (regole) Prolog = regole di produzione;
  - asserzioni Prolog = fatti;
  - data-base Prolog = memoria di lungo termine + memoria di lavoro;
  - interprete Prolog = motore inferenziale (basato su una strategia backward, con ricerca in profondità con backtracking).

USO DIRETTO DEL PROLOG

- USO DIRETTO DEL PROLOG NELLA COSTRUZIONE DI SISTEMI ESPERTI A REGOLE
- Un sistema a regole di produzione quale Prolog è solo il nucleo di un sistema esperto.
- Per ottenere tutte le funzionalità di un sistema esperto (quali la possibilità di effettuare ragionamento approssimato, la capacità di interazione con l'utente, la capacità di spiegazione) è necessario estendere tale nucleo di base.

POSSIBILE FORMATO DELLE REGOLE

If precondizione P then conclusione C

If situazione S then azione A

If condizioni C1 and C2 sono vere

then la condizione C è falsa.

- Le regole sono una forma abbastanza naturale per esprimere la conoscenza e godono delle seguenti proprietà:
  - Modularità;
  - Incrementalità;
  - Modificabilità;
  - Trasparenza, cioè capacità di spiegare il proprio comportamento:

    "How", ovvero come sei arrivato a questa conclusione?

    "Why", ovvero perchè sei interessato a questa informazione?

ESEMPIO

- Si consideri una semplice regola di produzione (parte della base di conoscenza di un sistema per la diagnosi di guasti a una automobile):

if luci spente and

motorino di avviamento muto

then probabile (0.8) guasto alla batteria

- Il formato della regola è quello del sistema MYCIN. Il valore 0.8 associato alla regola rappresenta il “grado di certezza” con cui può essere raggiunto il conseguente se l’antecedente della regola è vero.

ESEMPIO

- La regola può essere tradotta in una clausola Prolog:

```
batteria(guasta)  :-
        luci(spente),
        motorino_avviamento(muto).
```

- Il passo successivo è quello di definire dei meccanismi per il **ragionamento approssimato**.

- La verità di un fatto non viene più valutata utilizzando una logica a due valori {vero, falso}, ma piuttosto utilizzando una logica a più valori (anche infiniti).

RAGIONAMENTO APPROSSIMATO (1)

- A ogni fatto F viene associato un **grado di evidenza** rappresentato mediante un numero reale nell'intervallo [0,1]:
  - se il grado di evidenza di F è 0, allora F è falso;
  - se il grado di evidenza di F è 1, allora F è vero;
  - valori intermedi corrispondono a casi di incertezza sulla falsità o verità di F.
- Grado di evidenza **EV** di un fatto **F**

```
F with EV
```

(in cui **`with`** è un operatore).

RAGIONAMENTO APPROSSIMATO (2)

- Relazioni per trattare conoscenza approssimata:

  `valuta_antecedente(LISTA_EV, EV)`

- data la lista **`LISTA_EV`** dei gradi di evidenza delle condizioni nell'antecedente di una regola, **`EV`** è il grado di evidenza globale dell'antecedente della regola

  `valuta_conseguente(EV_ANT,CF,EV_CONS)`

- data una regola con grado di certezza **`CF`** e il grado di evidenza **`EV_ANT`** dell'antecedente, allora **`EV_CONS`** è il grado di evidenza del conseguente

ESEMPIO

- La regola può allora essere rappresentata mediante la seguente clausola Prolog:

```
batteria(guasta)  with EV :-
        luci(spente) with EV1,
        motorino_avviamento(muto) with EV2,
        valuta_antecedente([EV1,EV2],EV_ANT),
        valuta_conseguente(EV_ANT,0.8,EV).
```

- Può essere significativo ricercare tutte le soluzioni per un dato goal.

MECCANISMI DI SPIEGAZIONE

- Una semplice spiegazione può essere ottenuta mediante argomenti aggiuntivi nelle asserzioni e nelle clausole.
- Più in particolare:
  - Ogni fatto **F** viene rappresentato mediante una asserzione del tipo
  - **`F with EV explain EXPL`**.
  - dove **`explain`** e **`with`** sono operatori e **`EXPL`** è la spiegazione associata al fatto **F**.
- Nella clausola corrispondente a una regola viene aggiunta una relazione per sintetizzare la spiegazione.

MECCANISMI DI SPIEGAZIONE

- Ad esempio, la regola precedente può essere rappresentata mediante la clausola:

```
batteria(guasta) with EV explain
  dimostrato(batteria(guasta) with EV,
  a_partire_da([EXPL1,EXPL2]))  :-

   luci(spente) with EV1 explain EXPL1,

   motorino_avviamento(muto) with EV2 explain EXPL2,

   valuta_antecedente([EV1,EV2],EV_ANT),

   valuta_conseguente(EV_ANT,0.8,EV).
```

PROLOG E SISTEMI ESPERTI

- **Vantaggi:**
  - possibilità di sfruttare a fondo le caratteristiche del Prolog;
  - efficienza;
  - facilità di realizzazione.

- **Svantaggi:**
  - approccio limitato all'uso di regole di produzione con strategia di inferenza backward;
  - scarsa leggibilità e modificabilità dei programmi.

META-INTERPRETAZIONE

- Il Prolog può essere utilizzato per definire linguaggi di rappresentazione della conoscenza e per la definizione di interpreti (motori inferenziali) per tali linguaggi.
- Si supponga che le regole di produzione siano rappresentate mediante asserzioni Prolog del tipo:

  `CONSEG cert_fact CF if ANTEC`

  in cui **`cert_fact`** e **`if`** sono operatori, **`CONSEG`** è un termine Prolog e **`ANTEC`** è una congiunzione di termini.
- Ogni fatto nella memoria di lavoro sia rappresentato mediante una asserzione del tipo:

  `F with EV`

ESEMPIO

- Definire un meta-interprete che realizza un motore inferenziale **backward** su tali regole. **`solve(GOAL,EVID)`**
- "GOAL può essere dimostrato con evidenza EVID utilizzando le regole contenute nel data base"

```
solve(true,1).
solve((A,B), EVID)  :-
        solve(A,EV1),
        solve(B,EV2),
        valuta_antecedente([EV1,EV2],EVID).
solve(A,EV)  :-
        A with EV.
solve(A,EV)  :-
        A cert_fact CF if B,
        solve(B,EVID_ANTEC),
        valuta_conseguente(EV_ANTEC,CF,EV).
```

- Può essere facilmente esteso per fornire spiegazioni e per interagire con l'utente. Non ci si deve limitare alla strategia di ragionamento backward.

ESEMPIO

- Meta-interprete che utilizza una strategia di inferenza **forward**.
- Consideriamo, in primo luogo, il caso di un interprete di base per regole rappresentate come asserzioni Prolog del tipo: **`CONSEG if ANTEC`**
- Un interprete forward per le regole è definito dal seguente programma:

```
interpreta:-  <verifica se si è raggiunto un obiettivo>
interpreta:-     CONSEG if ANTEC,
                 verifica_antec(ANTEC),
                 not(CONSEG),
                 assert(CONSEG),
                 interpreta.
interpreta  :-  <riporta fallimento della dimostrazione>.
verifica_antec(ANTEC) "la congiunzione ANTEC è
  soddisfatta"
verifica_antec((A,B)):-A,
verifica_antec(B).
verifica_antec(A) :- A.
```

REGOLE APPLICABILI

- Affinché una regola sia applicabile devono essere soddisfatte due condizioni:
  - L'antecedente della regola deve essere soddisfatto.
  - l conseguente della regola non deve essere già vero. Ciò permette di evitare che una regola venga applicata più volte sugli stessi dati e che l'interprete vada in ciclo.
- L'interprete non ha una vera e propria fase di risoluzione di conflitti: viene semplicemente attivata la prima regola applicabile.
- Non è difficile realizzare un meta-interprete in cui le fasi di MATCH e CONFLICT-RESOLUTION sono separate.
- Supponiamo che le regole siano rappresentate da asserzioni del tipo:

  `regola(NOME, CONSEG if ANTEC)`

  in cui **`NOME`** è un nome che identifica univocamente ogni regola.

INTERPRETE FORWARD: risoluzione di conflitti esplicita

```
interpreta1  :- <verifica se si è raggiunto un obiettivo>
interpreta1  :-
       match(REG_APPLICABILI),
       conflict_res(REG_APPLICABILI,REGOLA),
       applica(REGOLA),
       interpreta1.
interpreta1 :- <riporta fallimento della dimostrazione>.

match(REG_APPLICABILI)
"REG_APPLICABILI: l'insieme di regole  applicabili dato il
   contenuto della memoria di lavoro (data base)"
match(REG_APPLICABILI)  :-
   setof([REG,CONSEG],applic(REG,CONSEG),REG_APPLICABILI).
```

INTERPRETE FORWARD: risoluzione di conflitti esplicita

applic (REG,CONSEG)

"la regola REG con conseguente CONSEG è applicabile"

```
applic(REG,CONSEG)  :-
        regola(REG, CONSEG if ANTEC),
        verifica_antec(ANTEC),
        not(CONSEG).
```

conflict_res(REG_APPLIC,[REG,CONSEG])

"la regola **REG** con conseguente **CONSEG** è la regola selezionata all'interno della lista **REG_APPLIC** di regole"

```
conflict_res(REG_APPLIC,[REG,CONSEG]):-
        <selezione della regola da applicare>
        applica([REG,CONSEG])
```

"applicazione regola **REG** con conseguente **CONSEG**"

```
applica([REG,CONSEG]):- assert(CONSEG).
```


MECCANISMO DI SPIEGAZIONE

- È facile aggiungere un meccanismo di spiegazione.
- È sufficiente modificare la definizione della relazione "applica" nel modo seguente:

```
applica([REG,CONSEG]):-
            assert(CONSEG),
            assert(dimostrato(CONSEG,REG)).
```

- Una spiegazione può quindi essere fornita mediante il predicato:

```
spiega(GOAL)  :-
            dimostrato(GOAL,REGOLA), !,
            regola(REGOLA, GOAL if ANTEC),
            write('dimostrato '), write(GOAL),
            write('utilizzando la regola '),
            write(GOAL if ANTEC).
spiega(GOAL)  :-
            write('fatto '), write(GOAL).
```

COSTRUZIONE DI SISTEMI ESPERTI CON META-INTERPRETAZIONE

- **Vantaggi**
  - elevata flessibilità;
  - facilità di realizzazione dei meta-interpreti;
  - leggibilità e modificabilità (almeno per i meta-interpreti semplici);
  - portabilità;
  - possibilità di definire meta-interpreti per diversi linguaggi di rappresentazione della conoscenza e diverse strategie di controllo.
- **Svantaggi**
  - i meta-interpreti possono diventare difficili da mantenere se il linguaggio di rappresentazione e le strategie di controllo diventano molto complesse;
  - elevata inefficienza dovuta alla sovrapposizione di uno o più livelli di interpretazione al di sopra di quello del Prolog;
  - Per ovviare al problema di inefficienza è stato proposto di utilizzare tecniche di **valutazione parziale**.

ALTRI ESEMPI

- Presi da: I. Bratko: Programmare in Prolog per lIntelligenza Artificiale, Masson ed Addison-Weslay, 1988.
- **Base di conoscenza per identificare gli animali (problema di classificazione)**

```
:- op(100,xfx,[has,gives,'does not', eats,lays,isa]).
:- op(100,xf,[swims,flies]).
:- op(900,xfx,:).
:- op(800,xfx,was).
:- op(870,fx,if).
:- op(880,xfx,then).
:- op(550,xfy,or).
:- op(540,xfy,and).
:- op(300,fx,'derived by').
:- op(600,xfx,from).
:- op(600,xfx,by).
```

ALTRI ESEMPI

```
rule1:  if
    Animal has hair     or
    Animal gives milk   then
    Animal isa mammal.
rule2:  if
    Animal has feathers or
    Animal flies        and
    Animal lays eggs    then
    Animal isa bird.
rule3:  if
    Animal isa mammal           and
    (Animal eats meat           or
    Animal has pointed teeth    and
    Animal has claws            and
    Animal has 'forward pointing 'eyes') then
    Animal isa carnivore.
```


ALTRI ESEMPI

```
rule4:  if
    Animal isa carnivore        and
    Animal has 'tawny colour'   and
    Animal has 'dark spots'     then
    Animal isa cheetach.
rule5:  if
    Animal isa carnivore        and
    Animal has 'tawny colour'   and
    Animal has 'black stripes'  then
    Animal isa tiger.
rule6:  if
    Animal isa bird             and
    Animal 'does not' fly       and
    Animal swims                then
    Animal isa pinguin.
```

ALTRI ESEMPI

```
rule7:  if
    Animal isa bird              and
    Animal isa 'good flyer'                then
    Animal isa albatross.
fact: X isa animal:-
member(X,[cheetah,tiger,penguin,albatross]).
askable(_ gives_, 'Animal' gives 'What').
askable(_ flies, 'Animal' flies).
askable(_ lays eggs, 'Animal' lays eggs).
askable(_ eats_, 'Animal' eats 'What').
askable(_ has_, 'Animal' has 'Something').
askable(_ 'does not'_, 'Animal' 'does not' 'Do something').
askable(_ swims, 'Animal' swims).
askable(_ isa 'good flier', 'Animal' isa 'good flier').
```

Nota: sono tutti fatti Prolog dal punto di vista sintattico.

UN SECONDO ESEMPIO

Base di conoscenza per individuare guasti in una rete elettrica (diagnosi).

```
broken_rule: if
    on(Device)                  and
    device(Device)              and
    not working(Device)         and
    connected(Device,Fuse)      and
proved (intact(Fuse))
then
proved(broken(Device)).
fuse_ok_rule: if
    connected(Device,Fuse)      and
    working(Device)
then
proved(intact(Fuse)).
```

UN SECONDO ESEMPIO

```
fused_rule:
if connected(Device1,Fuse)          and
   on(Device1)                      and
   device(Device1)                  and
   not working(Device1)             and
   samefuse(Device2,Device1)        and
   on(Device2)                      and
   not working(Device2)
then
proved(failed(Fuse)).
```

se due differenti dispositivi sono connessi ad un fusibile e sono entrambi on ma non in funzione allora il fusibile è rotto (si assume che non possano essere rotti entrambi i dispositivi).

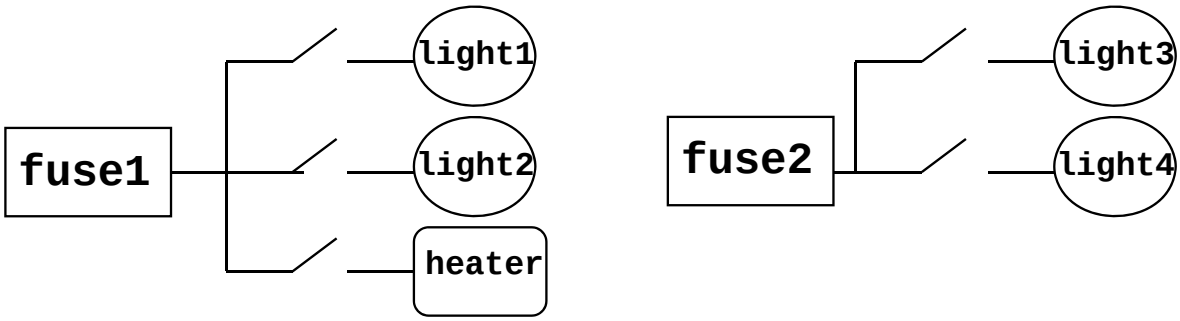
UN SECONDO ESEMPIO

```
same_fuse_rule: if
   connected(Device1,Fuse) and
   connected(Device2,Fuse) and
   different(Device1,Device2)
then
somefuse(Device1,Device2).
fact: different(X,Y) :- not (X=Y).
fact: device(heater).
fact: device(light1).
fact: device(light2).
fact: device(light3).
fact: device(light4).
fact: connected(light1,fuse1).
fact: connected(light2,fuse1).
fact: connected(heater,fuse1).
```

UN SECONDO ESEMPIO

```
fact: connected(light3,fuse2).
fact: connected(light4,fuse2).
askable(on(D), on('Device')).
askable(working(D),working('Device')).
```

- *Nota: sono tutti fatti Prolog.*



USO DEL PROLOG

- Per utilizzare direttamente Prolog andrebbero traslati come regole del tipo:

```
Animal isa mammal:-
          Animal has hair;
          Animal gives milk.
Animal isa carnivore:-
          Animal isa mammal,
          Animal eats meat.
```

- Adesso aggiungiamo dei fatti sul particolare problema:

```
peter has hair.
peter is lazy.
peter is big.
```

USO DEL PROLOG

```
peter has 'tawny colour'.
peter has 'black stripes'.
peter eats meat.
```

- e poi interroghiamo:

```
?- peter isa tiger.
yes
? peter isa cheetah.
no.
```

- Non ci va bene per due motivi:
  1) I fatti devono essere introdotti tutti all'inizio;
  2) Manca una spiegazione (il tracing di Prolog è troppo povero).
  ==> approccio meta-interpretato

MOTORE DI INFERENZA IN PROLOG

- Per dare una risposta Answ a una domanda Q (simile alla ricerca in grafi AND/OR):

  Se **Q** è un fatto allora Answ è: **Q is true**;

  Se c'è una regola del tipo:

  if Condition then Q allora esplora **Condition** per generare la risposta Answ;

  Se **Q** è askable allora chiedi all'utente per avere una risposta per **Q**;

  Se **Q** è della forma **Q1 and Q2** allora esplora **Q1**. Se **Q1** è falso allora Answ è "**Q** e falso", altrimenti esplora **Q2** e poi combina le risposte di **Q1** e **Q2** in Answ;

  Se **Q** è della forma **Q1 or Q2** allora esplora **Q1**. Se **Q1** è vero allora Answ è "**Q** è vero", o alternativamente esplora **Q2** e poi combina le risposte di **Q1** e **Q2** in Answ.

MOTORE DI INFERENZA IN PROLOG

- Le domande del tipo **not Q** sono più problematiche e le tratteremo nel seguito.

  Interfaccia con l'utente: why e how.

- La domanda **why** può essere generata dall'utente quando il sistema chiede all'utente qualche informazione e l'utente vuole sapere perchè gli viene chiesta tale informazione.

```
Is a true?
why?
```

- Because:

```
I can use a to investigate b by rule Ra, and
I can use b to investigate c by rule Rb, and ........
I can use y to investigate z by rule Ry, and
z was your original question.
```

- Catena di regole e (sotto)goals che connettono l'informazione richiesta con il goal originale (traccia). Why è ottenuto muovendosi in su nello spazio di ricerca dal corrente (sotto)goal al top goal.

MOTORE DI INFERENZA IN PROLOG

- Dunque la traccia (catena dei goals e regole fra il goal corrente e il top goal) deve essere mantenuta esplicitamente durante il processo di ragionamento.
- Inoltre, quando l'utente ottiene una risposta può avere interesse a sapere come questa risposta è stata ottenuta.
- How fa vedere i goal e sottogoal che dimostrano la conclusione, cioè, in pratica, l'albero AND/OR di soluzione.
- Esempio:

```
peter isa carnivore
was derived by rule3 from
peter isa mammal
was derived by rule1 from
peter has hair
was told
and
peter eats meat
was told
```

IMPLEMENTAZIONE

- Procedure principali:

  `explore(Goal, Trace, Answer)`

  che trova una risposta **`Answer`** a un goal **`Goal`** con la traccia **`Trace`**.

  `useranswer(Goal,Trace,Answer)`

  che genera la soluzione per un “askable” **`Goal`** chiedendola all'utente e risponde anche a domande di tipo 'why'.

  `present(Answer)`

  mostra i risultati e risponde a domande di tipo 'how'.

  `explore(Goal,Trace,Answer)`

  “trova una risposta **`Answer`** a un dato goal **`Goal`**. Cerca una soluzione positiva. **`Answer`** falso solo quando sono state tentate con insuccesso tutte le possibilità. Nota: si suppone che ci sia solo una regola applicabile per ogni tipo di goal; **`Goal`** negativi devono sempre essere istanziati”

IMPLEMENTAZIONE

```
:- op(900,xfx,:).
:- op(800,xfx,was).
:- op(870,fx,if).
:- op(880,xfx,then).
:- op(550,xfy,or).
:- op(540,xfy,and).
:- op(300,fx,'derived by').
:- op(600,xfx,from).
:- op(600,xfx,by).
explore(Goal,Trace,Goal is true was 'found
as a fact'):- fact : Goal.
```

IMPLEMENTAZIONE

```
explore(Goal,Trace,Goal is true was 'found  as a fact'):-
        fact : Goal.
explore(Goal,Trace,Goal is TruthValue was 'derived by' Rule from
   Answer):-
        Rule : if Condition then Goal,
        explore(Condition,[Goal by Rule|Trace],Answer),
        truth(Answer, TruthValue).
explore(not Goal, trace, Answer) :- !,
        explore(Goal,Trace,Answer1),
        invert(Answer1,Answer).
explore(Goal1 and Goal2, Trace, Answer):-!,
        explore(Goal1, Trace,Answer1),
        continue(Answer1, Goal1 and Goal2, Trace, Answer).
```


IMPLEMENTAZIONE

```
explore(Goal1 and Goal2, Trace, Answer):-!,
        explore(Goal1, Trace,Answer1),
        continue(Answer1, Goal1 and Goal2, Trace, Answer).
explore(Goal1 or Goal2, Trace, Answer):-
        exploreyes(Goal1, Trace,Answer);
        exploreyes(Goal2, Trace,Answer).
explore(Goal1 or Goal2, Trace, Answer1 and Answer2):-!,
        not exploreyes(Goal1, Trace,_);
        not exploreyes(Goal2, Trace,_),
        explore(Goal1, Trace,Answer1);
        explore(Goal2, Trace,Answer2).
explore(Goal,Trace,Goal is Answer was told):-
        useranswer(Goal,Trace,answer).
```

IMPLEMENTAZIONE

```
exploreyes(Goal,Trace,Answer):-
       explore(Goal,Trace,Answer),
       positive(Answer).
continue(Answer1, Goal1 and Goal2, Trace, Answer):-
       positive(Answer1),
       explore(Goal2,Trace,Answer2),
       (positive(Answer2),
        Answer=Answer1 and Answer2;
        negative(Answer2), Answer=Answer2).
continue(Answer1, Goal1 and Goal2, _, Answer1):-
       negative(Answer1).
truth(Question is TruthValue was Found, TruthValue) :- !.
```


IMPLEMENTAZIONE

```
truth(Answer1 and Answer2, TruthValue):-
       truth(Answer1,true),
       truth(Answer2,true),!,
       TruthValue = true;
       TruthValue = false.
positive(Answer) :-
       truth(Answer,true).
negative(Answer) :-
       truth(Answer,false).
       invert(Goal is true was Found,
              (not Goal) is false was Found).
invert(Goal is false was Found,
              (not Goal) is true was Found).
```

ESERCITAZIONE PROPOSTA: Terminare l'intero shell

- Definire le procedure:

`useranswer(Goal,Trace,Answer)`

- Tenendo conto del fatto che:
  - deve fare la domanda una sola volta per lo stesso goal controllando che sia askable;
  - fare il trace se richiesto dall'utente mediante why;
  - se goal contiene delle variabili farle istanziare dall'utente.

`present(Answer)`

per mostrare la soluzione e eventualmente la spiegazione how

`expert`

Il goal top-level dello shell che chiede il goal e poi mostra la risposta;