TERMINI, OPERATORI e METALIVELLO

- In Prolog tutto è un termine: variabile, costante (numerica o atomica), struttura (lista come caso particolare con funtore ".")

- Ciascun termine struttura ha un funtore e argomenti che sono termini

- L'espressione 2+3 è un termine: +(2,3)
- Ma anche ogni clausola, ad esempio:
- member(X,[X|_]):-!. è un termine

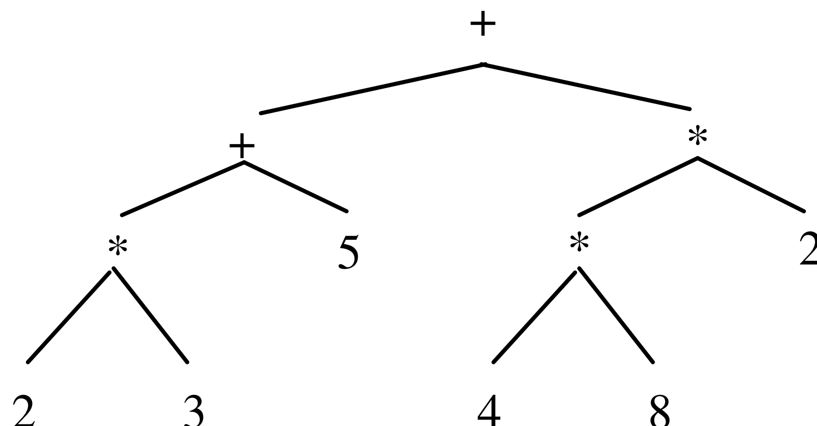
OPERATORI

- In Prolog è possibile definire "operatori" ed assegnare agli operatori regole di associatività e precedenza.

- Quando ci troviamo ad analizzare un'espressione del tipo:

2*3+5+4*8*2

siamo in grado di interpretare univocamente tale stringa



Associatività

- Si consideri l'espressione: 5-2-2, ci sono due possibili interpretazioni:

  (a) (5-2)-2

  (b) 5-(2-2)

- Per risolvere tale ambiguità è necessario specificare la regola di "associatività" dell'operatore.
-
- Nel caso degli operatori aritmetici "+", "*", "-" e "/" si assume per convenzione che gli operatori siano associativi a sinistra, ossia si privilegia la lettura (a) dell'espressione.

OPERATORE, CARATTERIZZATO DA:

- **nome**;
- **numero di argomenti**;
- **priorità** (o precedenza rispetto agli altri operatori);
- **associatività**; in particolare, un operatore può essere non associativo (come ad esempio l'operatore "=") o associativo; nel secondo caso l'operatore può essere associativo a destra o a sinistra.

DEFINIZIONEdi un OPERATORE

```
?- op(PRIORITA, TIPO, NOME)
```

- **`NOME`**, atomo alfanumerico con primo carattere alfabetico oppure una lista di tali atomi; nel secondo caso tutti gli operatori della lista vengono definiti con lo stesso tipo e priorità;
- **`PRIORITA`** è un numero (generalmente tra 0 e 1200) e specifica la priorità dell'operatore;
- **`TIPO`**, indica numero degli argomenti e associatività dell'operatore:
  - **`fx, fy`** (per operatori unari prefissi)
  - **`xf, yf`** (per operatori unari postfissi)
  - **`xfx, yfx, xfy`** (per operatori binari)

TIPO di un OPERATORE (BINARIO)

- **`xfx`** operatore non associativo
- **`yfx`** operatore associativo a sinistra

per cui, se “newop" ha tipo "yfx", l'espressione

E1 newop E2 newop E3 ... newop En

viene interpretata come

(...(E1 newop E2) newop E3) ....) newop En)

ossia come il termine

newop(newop(...newop(E1,E2), E3,...), En)

- **`xfy`** operatore associativo a destra

E1 newop (E2 newop (... newop (En-1 newop En)..))

newop(E1, newop(E2,newop(... newop (En-1,En)..))

OPERATORI PREDEFINITI

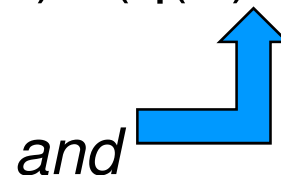
```
?-  op(1200,   xfx,      [:-]).
?-  op(1200,     fx,     [ :-, ?-]).
?-  op(1100,   xfy,      [ ;]).
?-  op(1000,   xfy,      [',']).
?-  op(  900,     fy,     [not]).
?-  op(  700,    xfx, [=, is, =.., ==, \==, =:=,
      =\=, <, >, =<, >=]).
?-  op(  500,    yfx,     [+, -]).
?-  op(  500,      fx,    [+, -]).
?-  op(  400,    yfx,     [*,  /]).
?-  op(  300,    xfx,     [mod]).
```

TERMINI e CLAUSOLE

- Anche i connettivi logici ",", ";" e ":-" sono definiti come operatori Prolog. In Prolog non esiste, infatti, alcuna distinzione tra dati e programmi per cui anche le congiunzioni e le clausole sono termini.

- member(X,[X|_]):-!. :-(member(X,[X|_]), !).

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



and

ESERCIZIO 7.1

- Introdurre un costrutto di iterazione di tipo "while":

  `while C do S`

- con il seguente significato informale: "fin tanto che la condizione `C` è vera, invoca la procedura `S` "
- Supponiamo che S possa essere una qualunque condizione, anche un ulteriore costrutto di iterazione; si considera quindi come legale una espressione del tipo: `while C1 do while C2 do S`

  imponendo l'interpretazione

  `while C1 do (while C2 do S)`

- Gli operatori `while` e `do` possono essere definiti come operatori Prolog.

SOLUZIONE ES. 7.1

- **`while`** operatore unario prefisso a priorità minore della priorità di **`do`** definito come un operatore binario associativo a destra.

```
?-  op(200, fy, while).
?-  op(300, xfy, do).
```

- Si ha allora che una espressione del tipo

  while c1 do while c2 do s

  viene interpretata secondo la seguente struttura di parentesi (while c1) do ((while c2) do s)

  ossia come il termine

  do((while c1), do((while c2), s))

SOLUZIONE ES. 7.1

- Insieme di regole Prolog per tali operatori:

```
while C do S   :-  C,
                  !,
                  S,
                  while C do S.
while C do S.
```

Verifica del “tipo” di un termine

- Determinare, dato un termine T, se T è un atomo, una variabile o una struttura composta.
  - atom(T) "T è un atomo non numerico"
  - number(T) "T è un numero (intero o reale)"
  - integer(T) "T è un numero intero"
  - atomic(T) "T è un'atomo oppure un numero (ossia T non è una struttura composta)"
  - var(T) "T è una variabile non istanziata"
  - nonvar(T) "T non è una variabile“
  - compound(T) “T e’ un termine composto”
- E’ anche possibile accedere alle componenti di un termine …

Accesso alle componenti di un termine

functor(TERM, FUNCTOR, ARITY)

- Determina il funtore principale **FUNCTOR** e il numero di argomenti **ARITY** di un termine **TERM**

```
?-   functor(f(a,b,c),f,3).
yes
?-   functor(f(a,b,g(X)),F,A).
yes       F=f   A=3
?-   functor(T,f,2).
yes       T=f(_1,_2)
?-   functor(a,F,A).
yes       F=a        A=0
```

Usi diversi possibili in base a quali argomenti sono istanziati e quali variabili

Accesso alle componenti di un termine

arg(POS, TERM, ARG)

- Determina (unifica) l'argomento **ARG** con quello in posizione **POS** di un termine **TERM**
- Il primo argomento **POS** deve sempre essere istanziato ad una espressione aritmetica al momento della valutazione.

```
?-  arg(1,f(a,b),A).
yes A=a
?-  arg(1+2*3,p(a,b,c,d,e,f,g,h,i,j),A).
yes     A=g
?-  arg(1,f(g(X),b),A).
yes     A=g(_1)
```

Accesso alle componenti di un termine

arg(POS, TERM, ARG)

```
?-   arg(2,p(a,Y),b).
yes       Y=b
?-   arg(1+1,p(a,g(X)),g(b)).
yes       X=b
?-   arg(X,p(a,b),a).
Error in arithmetic expression
```

Lista delle componenti di un termine

```
TERM =.. [FUNCTOR, ARG1, .., ARGn]
TERM =.. [FUNCTOR |[ ARG1, .., ARGn]]
```

```
?-  f(a,b)  =..  [f,a,b].
yes
?-  a  =..  L
yes        L=[a]
?-  f(h(a),b) =.. [FUNCTOR | ARGLIST].
yes        FUNCTOR=f          ARGLIST=[h(a),b]
?- T =.. [g,1,X,h(a)].
yes        T=g(1,_1,h(a))
?-  T =.. [f | [1,2,3]].
yes        T=f(1,2,3).
```

Uso bidirezionale di `=..`
Se `TERM` istanziato e lista variabile, o viceversa

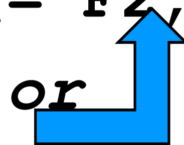
ESERCIZIO 7.2 - MSG

- Dati due termini T1 e T2, determinare la loro generalizzazione più specifica (**MSG, Most Specific Generalization**), ossia il termine più specifico di cui sia T1 sia T2 sono istanze.
- Ad esempio:

T1	T2	MSG
X	X	X
X	Y	Z
a	X	X
a	b	X
f(X)	g(Z)	Y
f(X,a)	f(b,Y)	f(X,Y)

SOLUZIONE ES 7.2 - MSG

```
msg(T1,T2,T1)   :- var(T1), var(T2), T1==T2,!.
msg(T1,T2,_)    :- var(T1),var(T2),!.
msg(T1,T2,T1)   :- var(T1),nonvar(T2),!.
msg(T1,T2,T2)   :- nonvar(T1),var(T2),!.
msg(T1,T2,_)    :- nonvar(T1),
                   nonvar(T2),
                   functor(T1,F1,N1),
                   functor(T2,F2,N2),
                   (F1 =\= F2; N1 =\= N2),!.
```



or

diverso in SICStus Prolog

SOLUZIONE ES 7.2 – MSG (cont.)

```
msg(T1,T2,T3)   :-    nonvar(T1),
                      nonvar(T2),
                      functor(T1,F,N),
                      functor(T2,F,N),
                      T1 =.. [F | ARGS1],
                      T2 =.. [F | ARGS2],
                 msg_list(ARGS1,ARGS2,ARGS),
                      T3 =.. [F | ARGS].
```

SOLUZIONE ES 7.2 – MSG (cont.)

```
msglist(L1,L2,L)
"L è la lista delle generalizzazioni
più specifiche delle coppie di elementi
delle liste L1 e L2 in ugual posizione"
```

```
msg_list([],[],[]).
msg_list([T1|REST1], [T2|REST2], [T|REST]):-
                msg(T1,T2,T),
                msg_list(REST1,REST2,REST).
```

PREDICATI DI META LIVELLO

- In Prolog non vi è alcuna differenza sintattica tra programmi e dati e che essi possono essere usati in modo intercambiabile.

- Vedremo:

  - la possibilità di accedere alle clausole che costituiscono un programma e trattare tali clausole come termini;
  - la possibilità di modificare dinamicamente un programma (il data-base);
  - la meta-interpretazione.

Accesso alle clausole

- Una clausola (o una query) è rappresentata come un termine.
- Le seguenti clausole:

```
h.
h :- b1, b2, ..., bn.
```

e la loro forma equivalente:

```
h :-true.
h :- b1, b2, ..., bn.
```

corrispondono ai termini:

```
:- (h, true)
:- (h, ','(b1, ','(b2, ','( ...','( bn-1,
  bn,) ...)))
```

Accesso alle clausole: `clause`

`clause(HEAD,BODY)`

- “vero se `:-(HEAD,BODY)` è (unificato con) una clausola all'interno del data base“

- Quando valutata, `HEAD` deve essere istanziata ad un termine non numerico, `BODY` può essere o una variabile o un termine che denota il corpo di una clausola.

- Apre un punto di scelta per procedure non-deterministiche (più clausole con testa unificabile con `HEAD`)

Esempio `clause(HEAD,BODY)`

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

```
?- clause(p(1),BODY).
   yes BODY=true

?- clause(p(X),true).
   yes X=1

?- clause(q(X,Y), BODY).
   yes X=_1  Y=a   BODY=p(_1),r(a);
       X=2   Y=_2  BODY=d(_2);
   no

?- clause(HEAD,true).
    Error - invalid key to data-base
```

Modifiche al database: `assert`

`assert(T)`, "la clausola `T` viene aggiunta al data-base"

- Alla valutazione, `T` deve essere istanziato ad un termine che denota una clausola (un atomo o una regola). `T` viene aggiunto nel data-base in una posizione non specificata.
- Ignorato in backtracking (non dichiarativo)
- Due varianti del predicato "assert":

  `asserta(T)`

"la clausola T viene aggiunta all'inizio data-base"

  `assertz(T)`

"la clausola T viene aggiunta al fondo del data-base"

ESEMPI assert

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

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

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

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

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

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

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

Modifiche al database: `retract`

- **`retract(T)`**, "la prima clausola nel data-base unificabile con **`T`** viene rimossa"

- Alla valutazione, **`T`** deve essere istanziato ad un termine che denota una clausola; se più clausole sono unificabili con T è rimossa la prima clausola (con punto di scelta a cui tornare in backtracking in alcune versioni del Prolog).
- Alcune versioni del Prolog forniscono un secondo predicato predefinito: il predicato "abolish" (o "retract_all", a seconda delle implementazioni):

  `abolish(NAME,ARITY)`

ESEMPI retract

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

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

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

```
?-  abolish(a,1).
```

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

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

ESEMPI retract

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

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

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

```
yes X=1;
```

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

```
yes X=2;
```

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

```
yes X=4;
```

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

```
no
```

Problemi di `assert` e `retract`

- Si perde la semantica dichiarativa dei programmi Prolog.

- Si considerino le seguenti query, in un database vuoto.:

```
?-   assert(p(a)), p(a).
?-   p(a), assert(p(a)).
```

- La prima valutazione ha successo, la seconda genera un fallimento.

- L'ordine dei letterali è rilevante nel caso in cui uno dei due letterali sia il predicato predefinito **`assert`**.

Problemi di `assert` e `retract`

- Un altro esempio è dato dai due programmi:

```
a(1).
p(X) :- assert((b(X))), a(X).
```
(P1)

```
a(1).
p(X) :- a(X), assert((b(X))).
```
(P2)

- La valutazione della query `:- p(X).` produce la stessa risposta, ma due modifiche differenti del data-base:
  - in P1 viene aggiunto `b(X).` nel database, ossia $\forall$`X p(X)`
  - in P2 viene aggiunto `b(1).`

Problemi di `assert` e `retract`

- Un ulteriore problema riguarda la quantificazione delle variabili.
  - Le variabili in una clausola nel data-base sono quantificate universalmente mentre le variabili in una query sono quantificate esistenzialmente.

- Si consideri la query: `:- assert((p(X))).`

- Sebbene **X** sia quantificata esistenzialmente, l'effetto della valutazione della query è l'aggiunta al data-base della clausola

  `p(X).`

  ossia della formula ∀**X p(X)**

ESEMPIO: GENERAZIONE DI LEMMI

- Il calcolo dei numeri di Fibonacci risulta estremamente inefficiente.

  fib(N,Y) "Y è il numero di Fibonacci N-esimo"

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

GENERAZIONE DI LEMMI

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

- Oppure:

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

- In questo secondo modo, la stessa soluzione (lo stesso fatto/lemma) non è asserita più volte all'interno del database.

METAINTERPRETI

- Realizzazione di meta-programmi, ossia di programmi che operano su altri programmi.
- Rapida prototipazione di interpreti per linguaggi simbolici (meta-interpreti)
- In Prolog, un meta-interprete per un linguaggio L è, per definizione, un interprete per L scritto nel linguaggio Prolog.
- Discuteremo come possa essere realizzato un semplice meta-interprete per il Prolog (in Prolog).

METAINTERPRETE PER PROLOG PURO

solve(GOAL) "il goal **GOAL** è deducibile dal programma Prolog puro definito da **clause** (ossia contenuto nel data-base)"

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

- Può facilmente essere esteso per trattare i predicati predefiniti del Prolog (almeno alcuni di essi). E` necessario aggiungere una clausola speciale per ognuno di essi prima dell'ultima clausola per "solve".

PROLOG MA CON REGOLA DI CALCOLO RIGHT-MOST

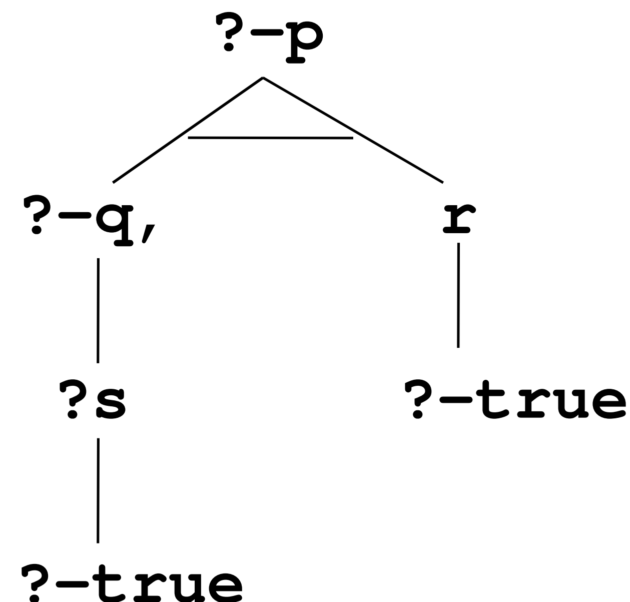
- Il meta-interprete per Prolog puro può essere modificato per adottare una regola di calcolo diversa (ad esempio right-most):

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

ESEMPIO METAINT. CON SPIEGAZIONE

- Si desidera avere, al termine della dimostrazione di un certo goal, una spiegazione della dimostrazione effettuata.
- Un semplice modo per fornire una spiegazione per un goal "g" è quello di stampare l'albero di dimostrazione per "g".
- Esempio:

```
p  :-  q,r.
q  :-  s.
r.
s.
```



ESEMPIO METAINT. (cont.)

- Dato il programma:

```
p  :-  q,r.
q  :-  s.
r.
s.
```

- l'albero di dimostrazione per la query ?-**p.** può essere visualizzato mediante la seguente espressione:

```
p   <-   (q <- (s <- true)),
         (r <- true)
```

- E` sufficiente aggiungere un argomento al predicato "solve" e utilizzare tale argomento per la costruzione della spiegazione.

ESEMPIO METAINT. (cont.)

solve(GOAL,PROVA)

"**PROVA** è un albero di dimostrazione per il goal **GOAL**"

```
?-  op(1200,   xfx, [<-]).
solve(true,true).
solve((A,B),(PROVA1,PROVA2) ) :-
        solve(A, PROVA1),
        solve(B, PROVA2).
solve(A,(A <- PROVA))   :-
        clause(A,B),
        solve(B,PROVA).
```

ESERCIZIO 7.3: METAINTERPRETE

- In un linguaggio simbolico Prolog-like la base di conoscenza è costituita da fatti e regole del tipo:

  `rule(Testa,Body).`

- Si scriva un metainterprete **`solve(Goal,Step)`** per tale linguaggio, in grado verificare se **`Goal`** è dimostrato e, in questo caso, in grado di calcolare in quanti passi di risoluzione (**`Step`**) tale goal viene dimostrato.
- Per le congiunzioni, il numero di passi è dato dalla somma del numero di passi necessari per ciascun singolo congiunto atomico.

ESERCIZIO 7.3 METAINTERPRETE

- Per esempio, per il programma:

```
rule(a,(b,c)).
rule(b,d).
rule(c,true).
rule(d,true).
```

il metainterprete deve dare la seguente risposta:

```
?-solve(a,Step).
yes  Step=4
```

- poiché a è dimostrato applicando 1 regola (1 passo) e la congiunzione (b,c) è dimostrata in 3 passi (2 per b e 1 per c).
- Non si vari la regola di calcolo e la strategia di ricerca di Prolog.

SOLUZIONE ESERCIZIO 7.3 METAINT.

```
solve(true,0):-!.
solve((A,B),S) :- !,solve(A,SA),
                    solve(B,SB),
                    S is SA+SB.
solve(A,S) :-  rule(A,B),
               solve(B,SB),
               S is 1+SB.
```

ESERCIZIO 7.4: FATTORI DI CERTEZZA

- In un linguaggio simbolico Prolog-like la base di conoscenza è costituita da fatti e regole del tipo:

  `rule(Testa,Body,CF).`

- dove **`CF`** rappresenta il fattore di certezza della regola (quanto è vera in termini probabilistici, espressa come intero percentuale – tra 0 e 100).
- **`rule(a, (b,c), 10).`**
- **`rule(b,true, 100).`**
- **`rule(c,true, 50).`**

ESERCIZIO 7.4: METAINTERPRETE

- Si scriva un metainterprete **`solve(Goal,CF)`** per tale linguaggio, in grado verificare se **`Goal`** è dimostrato e con quale probabilità.

- Per le congiunzioni, la probabilità sia calcolata come il minimo delle probabilità con cui sono dimostrati i singoli congiunti.

- Per le regole, è il prodotto della probabilità con cui è dimostrato il corpo per il CF della regola, diviso 100.

ESERCIZIO 7.4: Esempio

```
rule(a, (b,c), 10).
rule(a, d, 90).
rule(b,true, 100).
rule(c,true, 50).
rule(d,true, 100).

?-solve(a,CF).
yes CF=5;
yes CF=90
```

SOLUZIONE ESERCIZIO 7.4 METAINT.

```
solve(true,100):-!.
solve((A,B),CF) :- !,solve(A,CFA),
                     solve(B,CFB),
                     min(CFA,CFB,CF).
solve(A,CFA) :- rule(A,B,CF),
                solve(B,CFB),
                CFA is ((CFB*CF)/100).

min(A,B,A) :- A<B,!.
min(A,B,B).
```