Prolog

INDICE

Introduzione al prolog 1

Sintassi 3

Interpretazione dichiarativa 4

Esecuzione di un programma e Strategia di ricerca del goal 5

Soluzioni multiple e disgiunzione 8

Interpretazione procedurale 8

Aritmetica 9

Operatori aritmetici e predicato “is” 10

Operatori relazionali 11

Verifica del tipo di un termine 13

Calcolo di funzioni 14

Ricorsione ed iterazione 15

Liste 17

Operazioni sulle liste 19

- verificare se un termine è una lista
- verificare se un termine appartiene ad una lista
- determinare la lunghezza di una lista
- concatenare due liste
- eliminare un elemento da una lista
- invertire una lista

Reversibilità delle procedure prolog 23

Predicato cut 26

Predicato fail e negazione 27

Stringhe 29

Modifiche del database 30

Strutture dati complesse: Alberi e grafi 31

Alberi binari di ricerca 33

Prolog ed Intelligenza Artificiale 35

IL LINGUAGGIO PROLOG

(PROgramming in LOGic)

Nasce nel 1973 e si fonda sulle idee di PROGRAMMAZIONE LOGICA avanzate da R.Kowalski.
E' rimasto confinato a pochi circoli accademici europei fino all'annuncio del progetto giapponese (1985-1990).
E' il più noto linguaggio di programmazione dichiarativo e si basa sulla logica dei predicati del I ordine (prova automatica di teoremi - Risoluzione).
Non manipolazione di numeri, ma di simboli; programmazione esplorativa ed incrementale.
Il linguaggio ad "altissimo livello": utilizzabile anche da non- programmatori.
Molto utilizzato Applicazioni di A.I.
Le strutture dati su cui lavora sono alberi e anche i programmi sono strutture dati manipolabili. Vasto utilizzo della ricorsione, non-assegnamento, non presuppone una macchina di Von Neumann.

Metodologicamente ci si concentra più sulla specifica del problema che non sulla strategia di soluzione (efficienza).

PROBLEMI:

- Per programmi complessi con particolari vincoli la programmazione dichiarativa è ancora un'utopia (cut).
- Non adatto per applicazioni numeriche o in tempo reale.
- ALGORITMO = LOGICA + CONTROLLO
- La conoscenza sul problema e' espressa indipendentemente dal suo utilizzo (COSA non COME).
- Alta modularità e flessibilità.
- E' lo schema progettuale di gran parte dei sistemi basati sulla conoscenza (Sistemi Esperti).

LOGICA = CONOSCENZA SUL PROBLEMA
CONTROLLO = STRATEGIA DI SOLUZIONE
LOGICA: CORRETTEZZA ED EFFICIENZA
CONTROLLO: SOLO EFFICIENZA

PROGRAMMA PROLOG

Un programma PROLOG è un insieme di clausole di Horn, che rappresentano:
- fatti, riguardanti gli oggetti in esame e le relazioni che intercorrono;
- regole sugli oggetti e sulle relazioni; (Se.allora)
- goal (clausole senza testa), sulla base di conoscenza definita.

ESEMPIO:
Due individui sono colleghi se lavorano presso la stessa ditta.

```
collega(X,Y) :- lavora(X,Z), lavora(Y,Z),div(X,Y).

lavora(emp1,ibm). lavora(emp2,ibm).
lavora(emp3,txt).
lavora(emp4,olivetti).
lavora(emp5,txt).

?- collega(X,Y).

X=emp1 Y=emp2;
X=emp2; emp1;
X=emp3 Y=emp5;
X=emp5 Y=emp3.
```

DIMOSTRAZIONE DI UN GOAL

Un goal viene dimostrato provando i singoli letterali da sinistra verso destra (regola di selezione *left- most*).

```
:- collega(X,Y),persona(X),persona(Y).
```

Un goal dato da un singolo letterale (goal atomico) viene provato confrontandolo con le teste delle clausole contenute nella base di conoscenza.

Se esiste una sostituzione per cui il confronto ha successo:
- se la clausola con cui si unifica è una clausola unitaria (fatto), la prova termina;
- se è una clausola condizionale (regola), ne viene provato il body.

Se non esiste una sostituzione, il goal fallisce.

ESEMPIO:

```
append([],X,X).
append([X|Z],Y,[X|T]):-append(Z,Y,T).
```

```
?- append([a,b],[c,d],[a,b,c,d]).
?- append([a,b],Y,[a,b,c,d]).
?- append(X,[c,d],[a,b,c,d]).
?- append(X,Y,[a,b,c,d]).
?- append(X,Y,Z).
```

SINTASSI

Un programma Prolog è costituito da un insieme di c*lausole definite* della forma:

(cl1) A.
(cl2) A :- B_1, B_2, ., B_n.

in cui A e B_i sono formule atomiche. A viene detta *testa* della clausola e B_1, B_2, ., B_n viene detto *corpo* della clausola.

- Una clausola quale (cl1) (ossia una clausola il cui corpo è vuoto) prende spesso il nome di *fatto* (o *asserzione*) mentre una clausola quale (cl2) prende spesso il nome di *regola*.
  Il simbolo "," viene usato per indicare la congiunzione mentre il simbolo ":-" indica l'implicazione logica (in cui A è il conseguente e B_1, B_2, ., B_n è l'antecedente).
- Una formula atomica è una formula del tipo:
  p(t_1, t_2, ., t_m) in cui "p" è un simbolo di predicato e t_1, t_2, ., t_m sono *termini*.
- Un termine è definito ricorsivamente nel modo seguente:
  - le costanti sono dei termini; le costanti sono costituite dai numeri (interi e floating point) e dagli atomi alfanumerici (il cui primo carattere deve essere un carattere alfabetico minuscolo). Le costanti possono essere considerate come dei simboli funzionali a zero argomenti;
  - le variabili sono dei termini; le variabili sono stringhe alfanumeriche aventi come primo carattere un carattere alfabetico maiuscolo oppure il carattere "_";
  - f(t_1, t_2, ., t_k) è un termine se "f" è un simbolo di funzione (o operatore) a "k" argomenti e t_1, t_2,., t_k sono dei termini. Un termine del tipo f(t_1, t_2, ., t_k) prende spesso il nome di *struttura.*

Esempio:

- a, pippo, aB, 9, 10.135, a91 sono delle costanti;
- X, X1, Pippo, _pippo, _x, _ sono delle variabili (la variabile "_", in particolare, prende il nome di "variabile anonima");
- f(a), f(g(1)), f(g(1),h(a),27) sono dei termini composti;
- p, p(a,f(X)), p(Y), q(1) sono delle formule atomiche;
- q., p :- q,r. r(Z)., p(X) :- q(X,g(a)). sono delle clausole definite.

Non viene introdotta nessuna distinzione tra le costanti, i simboli funzionali e i simboli di predicato.
Struttura uniforme per i dati (i termini) e i programmi (le formule).

Un goal (o "query") ha come forma generale:

?- B_1, B_2, ., B_n.

dove B_1, B_2, ., B_n. sono delle formule atomiche.

INTERPRETAZIONE DICHIARATIVA

- Le variabili all'interno di una clausola sono considerate come quantificate universalmente (e lo scope del quantificatore è la clausola stessa).
- per ogni asserzione del tipo:
  $p(t_1,t_2,.,t_m)$ se X_1, X_2, ., X_n sono le variabili che occorrono in $t_1,t_2,.,t_m$, il significato è:
  $\forall X_1, \forall X_2, ., \forall X_n\ (p(t_1,t_2,.,t_m))$
- per ogni regola del tipo: A :- B_1, B_2, . B_k. Se Y1, Y2, . ,Y_m sono le variabili che occorrono solo nel corpo della regola e X_1, X_2, ., X_n sono le variabili che occorrono nella testa e nel corpo regola, il significato è:

  $\forall X_1, \forall X_2, ., \forall X_n, \forall Y_1, \forall Y_2, ., \forall Y_m$
  $(B_1, B_2, .\ B_k \rightarrow A)$
  ossia:
  $\forall X_1, \forall X_2, ., \forall X_n$
  $(\exists Y_1, \exists Y_2, ., \exists Y_m(B_1, B_2, .\ B_k)) \rightarrow A)$

Ad esempio, relazioni di parentela:

`padre(X,Y).` "X è il padre di Y"
`madre(X,Y).` "X è la madre di Y"

```
nonno(X,Y):- padre(X,Z),padre(Z,Y).
```
"per ogni X e Y, X è il nonno di Y se esiste Z per cui X è il padre di Z e Z è il padre di Y"

```
nonno(X,Y):- padre(X,Z),madre(Z,Y).
```
"per ogni X e Y, X è il nonno di Y se esiste Z per cui X è il padre di Z e Z è la madre di Y".

ESECUZIONE DI UN PROGRAMMA PROLOG

Una computazione corrisponde al tentativo di dimostrare (mediante risoluzione) che una formula segue logicamente da un programma (e` un teorema). Cioè Determinare una ***sostituzione*** per le variabili della "query" per cui la "query" segue logicamente dal programma.

Dato un programma P e la "query":

?- $p(t_1,t_2,..,t_m)$.

se X_1, X_2, ., X_n sono le variabili che occorrono in $t_1,t_2,..,t_m$, il significato della "query" è:

$\exists X_1, \exists X_2, .,\exists X_n\ p(t_1,t_2,..,t_m)$

e l'obiettivo di una computazione è quello di determinare una sostituzione σ del tipo $\sigma=\{X_1/s_1,X_2/s_2,..,X_n/s_n\}$

(con s_i termini), tale per cui

P … $[p(t_1,t_2,..,t_m)]\sigma$

PROLOG E LA SUA STRATEGIA DI RICERCA

Il linguaggio Prolog, adotta la *strategia in profondità con "backtracking"* perché può essere realizzata in modo efficiente attraverso un unico stack di goal.
Tale stack rappresenta il ramo che si sta esplorando e contiene opportuni riferimenti a rami alternativi da esplorare in caso di fallimento.
Per quello che riguarda la scelta fra nodi fratelli, la strategia Prolog li ordina seguendo l'ordine testuale delle clausole che li hanno generati.

Scelte di Prolog.
Regola di computazione
Regola "left-most"; data una "query":
?- G_1, G_2, ., G_n.
viene sempre selezionato il letterale più a sinistra
G_1.

Strategia di ricerca
In *profondità* (*depth-first*) con *backtracking cronologico*.

STRATEGIA DI RICERCA

Dato un letterale G_1 da risolvere, viene selezionata la prima clausola (secondo l'ordine delle clausole nel programma P) la cui testa è unificabile con G_1.

Nel caso vi siano più clausole la cui testa è unificabile con G_1, la risoluzione di G_1 viene considerata come un*punto di scelta* nella dimostrazione.

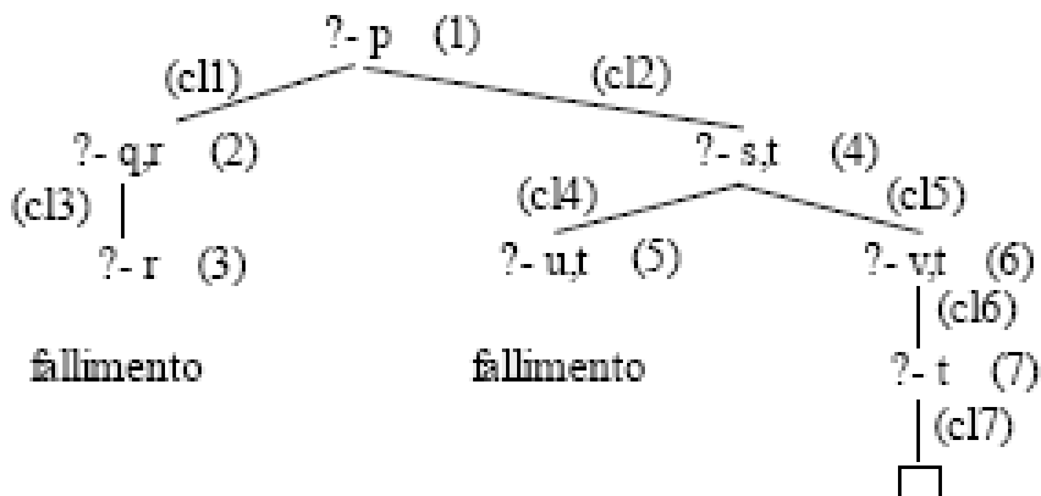
In caso di fallimento in uno dei passi della dimostrazione, il Prolog ritorna in backtracking all'ultimo punto di scelta (in senso cronologico, ossia al punto di scelta più recente) e seleziona la clausola successiva utilizzabile in quel punto per la dimostrazione.

Ciò corrisponde ad una ricerca in profondità con backtracking cronologico dell'albero di dimostrazione SLD.

Esempio:

```
P1   (cl1)  p :- q,r.
     (cl2)  p :- s,t
     (cl3)  q.
     (cl4)  s :- u.
     (cl5)  s :- v.
     (cl6)  t.
     (cl7)  v.
?- p.
```

L'albero di risoluzione SLD:



Esempio:

P2 (cl1') p :- q,r.
(cl2') p.
(cl3') q :- q,t.
?- p.

L'albero SLD:
(cammino infinito percorso)

?- p (1)
(cl1') (cl2')
?- q,r (2) □
(cl3')
?- q,t,r (3)
(cl3')
?- q,t,t,r (4)
?- q,t,t,....,t,r (n)

L'ordine delle clausole in un programma Prolog è rilevante.

Esempio:
Si considerino il programma P_2 ed il programma P_3 ottenuto scambiando le prime due clausole in P_2:

P2
(cl1') **p :- q,r.**
(cl2') **p.**
(cl3') **q :- q,t.**
?- p.

P_3
(cl1") **p.**
(cl2") **p :- q,r.**
(cl3") **q :- q,t.**

I due programmi P_2 e P_3 non sono due programmi Prolog equivalenti. Infatti, data la "query":

?- p. si ha che

- la dimostrazione di "p" in P_2 non termina;
- la dimostrazione di "p" in P_3 ha immediatamente successo.

Una strategia di ricerca in profondità può essere realizzata in modo efficiente utilizzando tecniche non troppo differenti da quelle utilizzate nella realizzazione dei linguaggi imperativi tradizionali.

SOLUZIONI MULTIPLE E DISGIUNZIONE

In alcuni casi può essere che vi siano più sostituzioni di risposta per una "query".
Per forzare l'interprete a fornire ulteriori soluzioni è sufficiente forzare un fallimento nel punto in cui si è determinata la soluzione e far partire il meccanismo di backtracking.
Tale meccanismo porta ad espandere ulteriormente l'albero di dimostrazione SLD alla ricerca del prossimo cammino di successo.
Nel Prolog standard tali soluzioni possono essere richieste mediante l'operatore ";".

```
?- sorella(maria,W).
Yes  W=giovanni;
     W=anna;
no
```

Il carattere ";" può essere interpretato come un operatore di disgiunzione che separa soluzioni alternative.
Lo stesso simbolo ";" può essere utilizzato anche all'interno di un programma Prolog per esprimere la disgiunzione.

INTERPRETAZIONE PROCEDURALE

Una *procedura* è un insieme di clausole di P con le teste costuituite a partire dallo stesso predicato con lo stesso numero di argomenti.
Gli argomenti che compaiono nella testa della procedura possono essere visti come i *parametri formali*.
Una "query" del tipo: ?- $p(t_1,t_2,..,t_n)$.

e` il *richiamo* (chiamata) della procedura "p". Gli argomenti di "p" (ossia i termini $t_1,t_2,..,t_n$) sono i *parametri attuali*.

L'unificazione e` il meccanismo di *passaggio dei parametri*.
Non vi è alcuna distinzione a priori tra i parametri di ingresso e i parametri di uscita (*reversibilità*).

Il corpo di una clausola può a sua volta essere visto come una sequenza di chiamate di procedure. Due clausole con le stesse teste corrispondono a due definizioni alternative del corpo di una procedura.
Tutte le variabili sono a *singolo assegnamento*. Il loro valore è unico durante tutta la computazione e slegato solo quando si cerca una soluzione alternativa (“backtracking”).

VERSO UN VERO LINGUAGGIO DI PROGRAMMAZIONE

Al Prolog puro devono, tuttavia, essere aggiunte alcune caratteristiche per poter ottenere un linguaggio di programmazione utilizzabile nella pratica.

In particolare:
- Strutture dati e operazioni per la loro manipolazione.
- Meccanismi per la definizione e valutazione di espressioni e funzioni.
- Meccanismi di input/output.
- Meccanismi di controllo della ricorsione e del backtracking.
- Negazione

Tali caratteristiche sono state aggiunte al Prolog puro attraverso la definizione di alcuni predicati speciali (*predicati built-in)* predefiniti nel linguaggio e trattati in modo speciale dall'interprete.

ARITMETICA E RICORSIONE

Non esiste, in logica, alcun meccanismo per la valutazione di funzioni.
Definire funzioni e valutare espressioni è fondamentale.
I numeri interi possono essere rappresentati come termini Prolog.
Il numero intero N è rappresentato dal termine:
s(s(s(...s(0)...)))
N volte

ESEMPIO:
Specifica della funzione somma:

```
sum(0,X,X).
sum(s(X),Y,s(Z)):- sum(X,Y,Z).
```

– non esiste interpretazione del simbolo sum;
– i numeri interi sono rappresentati dalla struttura successore;
– si utilizza ricorsione;
– esistono molte possibili interrogazioni per lo stesso database (programma).

POSSIBILI INTERROGAZIONI:

```
?- sum(s(0),s(s(0)),Y).
?- sum(s(0),Y,s(s(s(0)))).
?- sum(X,Y,s(s(s(0)))).
?- sum(X,Y,Z).
?- sum(X,Y,s(s(s(0)))), sum(X,s(0),Y).
```

PREDICATI PREDEFINITI PER LA VALUTAZIONE DI ESPRESSIONI

L'insieme degli atomi Prolog comprende tanto i numeri interi quanto i numeri "floating-point".
I principali operatori aritmetici sono simboli funzionali (operatori) predefiniti del linguaggio. In questo modo ogni espressione può essere rappresentata come un termine Prolog.
Per gli operatori aritmetici binari il Prolog consente tanto una notazione prefissa (funzionale), quanto la più tradizionale notazione infissa.

Tabella - Operatori aritmetici

operatori unari	-, exp, log, ln, sin, cos, tg
operatori binari	+, -, *, / , mod, div

+(2,3) e 2+3 sono due rappresentazioni equivalenti.
2+3*5 viene correttamente interpretata come l'espressione 2+(3*5)

Data una espressione è necessario un meccanismo per la valutazione.
Speciale predicato predefinito "is".

T is Expr   o   is(T,Expr)

dove:

- T può essere un atomo numerico o una variabile
- Expr deve essere un'espressione.
- L'espressione Expr viene valutata ed il risultato della valutazione viene unificato con T.
- Le variabili in Expr devono essere istanziate al momento della valutazione.

Esempi

```
?- X is 2+3.
yes      X=5

?- X1 is 2+3, X2 is exp(X1), X is X1*X2.
yes      X1=5   X2=148.413   X=742.065

?- 0 is 3-3.
yes
?- X is Y-1.
no
```

La variabile Y non è istanziata al momento della valutazione.

```
?- X is 2+3, X is 4+5.
no
```

```
?- X is 2+3, X is 4+1
yes      X=5
```
In questo caso il secondo goal della congiunzione risulta essere:
```
?- 5 is 4+1.          che ha successo.
?- X is 2+3, X is X+1.
no
```

- Nel caso dell'operatore "is" l'ordine dei goal è rilevante.

(a) `?- X is 2+3, Y is X+1.`

(b) `?- Y is X+1, X is 2+3.`

Mentre il goal (a) ha successo e produce la coppia di istanziazioni X=5, Y=6, il goal (b) fallisce.

Un termine che rappresenta un'espressione viene valutato solo se è il secondo argomento del predicato "is".

```
p(a,2+3*5).
q(X,Y) :- p(a,Y), X is Y.
```

Si ha:

```
? q(X,Y).
yes      X=17   Y=2+3*5  (Y=+(2,*(3,5)))
```

Il predicato predefinito "is" è un tipico esempio di un predicato predefinito non reversibile; come conseguenza le procedure che fanno uso di tale predicato non sono (in generale) reversibili.

OPERATORI RELAZIONALI

Il Prolog fornisce operatori relazionali per confrontare i valori di espressioni.
Tali operatori sono utilizzabili come goal all'interno di una clausola Prolog ed hanno notazione infissa.

Tabella - Operatori relazionali

>, <, <=, >=, ==,=:= (uguaglianza aritmetica), =/= (disuguaglianza)

Al momento in cui viene selezionato un goal del tipo: $Expr_1$ rel $Expr_2$ si effettuano i seguenti passi:

- $Expr_1$ ed $Expr_2$ vengono valutate; Expr1 ed Expr2 devono essere completamente istanziate.
- I risultati della valutazione di Expr1 ed Expr2 vengono confrontati secondo la relazione "rel".

Esempio
Calcolare la funzione abs(x) = |x| abs(X,Y)
"Y è il valore assoluto di X"

```
abs(X,X) :- X >= 0.
abs(X,Y) :- X < 0, Y is -X.
```

Esempio
Si consideri la definizione delle seguenti relazioni:
pari(X) = true se X è un numero pari, false se X è un numero dispari
dispari(X) =true se X è un numero dispari, false se X è un numero pari

pari(X)
"X è un numero pari"

```
pari(0).
pari(X) :- X > 0, X1 is X-1, dispari(X1).
dispari(X) :- X > 0, X1 is X-1, pari(X1).
```

RELAZIONI TRA TERMINI

Sono definiti un certo numero di operatori relazionali.

Confronto di espressioni aritmetiche

Sono definiti i seguenti operatori per il confronto di espressioni aritmetiche (tutti gli operatori sono binari e non associativi; gli argomenti di tali operatori sono espressioni che vengono valutate prima del confronto):

- $E_1 =:= E_2$

i valori delle due espressioni E_1 ed E_2 sono uguali; Si noti che "?- E_1=E_2" provocherebbe un tentativo di unificare i due termini senza la loro valutazione.

- $E_1 =/= E_2$

i valori delle due espressioni E_1 ed E_2 sono diversi;

- $E_1 > E_2$

- $E_1 >= E_2$

- $E_1 < E_2$

- $E_1 =< E_2$

Unificazione e uguaglianza tra termini

Due diversi operatori per effettuare la verifica se due termini sono unificabili o uguali.

- **$T_1 = T_2$**

verifica se T_1 e T_2 sono unificabili. Viene generata la sostituzione che unifica i due termini (e vengono pertanto legate le variabili nei due termini).

Es. ?- f(X,g(a)) = f(h(c),g(Y)).
yes X=h(c) Y=a

```
?- 2+3 = 5.
no
```

- **$T_1 == T_2$**

verifica se T_1 e T_2 sono uguali (identici). In parti- colare, se i due termini contengono due variabili in posizioni equivalenti, perché i termini siano uguali, le due variabili devono essere la stessa variabile. Si noti che in questo caso non viene generata alcuna sostituzione.

Es.
```
?- f(a,b) == f(a,b).        yes

?- f(a,X) == f(a,b).        no

?- f(a,X) == f(a,X).        yes

?- f(a,X) == f(a,Y).        no
```

- **T_1 \== T_2**

verifica se T_1 e T_2 non sono uguali (identici).

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 èun'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" Ad esempio:

```
?-    atom(pippo).          yes

?-    atomic(pippo).        yes
```

```
?-    numer(1).              yes

?-    atomic(1).             yes

?-    atom(1).               no

?-    var(X).                yes

?-    nonvar(p(a,X)).        yes

?-    X=f(Y),nonvar(X).      yes
```

CALCOLO DI FUNZIONI

Una funzione può essere realizzata mediante relazioni Prolog.
Data una funzione f ad n argomenti, essa può essere realizzata mediante un predicato ad n+1 argomenti nel modo seguente:

f: $x_1, x_2, \ldots, x_n \rightarrow y$
diventa
f(X1,X2, ...,Xn,Y) :- <calcolo di Y>

Esempio
Calcolare la funzione fattoriale così definita: fatt: n $\rightarrow$ n ! (n intero positivo)

fatt(0) = 1
fatt(n) = n * fatt(n-1) (per n>0) fatt(X,Y) "Y è il fattoriale di X" fatt(0,1).

```
fatt(N,Y) :- N>0, N1 is N-1, fatt(N1,Y1), Y is N*Y1.
```

Esempio
Calcolare la funzione "massimo comun divisore" così definita:
mcd: x,y $\rightarrow$ MCD(x,y) (x,y interi positivi)
MCD(x,0) = x
MCD(x,y) = MCD(y, x mod y) (per y>0)

mcd(X,Y,Z)
"Z è il massimo comun divisore di X e Y"

```
mcd(X,0,X).
mcd(X,Y,Z) :- Y>0, X1 is X mod Y, mcd(Y,X1,Z).
```

RICORSIONE ED ITERAZIONE

Il Prolog non fornisce alcun costrutto sintattico per l'iterazione (quali, ad esempio, i costrutti "while" e "repeat") e l'unico meccanismo per ottenere iterazione è la definizione ricorsiva.

RICORSIONE TAIL

Una funzione f è definita per ricorsione tail se f è la funzione "più esterna" nella definizione ricorsiva o, in altri termini, se sul risultato del richiamo ricorsivo di f non vengono effettuate ulteriori operazioni.
La definizione di funzioni (predicati) per ricorsione tail può essere considerata come una definizione per *iterazione*.
Potrebbe essere valutata in spazio costante mediante un processo di valutazione iterativo.
Si dice *ottimizzazione della ricorsione tail* valutare una funzione tail ricorsiva f mediante un processo iterativo ossia caricando un solo record di attivazione per f sullo stack di valutazione (esecuzione).

In Prolog l'ottimizzazione della ricorsione tail è un po' più complicata che non nel caso dei linguaggi imperativi.
Non-determinismo e presenza di punti di scelta nella definizione di un predicato.

```
            p(X) :- c1(X), g(X).
(a)         p(X) :- c2(X), h1(X,Y), p(Y).
(b)         p(X) :- c3(X), h2(X,Y), p(Y).
```

Effettuare l'ottimizzazione della ricorsione tail su questo programma risulta difficile perche' vi sono due possibilità nella valutazione ricorsiva di un goal "?- p(Z)":

- se viene scelta la clausola (a), si deve ricordare che (b) è un punto di scelta ancora aperto. Mantenere alcune informazioni contenute nel record di attivazione di p(Z) (i punti di scelta ancora aperti).

- se viene scelta la clausola (b) (più in generale, l'ultima clausola della procedura), non è più necessario mantenere alcuna informazione contenuta nel record di attivazione di p(Z) e la rimozione di tale record di attivazione può essere effettuata.

Quindi:
In Prolog l'ottimizzazione della ricorsione tail è possibile solo se la scelta nella valutazione di un predicato "p" è deterministica o, meglio, se al momento del richiamo ricorsivo (n+1)-esimo di "p" non vi sono alternative aperte per il richiamo al passo n-esimo (ossia alternative che potrebbero essere considerate in fase di backtracking).
Quasi tutti gli interpreti Prolog effettuano l'ottimizzazione della ricorsione tail ed è pertanto conveniente usare il più possibile ricorsione di tipo tail.

RICORSIONE NON-TAIL

Il predicato "fatt" è definito con una forma di ricorsione semplice (non tail).
Casi in cui una relazione ricorsiva può essere trasformata in una relazione tail ricorsiva.

Un modo alternativo per il calcolo del fattoriale:

```
fatt1(N,Y) :- fatt1(N,1,1,Y).
fatt1(N,M,ACC,ACC) :- M > N.
fatt1(N,M,ACC,Y) :- ACC1 is ACC*M, M1 is M+1, fatt1(N,M1,ACC1,Y).
```

il fattoriale viene calcolato utilizzando un argomento "ACC" di accumulazione:
ACC_0 = 1 (inizializzazione)
ACC_1 = *(1, ACC_0) = *(1, 1)
ACC_2 = *(2, ACC_1) = *(2, *(1, 1))
ACC_{N-1}=*(N-1,ACC_{N-2})= *(N-1,*(N-2,... *(2, *(1, 1)) ...))
ACC_N =*(N, ACC_{N-1}) = *(N, *(N-1, *(N-2, ... *(2, *(1, 1)) ...)))

Esempio
"fatt" in un' altra struttura di tipo iterativo:
fatt2(N,Y) "Y è il fattoriale di N"

```
fatt2(N,Y) :- fatt2(N,1,Y).
fatt2(0,ACC,ACC).
fatt2(M,ACC,Y) :-ACC1 is M*ACC, M1 is M-1, fatt2(M1,ACC1,Y).
```

Esempio
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(N) = fibonacci(N-1) + fibonacci(N-2) per N >1

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

LISTE

Le liste sono una delle strutture dati primitive più diffuse nei linguaggi di programmazione per l'elaborazione simbolica (es: Lisp).
In Prolog le liste sono dei termini costruiti a partire da uno speciale atomo (che denota la lista vuota) ed utilizzando un particolare operatore funzionale (l'operatore ".").

La definizione di lista può essere data ricorsivamente nel modo seguente:

Definizione

- l'atomo [] è una lista (lista vuota)
- il termine .(T,LISTA) è una lista se T è un termine (qualsiasi) e LISTA è una lista.

T prende il nome di *testa* (head) della lista e LISTA il nome di *coda* (tail) della lista

Esempio

I seguenti termini Prolog sono liste:

(1) []
(2) .(a, [])
(3) .(a, .(b,[]))
(4) .(f(g(X)), . (h(z), .(c, [])))
(5) .([], [])
(6) .(.(a,[]), .(b, []))

Il Prolog fornisce una notazione semplificata per la rappresentazione delle liste:
la lista .(T, LISTA) può essere rappresentata anche come [T | LISTA]

Esempio
Le liste nell'esempio precedente possono essere rappresentate nel modo seguente:

(1) []
(2) [a | []]
(3) [a | [b | []]]
(4) [f(g(X)) | [h(z) | [c | []]]]
(5) [[] | []]
(6) [[a | []] | [b | []]]

Ulteriore semplificazione; la lista [a | [b | [c]]] può essere rappresentata nel modo seguente:
[a, b, c]

Esempio

Le liste nell'esempio precedente possono essere rappresentate nel modo seguente:

(1) []
(2) [a]
(3) [a, b]
(4) [f(g(X)), h(z), c]
(5) [[]]
(6) [[a], b]

UNIFICAZIONE SU LISTE

L'unificazione (combinata con le varie notazioni per le liste) è un potente meccanismo per l'accesso alle liste.
D.B.: `p([1,2,3,4,5,6,7,8,9]).`

```
?- p(X).
yes      X=[1,2,3,4,5,6,7,8,9]

?- p([X|Y]).
yes      X=1         Y=[2,3,4,5,6,7,8,9]

?- p([X, Y | Z]).
yes      X=1         Y=2         Z=[3,4,5,6,7,8,9]

?- p([_ | X]).
yes      X=[2,3,4,5,6,7,8,9]

?- p([_, X | _]).
yes      X=2

?- p(X), Y=[0|X].
yes      X=[1,2,3,4,5,6,7,8,9]      Y=[0,1,2,3,4,5,6,7,8,9]

?- p([_ | X]), Y=[0 | X].
yes      X=[2,3,4,5,6,7,8,9]        Y=[0,2,3,4,5,6,7,8,9]
```

OPERAZIONI SULLE LISTE

Le procedure che operano su liste sono definite come procedure ricorsive basate sulla definizione ricorsiva di lista.

VERIFICARE SE UN TERMINE È UNA LISTA

is_list(T) = true se T è una lista false se T non è una lista

```
is_list([]).
is_list([X | L]) :- is_list(L).

?- is_list([1,2,3]).
yes
?- is_list([a | b]).
no
```

VERIFICARE SE UN TERMINE APPARTIENE AD UNA LISTA

member(T,L) "T è un elemento della lista L"

```
member(T, [T | _]).
member(T, [_ | L]) :- member(T, L).

?-      member(2, [1,2,3]).
yes
?-      member(1, [2,3]).
no
?-      member(X, [1,2,3]).
yes       X=1
yes       X=1; X=2; X=3;
no
```

La relazione "member" può quindi essere utilizzata in più di un modo (per la verifica di appartenenza di un elemento ad una lista o per individuare gli elementi di una lista).

DETERMINARE LA LUNGHEZZA (IL NUMERO DI ELEMENTI) DI UNA LISTA

length(L,N) la lista L ha N elementi

Si noti che un elemento può a sua volta essere una lista o una struttura complessa. Ogni elemento (indipendentemente dal tipo) conta 1 nel determinare la lunghezza di una lista.

Versione ricorsiva:

```
length([], 0).
length([HEAD | TAIL], N) :- length(TAIL, N1), N is N1+1.
```

Versione iterativa:

Usare un accumulatore per calcolare la lunghezza della lista.
Tale accumulatore viene inizializzato a zero e unificato con il risultato quando la lista è vuota.
Ad ogni passo l'accumulatore viene aumentato di 1 e la lista diminuita di un elemento.

```
length1(L, N) :- length1(L, 0, N).
lenght1([], ACC, ACC).
length1([HEAD | TAIL], ACC, N) :- ACC1 is ACC+1, length1(TAIL, ACC1, N).
```

CONCATENARE DUE LISTE

append(L1, L2, L) "la lista L è la concatenazione delle liste L1 e L2"

Ad esempio:

[1,2,3] è la concatenazione di [] e [1,2,3]
[1,2,3] è la concatenazione di [1] e [2,3]
[1,2,3] non è la concatenazione di [2,3] e [1]

Si ha allora la seguente procedura:

```
append([], L2, L2).
append([H|T],L2,[H|T1]):- append(T, L2, T1).

?- append([1,2], [3,4,5], L).
yes        L=[1,2,3,4,5]

?- append([a,b], [c,d], L).
yes        L=[a,b,c,d]

?- append(L1, [c,d], [a,b,c,d]).
yes        L1=[a,b]

?- append([a,b], L2, [a,b,c,d]).
yes        L2=[c,d]
```

```
?- append(L1, L2, [a,b,c,d]).
yes        L1=[]           L2=[a,b,c,d];
           L1=[a]          L2=[b,c,d];
           L1=[a,b]        L2=[c,d];
           L1=[a,b,c]      L2=[d];
           L1=[a,b,c,d]    L2=[]

?- append(L1, [c,d], L).
yes        L1=[]           L=[c,d];
           L1=[_1]         L=[_1,c,d];
           L2=[_1,_2]      L=[_1,_2,c,d];
           (infinite soluzioni)

?- append([a,b], L1, L).
yes        L1=_1           L=[a,b | _1]

?- append(L1, L2, L3).
yes        L1=[]           L2=_1              L3=_1;
           L1=[_1]         L2=_2,             L3=[_1 | _2];
           L1=[_1,_2]      L2=_3              L3=[_1,_2 | _3];
           (infinite soluzioni)
```

ELIMINARE UN ELEMENTO DA UNA LISTA

delete1(T,L,L1)	L1 è la lista da cui è stato eliminato il primo termine unificabile con T
delete2(T,L,L1)	L1 è la lista L da cui sono stati eliminati tutti i termini unificabili con T

```
delete1(T, [], []).
delete1(T, [T | TAIL], TAIL).
delete1(T, [HEAD | TAIL], [HEAD | L]) :- delete1(T, TAIL, L).

delete2(T, [], []).
delete2(T, [T | TAIL], L):-delete(T, TAIL, L).
delete2(T, [HEAD | TAIL], [HEAD | L]):- delete(T, TAIL, L).
```

Le due procedure "delete1" e "delete2" forniscono una risposta corretta ma non sono corrette in fase di backtracking.

```
?- delete1(a,[a,b,a,c],L).
yes        L=[b,c];
           L=[b,a,c];
           L=[a,b,c];
           L=[a,b,a,c];
no
```

Solo la prima soluzione è corretta.
Il problema è legato alla mutua esclusione tra la seconda e la terza clausola della relazione "delete".
Se T appartiene alla lista (per cui la seconda clausola di "delete" ha successo), allora la terza clausola non deve essere considerata una alternativa valida.

INVERTIRE UNA LISTA

reverse(L,Lr) "Lr è una lista che contiene gli stessi elementi di L ma in ordine invertito"

Ad esempio:

[] è l'inversa di []
[2,1] è l'inversa di [1,2]
[9,8,7,6,5,4,3,2,1] e' l'inversa di [1,2,3,4,5,6,7,8,9]

```
reverse([], []).
reverse([H | T], Lr) :- reverse(T,T1), append(T1, [H], Lr).
```

Altra versione (iterativa)

```
reverse1(L,Lr):- reverse1(L,[],Lr).
reverse1([], ACC, ACC).
reverse1([H | T], ACC, Y) :- append([H], ACC, ACC1), reverse1(T, ACC1, Y).
```

Versione finale della procedura "reverse1":

```
reverse1(L,Lr) :- reverse1(L,[],Lr).
reverse1([], ACC, ACC).
reverse1([H | T], ACC, Y) :- reverse1(T, [H | ACC], Y).
```

```
?- reverse1([1,2,3], Lr).
yes         Lr=[3,2,1]
```

La lista L da invertire viene diminuita ad ogni passo di un elemento e tale elemento viene accumulato in una nuova lista (in testa).
Un elemento viene accumulato davanti all'elemento che lo precedeva nella lista originaria, ottenendo in questo modo l'inversione. Quando la prima lista è vuota, l'accumulatore contiene la lista invertita.

REVERSIBILITA' DELLE PROCEDURE PROLOG

Le procedure dei linguaggi di programmazione imperativi e le funzioni dei linguaggi di programmazione funzionali hanno precisa "direzione di esecuzione", in Prolog (puro) non c'è distinzione tra variabili di ingresso e variabili di uscita.Ogni variabile può essere usata indifferentemente come variabile di ingresso o di uscita.

Ricordiamo il programma relativo alla concatenazione di due liste:

append([ ], L2, L2).
append([H | T], L2, [H | T1]) :-append(T, L2, T1).

Consideriamo: **append(L1, L2, L)** sappiamo che L è la concatenazione di L1 e L2, ma cosa succede se utilizziamo la procedura in modi differenti?

1) Consideriamo i primi due argomenti istanziati ed il terzo una variabile

   :-append([a,b], [c,d], L).
   yes   L = [a, b, c, d]

   In questo caso append è utilizzata per la concatenazione di due liste.

2) il primo argomento è una variabile, il secondo e il terzo sono istanziati

   :-append(L1, [c, d], [a, b, c, d]).
   Yes   L1 = [a, b]

   :-append(L1, [c, d], [c, d, e]).
   No

   Il risultato è una lista L1 (se esiste) tale che concatenata con la lista presente come secondo argomento produce come risultato la lista presente come terzo argomento.

3) Il primo argomento e il terzo sono istanziati, il secondo è una variabile

   :-append([a, b], L2, [a, b, c, d]).
   Yes   L2 = [c, d]

   :-append([a, b], L2, [e, a, b]).
   No

   Il risultato è una lista L2 (se esiste) tale che concatenata alla lista presente come primo argomento produce come risultato la lista presente come terzo argomento.

4) Il terzo argomento è istanziato, il primo e il secondo sono variabili

:-append(L1, L2, [a, b, c, d]).

Cosa succede a questo punto?

Yes	**L1 = [ ]**	**L2 = [a, b, c, d];**
	L1 = [a]	**L2 = [b, c, d];**
	L1 = [a, b]	**L2 = [c, d];**
	L1 = [a, b, c]	**L2 = [d];.**
	L1 = [a, b, c, d]	**L2 = [ ];**

Il risultato è dato da tutte le combinazioni di L1 ed L2 la cui concatenazione produce la lista al terzo argomento

5) Il primo argomento e il terzo sono variabili, il secondo è istanziato

:-append(L1, [c, d], L).

Cosa succede a questo punto?

Yes	**L1 = [ ]**	**L = [c, d];**
	L1 = [_1]	**L = [_1, c, d];**
	L1 = [_1, _2]	**L = [_1, _2, c, d];**
	…	
	…	
	…	
	(infinite soluzioni)	

6) Il primo argomento è istanziato, il secondo e il terzo sono variabili

:-append([a, b], L1, L).

Yes	**L1 = _1**	**L = [a, b | _1]**

7) Tutti gli argomenti sono variabili

:-append(L1, L2, L3).

Yes	**L1 = [ ]**	**L2 = _1**	**L3 = _1;**
	L1 = [_1]	**L2 = _2**	**L3 = [_1 | _2];**
	L1 = [_1, _2]	**L2 = _3**	**L3 = [_1, _2 | _3];**
	…		
	…		
	…		
	(infinite soluzioni)		

Altro esempio, ricordiamo le procedure elimina1 ed elimina2:

elimina1(X, [ ], [ ]).
elimina1(X, [X | Tail], Tail).
elimina1(X, [Head | Tail], [Head | L]) :- elimina1(X, Tail, L).

elimina2(X, [ ], [ ]).
elimina2(X, [X | Tail], L) :-elimina2(X, Tail, L).
elimina2(X, [Head | Tail], [Head | L]) :-elimina2(X, Tail, L).

Queste procedure possono essere utilizzate, oltre che per eliminare termini da una lista anche per aggiungere termini ad una lista o per determinare elementi della differenza tra due liste, consideriamo il seguente esempio:

:- elimina(3, L, [1, 2]).

Yes **L = [3, 1, 2];**
L = [1, 3, 2];
L = [1, 2, 3];

Per concludere:

- Tutte le operazioni operanti su liste che abbiamo visto fin qui sono reversibili
- La reversibilità non è una caratteristica dei programmi che operano su liste: ogni programma in Prolog puro è reversibile
- Esistono alcune eccezioni:
  L'uso di alcuni predicati predefiniti fa perdere la reversibilità, per esempio il predicato is e tutte le procedure che fanno uso di tale predicato non sono (in generale) reversibili

CONTROLLO DI UN PROGRAMMA

Predicati predefiniti che consentono di influenzare e controllare il processo di esecuzione (dimostrazione) di un goal.

IL PREDICATO "CUT" ("!")

Il predicato predefinito "cut" (denotato dal simbolo "!") è uno dei più importanti e più complessi predicati di controllo forniti dal Prolog.

Si consideri il seguente programma Prolog:

(cl1) **a :- p,b.**
(cl2) **a :- p,c.**
(cl3) **p.**

e la valutazione del goal

?- a

Si ha:

"p" ha successo ma "c" fallisce e si ha quindi un nuovo fallimento.
Poiché non vi sono punti di scelta ancora attivi, si ha un fallimento globale e quindi la risposta:

?- a.
no

Effetto del cut
L'effetto del "cut" è quello di "congelare", cioè rendere definitive, alcune delle scelte fatte dall'interprete Prolog nel corso della valutazione, ossia quello di eliminare alcuni blocchi dallo stack di backtracking.
Si consideri la clausola:
p :- q_1, q_2, ..., q_i, !, q_{i+1}, q_{i+2}, ... q_n. l'effetto della valutazione del goal "!" (cut) durante la dimostrazione del goal "p" è il seguente:

- la valutazione di "!" ha successo (come quasi tutti i predicati predefiniti, "!" ha sempre successo e viene ignorato in fase di backtracking);
- tutte le scelte fatte nella valutazione dei goals q_1, q_2, ..., q_i e in quella del goal p vengono congelate (ossia rese definitive); in altri termini, tutti i punti di scelta per tali goal (per le istanze di tali goal utilizzate) vengono rimossi dallo stack di backtracking.

Le alternative riguardanti i goal precedenti il goal"p" non vengono modificate dal "cut".
Se q_{i+1}, q_{i+2}, ... q_n fallisce si ha quindi un fallimento generale del goal "p" (poiché tutti gli eventuali punti di scelta per q_1, q_2, ...,q_i e per p stesso erano stati rimossi dal "!").
Vengono cioè tagliati alcuni rami dell'albero SLD di ricerca.

SPECIFICA DEL DETERMINISMO

Specificare la mutua esclusione tra due clausole in modo semplice attraverso l'uso del predicato "cut".

Si consideri il seguente schema per la procedura "p":

p(.) :- a(.),b.
p(.) :- c.

Si supponga che la condizione "a(.)" debba rendere le due clausole mutuamente esclusive. Realizzare uno schema del tipo:

if a(.) then b else c

Utilizzando il predicato predefinito "cut":

p(.) :- a(.), !, b.
p(.) :- c.

In questo modo se a(.) è vero si valuta b, e il successo dell'intera procedura dipende da quello di b (se a(.) è vero c non verrà mai valutato, neanche in backtracking).

IL PREDICATO "FAIL"

Fail e' un predicato predefinito senza argomenti. La valutazione del predicato "fail" fallisce sempre.

LA NEGAZIONE IN PROLOG

Non abbiamo preso in esame il trattamento di informazioni negative.
Solo programmi logici costituiti da clausole definite e che quindi non possono contenere atomi negati. Inoltre, attraverso la risoluzione SLD, non è possibile derivare informazioni negative.
La forma di negazione introdotta in quasi tutti i linguaggi logici, Prolog compreso, prende il nome di *negazione per fallimento*.
Puo' essere realizzata facilmente scambiando tra loro la nozione di successo e di fallimento.

not P

"not P ha successo su un insieme DB di clausole se e solo se la dimostrazione di P su DB fallisce in modo finito"

La combinazione "!,fail" è interessante ogni qual volta si voglia, all'interno di una delle clausole per una relazione "p", generare un fallimento globale per "p" (e non soltanto un backtracking verso altre clausole per "p").

Consideriamo il problema di voler definire una proprietà "p" che vale per tutti gli individui di una data classe tranne alcune eccezioni.

Tipico esempio è la proprietà "volare" che vale per ogni individuo della classe degli uccelli tranne alcune eccezioni (ad esempio, i pinguini o gli struzzi).

Tale relazione potrebbe essere rappresentata nel modo seguente:

```
vola(X) :- pinguino(X), !, fail.
vola(X) :- struzzo(X), !, fail.
....
vola(X) :- uccello(X).
```

STRINGHE

Sebbene il Prolog standard non fornisca un tipo di dati primitivo "stringa", esso fornisce la possibilità di trattare le stringhe come particolari tipi di liste.
Più in particolare, una stringa viene identificata come la lista dei codici ASCII dei caratteri che la compongono; così, ad esempio, la stringa "abc" viene identificata con la lista [97,98,99].
Per rendere i programmi più leggibili, viene comunque mantenuta la possibilità di descrivere le stringhe come sequenze di caratteri tra doppi apici: l'interfaccia del Prolog provvede a passare da tale notazione a quella a liste.

Poiché una stringa viene rappresentata come una lista, tutte le operazioni sulle liste possono essere utilizzate per definire operazioni su stringhe.
Ad esempio, l'operazione di concatenazione tra stringhe può essere effettuata mediante l'operazione di concatenazione tra liste

?- append("abc","def",S).
yes S=[97,98,99,100,101,102]
(ossia S="abcdef")

Predicato predefinito che permette di trasformare un atomo nella stringa di caratteri corrispondente (e viceversa).

name(ATOMIC,LIST)

- ATOMIC deve essere un atomo (una struttura T per cui atomic(T) è vero) o una variabile;
- LIST può essere una lista di codici ASCII o una variabile;
- i due argomenti non possono essere simultaneamente delle variabili.

Il significato di tale predicato è il seguente:

name(ATOMIC,LIST)
"LIST è la stringa (lista di codici ASCII) corrispondente all'atomo ATOMIC"

?- name(a,[97]). **yes**

?- name(abc,L). **yes L=[97,98,99]**

?- name(A,"pippo"). **yes A=pippo**

?- name(A,[97,99]). **yes A=ac**

?- name(f(a),L). **error - illegal argument to name(atomic,list)**

?- name(A,[2000]). **error - illegal argument to name(atomic,list)**

MODIFICHE DEL DATA-BASE

Predicati predefiniti per modificare dinamicamente il data-base con l'aggiunta o la rimozione di una o più clausole.

Molto pericolosi in quanto fanno perdere la semantica dichiarativa dei programmi e, in molti casi, rendono il comportamento di un programma difficile da comprendere.

Aggiunta

Predicato "assert":

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

Al momento della 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.

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" Come quasi tutti i predicati predefiniti, il predicato "assert" viene ignorato in fase di backtracking.

Rimozione

Predicato "retract":

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

Al momento della valutazione, T deve essere istanziato ad un termine che denota una clausola.

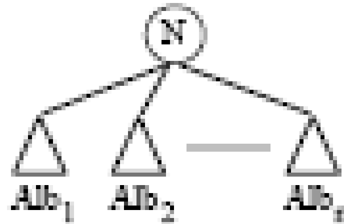
Può capitare che più clausole siano unificabili con T: in tal caso viene rimossa la prima clausola.

STRUTTURE DATI COMPLESSE

ALBERI

Un albero può essere definito ricorsivamente nel modo seguente:

- l'albero vuoto (denotato spesso con il simbolo "nil") è un albero;
- un nodo "N" cui siano associati n alberi (sottoalberi di N, con n = 0) è un albero.

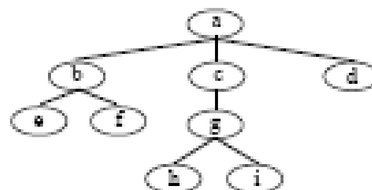


- N prende il nome di *radice* dell'albero;
- Alb_1, Alb_2, ..., Alb_n prendono il nome di *sottoalberi* di N;
- un nodo senza sottoalberi prende il nome di *foglia*.

RAPPRESENTAZIONE IN PROLOG DI ALBERI

Rappresentazione come termine strutturato

Ad ogni nodo N corrisponde un simbolo funzionale che ha tanti argomenti quanti sono i sottoalberi del nodo. Tali argomenti saranno a loro volta dei termini che rappresentano i sottoalberi di N.



a(b(e,f),c(g(h,i)),d)

Oppure (se il numero massimo di figli di un nodo è noto):

a(b(e(nil,nil,nil),f(nil,nil,nil),nil), c(g(h(nil,nil,nil),i(nil,nil,nil),nil)), d(nil,nil,nil)

Rappresentazione mediante liste

Un albero viene rappresentato come una lista il cui primo elemento è la radice dell'albero e i cui elementi successivi sono le liste che rappresentano i sottoalberi della radice stessa (da sinistra verso destra).

Ad esempio, l'albero precedente è rappresentato dalla seguente lista:

[a, [b, [e], [f]], [c, [g, [h], [i]], [d]]

Oppure, utilizzando tutte liste della stessa lunghezza:

[a, [b, [e, nil, nil, nil], [f, nil, nil, nil], nil], [c, [g, [h, nil, nil, nil], [i, nil, nil, nil], nil], [d, nil, nil, nil]]

Rappresentazione come insieme di fatti

Introdurre uno speciale predicato binario, denotato ad esempio con "f", con il seguente significato: f(X,Y) "Y è figlio di X"

In questo modo un albero è rappresentato dall'insieme di clausole che descrivono le relazioni di "figlio" tra i nodi che costituiscono l'albero stesso.

Ad esempio, l'albero precedente è rappresentato dal seguente insieme di fatti:

f(a,b).
f(a,c).
f(a,d).
f(b,e).
f(b,f).
f(c,g).
f(g,h).
f(g,i).

Si potrebbe anche pensare che "f" abbia tanti argomenti quanto è il numero massimo di figli di un nodo e rappresentare quindi mediante un unico fatto tutti i figli di un certo nodo "N".

I tre tipi di rappresentazione hanno caratteristiche differenti:

- Rappresentazione come termine o lista:
  - E` sicuramente la più elegante e vicina alla filosofia della programmazione logica.
  - L'operazione di unificazione può utilizzata per isolare e accedere alle componenti di un albero.
  - Le operazioni sugli alberi sono realizzabili come operazioni su termini (e utilizzando le primitive per operare sui termini).
  - Non sempre le strutture sono facilmente leggibili.
  - L'accesso alle componenti di un albero può avvenire solamente percorrendo l'albero a partire dalla radice.

- Rappresentazione come insieme di fatti:
  - L'accesso ad ogni nodo (e ai suoi figli) è diretto.
  - Il padre e l'insieme dei figli di un nodo possono determinati direttamente ed efficientemente.
  - La visita e il percorrimento di un albero possono effettuati a partire da ogni nodo epossono avvenire tanto da un nodo verso i suoi antenati quanto verso i suoi discendenti.
  - La rappresentazione può facilmente generalizzata al caso della rappresentazione di grafi qualsiasi (come vedremo più avanti).
  - Le sottostrutture (i sottoalberi di un nodo, ad esempio) sono rappresentate in modo implicito enon possono isolate direttamente.
  - La modifica di un albero può un'operazione complessa e deve effettuata

mediante operazioni di rimozione e aggiunta di clausole al data-base.

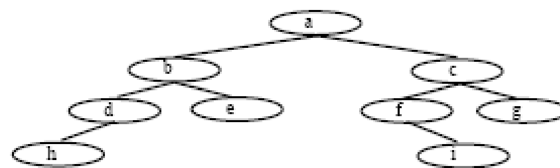
OPERAZIONI SU ALBERI BINARI

Visita di un albero binario
Percorrimento dell'albero per effettuare una operazione (o verificare una condizione) su ognuno dei nodi dell'albero stesso.

Esistono per lo meno tre strategie differenti per effettuare la visita di un albero.

- ✓- strategia "*pre-order*"; il nodo N viene visitato per primo, quindi vengono visitati (nell'ordine) i sottoalberi sinistro e destro (se esistono).
- ✓ strategia "*in-order*"; in primo luogo viene visitato il sottoalbero sinistro (se esiste) di N, quindi N e infine il suo sottoalbero destro.
- ✓ strategia "*post-order*"; vengono visitati dapprima i sottoalberi sinistro e destro di N (se esistono e nell'ordine) e quindi il nodo N stesso.

Si consideri l' albero binario seguente:



- strategia "pre-order": a, b, d, h, e, c, f, i, g

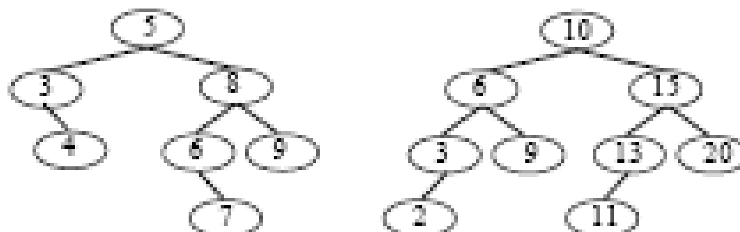
- strategia "in-order": h, d, b, e, a, f, i, c, g

- strategia "post-order": h, d, e, b, i, f, g, c, a

La strategia "pre-order" prende spesso anche il nome di strategia di visita in profondità.

Alberi binari di ricerca

Un albero binario T in cui i nomi dei nodi sono elementi di un insieme ordinato (secondo la relazione "<"). Si dice che T è un *albero di ricerca* se e solo se per ogni nodo N in T vale la seguente proprietà: se M e P sono rispettivamente il figlio sinistro e destro di N si ha che M = N < P



La verifica di appartenenza di un elemento all'insieme può effettuata efficientemente nel

modo seguente (*ricerca binaria*):

- se l'albero è vuoto si ha un fallimento;
- altrimenti si confronta l'elemento da cercare con la radice dell'albero e
  - se gli elementi sono uguali ci si ferma con un successo;
  - se l'elemento da cercare è minore della radice, si prosegue la ricerca nel sottoalbero sinistro;
  - se l'elemento da cercare è maggiore della radice, si prosegue la ricerca nel sottoalbero destro.

VERIFICA DI APPARTENENZA A UN ALBERO DI RICERCA

Sia dato un albero di ricerca i cui nodi sono numeri interi e rappresentato in Prolog mediante una lista.
Procedura Prolog che verifica l'appartenenza di un elemento all'albero.

appartiene(ELEM,TREE) "ELEM appartiene all'albero binario di ricerca TREE"

```
appartiene(ELEM, [ELEM, LEFT, RIGHT]) :- !.
appartiene(ELEM, [NODE, LEFT, RIGHT]) :- ELEM < NODE, !, appartiene(ELEM, LEFT).
appartiene(ELEM, [NODE, LEFT, RIGHT]) :- ELEM > NODE, appartiene(ELEM, RIGHT).
```

AGGIUNTA DI UN ELEMENTO AD A UN ALBERO DI RICERCA

Sia dato un albero binario di ricerca i cui nodi sono numeri interi e sia rappresentato in Prolog mediante una lista.

Si scriva una procedura Prolog che aggiunge un elemento all'albero costruendo un nuovo albero di ricerca:

aggiungi(ELEM, TREE, NEWTREE)
"NEWTREE è l'albero di ricerca che contiene gli elementi dell'albero di ricerca TREE e l'elemento ELEM"

```
aggiungi(ELEM, nil, [ELEM, nil, nil]).
aggiungi(ELEM, [NODE, L, R], [NODE, NEWL, R]) :- ELEM =< NODE, !, aggiungi(ELEM, L,
NEWL).
aggiungi(ELEM,[NODE,L,R],[NODE,L, NEWR]) :- ELEM > NODE, aggiungi(ELEM, R, NEWR).
```

PROLOG E INTELLIGENZA ARTIFICIALE CARATTERISTICHE DI UN LINGUAGGIO PER INTELLIGENZA ARTIFICIALE

Serie di requisiti per un linguaggio di programmazione per l'I.A. sulla base dei quali il Prolog risulta essere particolarmente adeguato:

- *Non determinismo*.
- *Capacità di elaborazione simbolica*.
- *Pattern Matching*.
- *Uniformità di programmi e dati*.
- *Rappresentazione della conoscenza e ragionamento*.
- *Uso interattivo del linguaggio (interpretato)*.
- *Prototipazione rapida* ("rapid prototyping").

RISOLUZIONE DI PROBLEMI

Si è interessati a quei problemi la cui soluzione può essere costruita attraverso un processo di ricerca.
Vi sono due tecniche fondamentali per la risoluzione dei problemi sviluppate nel campo dell'I.A.:
- Risoluzione mediante ricerca nello spazio degli stati;
- Risoluzione mediante scomposizione in sottoproblemi.

RISOLUZIONE NELLO SPAZIO DEGLI STATI
L'idea alla base di tale approccio è che un problema può essere rappresentato mediante:
• *Un insieme di stati*.
• *Un insieme di operatori di trasformazione di stato*.

L'insieme degli stati e degli operatori di un problema definiscono lo *spazio degli stati* del problema stesso. Tale spazio può essere rappresentato come un grafo orientato in cui i nodi sono gli stati del problema e gli archi orientati sono gli operatori di trasformazione.

Una particolare istanza di un problema può essere allora definita come una coppia di stati:
- *stato iniziale*;
- *stato finale* (o insieme di stati finali) o obiettivo (goal) che si vuole raggiungere.

Una soluzione del problema è una sequenza di operatori che permette di passare dallo stato iniziale ad uno stato finale.
Determinare una soluzione significa determinare nel grafo che rappresenta lo spazio degli stati un cammino che congiunge lo stato iniziale ad uno stato finale.