Linguaggi di Programmazione

UniShare

Davide Cozzi
@dlcgold

Gabriele De Rosa
@derogab

Federica Di Lauro
@f_dila

Indice

Capitolo 1

Introduzione

Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`.
Grazie mille e buono studio!

Capitolo 2

I linguaggi

Si possono classificare in 3 gruppi i linguaggi di programmazione:

1. **Linguaggi imperativi**, come *C, Assembler, Python etc....* Le caratteristiche dei linguaggi imperativi sono legate all'architettura di Von Neumann, composta da una componente passiva (la memoria) e una attiva (il processore). Il processore esegue calcoli e assegna valori a varie celle di memoria. Si ha quindi il concetto di *astrazione*. Una variabile non è altro che un'astrazione di una cella di memoria fisica. Ogni linguaggio ha diversi livelli di astrazione dell'archiettura di Von Neumann (che ricordiamo usare il "ciclo" formato da *Fetch instruction, Execute e Store result*), con i cosiddetti linguaggi di *alto* e *basso* livello. Possono essere sia linguaggi compilati (come *C*) che interpretati (come *Python*).
   I linguaggi imperativi usano quindi il *Paradigma Imperativo*, detto anche *Procedurale*. In questo paradigma si adotta uno *stile prescrittivo*, si prescrivono infatti operazioni che il processore deve eseguire e le istruzioni vengono eseguite in ordine, al più di strutture di controllo, e per questo è il miglior paradigma per rappresentare gli algoritmi. Questi linguaggi sono tra i più vecchi e tutt'ora tra i più usati soprattutto per la manipolazione numerica. Si ha la seguente formula che ben descrive il paradigma imperativo:

   $$Programma = Algoritmi + Strutture\ Dati$$

   In un linguaggio imperativo si ha sia una parte dedicata alla dichiarazione di variabili che una parte dedicata agli algoritmi risolutivi del problema. Inoltre le istruzioni possono essere così divise:

   - istruzioni di I/O

- istruzioni di assegnamento
- istruzioni di controllo

La ricerca di gestire applicazioni ancora a più alto livello con codice più conciso e semplice, che affrontano i problemi in maniera più logica (o comunque in maniera differente) ha portato alla nascita di altri paradigmi. Linguaggi logici e funzionali sono accomunati dall'essere di altissimo livello, dall'essere generati per manipolazione simbolica e non numerica, dal non distinguere perfettamente programma e strutture dati, dall'essere basati su concetti matematici e sull'adottare uno *stile dichiarativo*

2. **Linguaggi a oggetti**, come *C++* utilizzano il paradigma ad oggetti con l'uso di classi etc. Non vengono affrontati nel corso.

3. **Linguaggi Logici**, come il *Prolog*. Si basano sul concetto della deduzione logica e hanno come base la logica formale e come obbiettivo la formalizzazione del ragionamento. Programmare con un linguaggio logico significa descrivere un problema con frasi del linguaggio (ovvero con formule logiche) e interrogare il sistema che effettua deduzioni sulla base della conoscenza rappresentata. Il paradigma logico si può rappresentare con la seguente formula:

$$Programma = Conoscenza + Controllo$$

Si ha uno *stile dichiarativo* in quanto la conoscenza del problema è espressa indipendentemente dal suo utilizzo (si usa il **cosa** e non il **come**). Si ha quindi un'alta modularità e flessibilità ma si ha la problematica della rappresentazione della conoscenza, infatti definire un linguaggio logico significa definire come il programmatore può esprimere la conoscenza e quale tipo di controllo si può utilizzare nel processo di deduzione.
Analizziamo le basi del Prolog:

- dopo ogni asserzione si mette un . mentre le **,** sono degli *and* logici
- le costanti si indicano in minuscolo e le variabili in maiuscolo
- **Asserzioni Incondizionate (fatti)** così indicate:

  ```
  A.
  ```

- **Asserzioni Condizionate (regole)**, che ricordiamo non essere regole di inferenza, così indicate:

```
A :- B, C, D, ..., Z.
```

dove A è il *conseguente*, ovvero la conclusione, mentre le altre sono gli antecedenti, ovvero le premesse. Il simbolo ":-" è un implica che per ragioni di interprete si legge al contrario rispetto al solito: seguendo l'esempio si ha che *B, C, D, ..., Z implicano A* ovvero $B, C, D, ..., Z \rightarrow A$

4. **Interrogazione**, che rappresenta l'input utente, è così espressa:

   ```
   :- K, L, M, ..., P.
   ```

   e indica che si chiede cosa implicano quei dati antecedenti.

   Vediamo un esempio più completo (anche se non del tutto):
   due individui sono colleghi se lavorano per la stessa ditta:

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

   lavora(ciro, ibm).
   lavora(ugo, ibm).
   lavora(olivia, samsung).
   lavora(ernesto, olivetti).
   lavora(enrica, samsung).

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

   dove la prima asserzione rappresenta la regola, le successive 5 i fatti e l'ultima riga è l'interrogazione. Il programma non è completo in quanto non si definisce concretamente *diverso*. La logica di risoluzione è la seguente: L'interprete cerca un X e un Y (che sono variabili) in grado di rappresentare quella regola, infatti l'interrogazione è la conseguenza, e li cerca tra i fatti partendo dal primo ("ciro, ibm") che viene messo come X = ciro e Z = ibm. Parte il confronto con se stesso (si ha tanto la funzione *diverso*) e con gli altri (che mano a mano diventeranno gli Y e Z del secondo lavora) dando alla fine come risultato solo i colleghi cercati.

5. **Linguaggi funzionali:**, come *Lisp* i suoi "dialetti" come *Common Lisp*, hanno come concetto primitivo la *funzione* che è una regola di associazione tra due insiemi (dominio e codominio). La regola di una

funzione ne specifica dominio, codominio e regola di associazione. Una funzione può essere applicata ad un elemento del dominio (detto *argomento*) per restituire l'elemento del codominio associato (mediante il processo di *valutazione* o *esecuzione*). Nel paradigma funzionale puro l'unica applicazione è l'applicazione di funzioni e il ruolo dell'esecutore si esaurisce nel valutare l'applicazione di una funzione e produrre un valore. In questo paradigma "puro" il valore di una funzione è determinato soltanto dal valore degli argomenti che riceve al momento della sua applicazione, e non dallo stato del sistema rappresentato dall'insieme complessivo dei valori associati a variabili (e/o locazioni di memoria) in quel momento, comportando l'assenza di effetti collaterali. Il concetto di variabile è qui quello di *costante matematica* con valori immutabili (non si ha l'operazione di assegnamento). La programmazione funzionale consiste nel combinare funzioni mediante composizioni e utilizzare la **ricorsione**. Il paradigma è ben rappresentato da questa formula:

$$\textit{Programma} = \textit{Composizione di Funzioni} + \textit{Ricorsione}$$

Si ha quindi un insieme di funzioni mutualmente ricorsive e l'esecuzione del programma consiste nella valutazione dell'applicazione di una funzione principale a degli argomenti.
Il linguaggio *Lisp*, inizialmente proposto da John McCarthy nel '58 era un linguaggio funzionale puro. Si sono poi sviluppati molti ambienti di programmazione Lisp come: *Common Lisp, Scheme e Emacs Lisp.*
Si analizza un esempio di codice in *Lisp*: *Controllare se un elemento (item) appartiene ad un insieme (rappresentato con una lista);*

```
(defun member (item list)
^^I(cond((null list) nil)
^^I^^I((equal item (first list)) T)
^^I^^I(T (member item (rest list)))))
(member 42 (list 12 34 42))
```

Si ha che tutto è rappresentato da una lista, si hanno delle funzioni standard (*defun, equal, first, rest e list*) e *member* che viene definita dal programmatore. L'ultima linea definisce il numero da cercare e la lista, sempre con la logica di funzioni dentro ad altre. L'esecuzione è la seguente: Si definisce, nella prima riga, la funzione che cerca un valore (*item*) in una lista (*list*). Nella seconda riga cominciano le condizioni:

(a) *cond* definisce una condizione su liste formate da coppie

(b) prima si controlla se la lista è nulla con *nil* (che sarebbe falso). Se *nil* si esce. Questo rappresenta anche il caso base della nostra funzione ricorsiva.

(c) si controlla se l'elemento cercato è uguale al primo della lista e con *T* si indica *true*. Se *T* si esce.

(d) con l'ultima parte si ha la vera e propria ricorsione, forzata da *T* iniziale, che ripete l'operazione togliendo ogni volta il primo elemento, facendo ricominciare i controlli con l'elemento successivo finché non si trova o non si ha il caso base della lista vuota

Si nota l'assenza di assegnamenti.

2.1 Richiami di Architettura e Programmazione

Per eseguire un programma in un qualsiasi linguaggio il sistema (ovvero il sistema operativo) deve mettere a disposizione un ambiente *run time*, che fornisca almeno due funzionalità:

- mantenimento dello stato della computazione (program counter, limiti di memoria etc)
- gestione della memoria disponibile (fisica e virtuale)

inoltre l'ambiente run time può essere una macchina virtuale (come la *JVM*, *Java Virtual Machine*, per Java).
La gestione della memoria avviene usando due aree concettualmente ben distinte con funzioni diverse:

- lo *Stack* dell'ambiente run time serve per la gestione delle chiamate (soprattutto ricorsive) a procedure, metodi, funzioni etc...
- lo *Heap* dell'ambiente run time serve per la gestione di strutture dati dinamiche (liste, alberi etc...)

I linguaggi logici e funzionali (ma anche Java) utilizzano pesantemente lo Heap dato che forniscono come strutture dati *built in* liste e, spesso, vettori di dimensione variabile.
La valutazione di procedure avviene mediante la costruzione (sullo stack di sistema) di *activation frames*. I parametri formali di una procedura vengono associati ai valori. Il corpo della procedura viene valutato (ricorsivamente) tenendo conto di questi legami in maniera *statica*. Ad ogni sotto-espressione del

corpo si sostituisce il valore che essa denota (computa). Il valore (valori) restituito dalla procedura in un'espressione return o con meccanismi analoghi è il valore del corpo della procedura (che non è altro che una sotto-espressione). Quando il valore finale viene ritornato i legami temporanei ai parametri formali spariscono (lo stack di sistema subisce una *pop* e l'activation frame viene rimosso).

Esempio 1. *Vediamo un banale esempio in C*

```
/* procedura/metodo */
int doppio(int x) {
^^Ireturn 2 * x;
}

/* main con la chiamata al metodo*/
^^Iint d=doppio(3)
```

la chiamata:

- *estende l'ambiente corrente, dove è stata dichiarata la variabile d, con quello locale che contiene i legami tra parametri formali e valori dei parametri attuali; un activation frame viene inserito in cima allo stack di valutazione*
- *valuta il corpo della procedura*
- *ripristina l'ambiente di partenza:*
  - *Il risultato della chiamata viene salvato nella variabile d*
  - *l'activation frame viene rimosso dalla cima dello stack di valutazione*

Per l'esecuzione di una procedura un programma deve eseguire i seguenti sei passi:

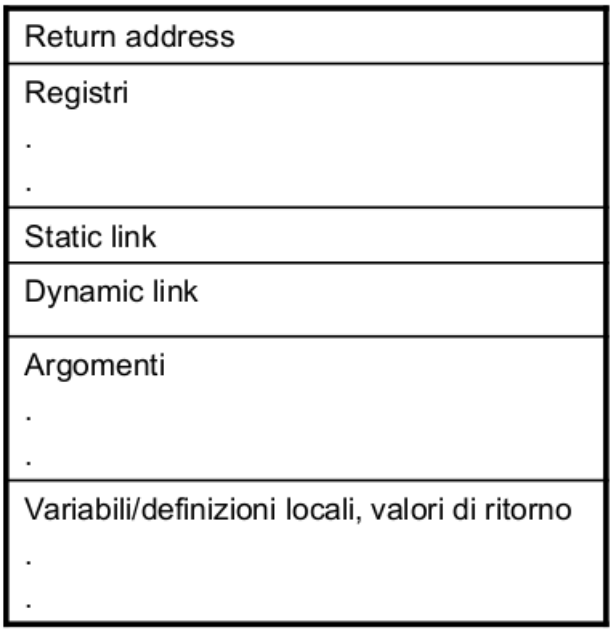
1. mettere i parametri in un posto dove la procedura possa recuperarli
2. trasferire il controllo alla procedura
3. allocare le risorse (di memorizzazione dei dati) necessarie alla procedura
4. effettuare la computazione della procedura
5. mettere i risultati in un posto accessibile al chiamante

6. restituire il controllo al chiamante

Queste operazioni agiscono sui registri a disposizione e sullo stack utilizzato dal runtime (esecutore) del linguaggio.
Lo spazio richiesto per salvare (sullo stack) tutte le informazioni necessarie all'esecuzione di una procedura ed al ripristino dello stato precedente alla chiamata è quindi costituito da:

- Spazio per i registri da salvare prima della chiamata di una sotto procedura
- Spazio per l'indirizzo di ritorno (nel codice del corpo della procedura)
- Spazio per le variabili, definizioni locali, e valori di ritorno
- Spazio per i valori degli argomenti
- Spazio per il riferimento statico (*static link*)
- Spazio per il riferimento dinamico (*dynamic link*)
- Altro spazio dipendente dal particolare linguaggio e/o politiche di allocazione del compilatore

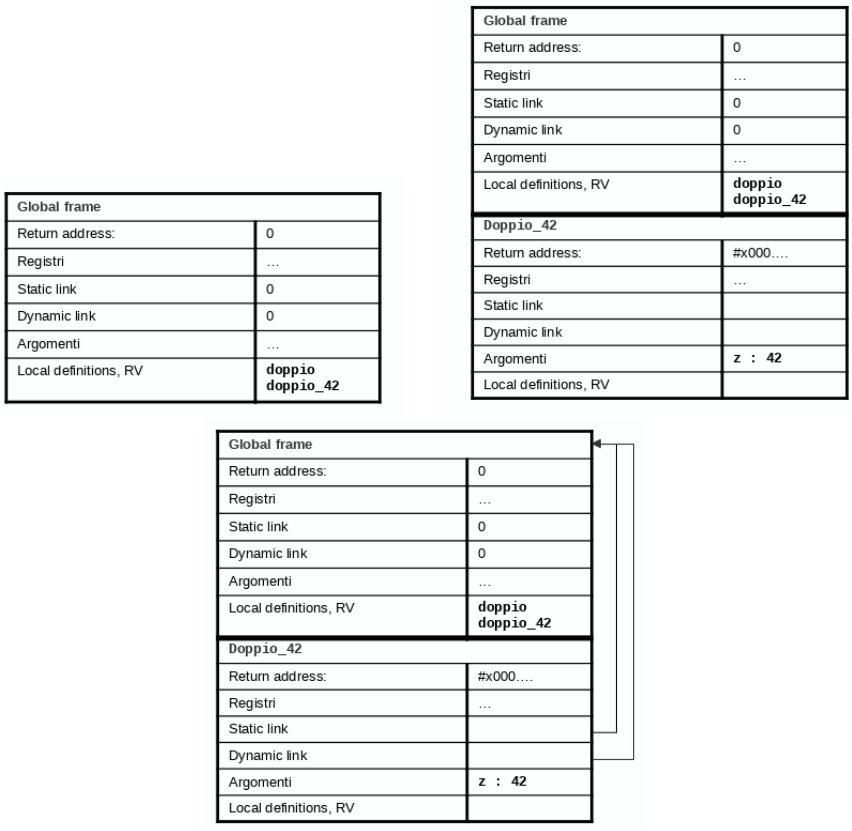
Analizziamo meglio l'*activation frame* di una procedura mediante la seguente immagine:



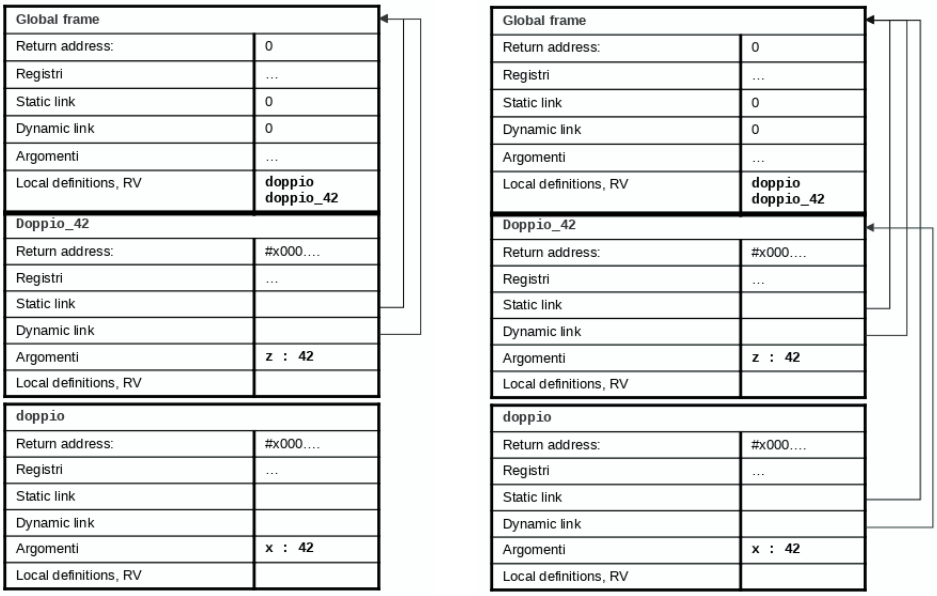
Partiamo ora dal seguente codice in *C*, e analizziamo i passaggi che avvengono nello stack:

```
/* prima procedura */
int doppio(x) {
^^Ireturn 2 * x;
}

/* seconda procedura, che chiama la prima */
int doppio_42(z) {
^^Ireturn doppio(z) - 42;
}

/* chiamata nel main */
int r = doppio_42(42)
```

vediamo cosa accade nello stack: si definiscono in ordine *global frame* e *doppio_42*, con i collegamenti di static link e dynamic link:



Si aggiunge poi *doppio*, con i collegamenti di static link e dynamic link:



Global frame	
Return address:	0
Registri	…
Static link	0
Dynamic link	0
Argomenti	…
Local definitions, RV	**doppio** **doppio_42**

Doppio_42	
Return address:	#x000....
Registri	…
Static link	
Dynamic link	
Argomenti	**z : 42**
Local definitions, RV	

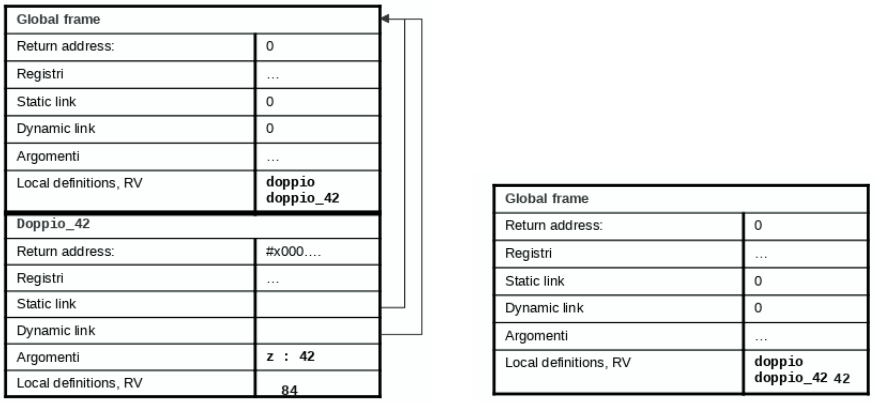
doppio	
Return address:	#x000....
Registri	…
Static link	
Dynamic link	
Argomenti	**x : 42**
Local definitions, RV	

Global frame	
Return address:	0
Registri	…
Static link	0
Dynamic link	0
Argomenti	…
Local definitions, RV	**doppio** **doppio_42**

Doppio_42	
Return address:	#x000....
Registri	…
Static link	
Dynamic link	
Argomenti	**z : 42**
Local definitions, RV	

doppio	
Return address:	#x000....
Registri	…
Static link	
Dynamic link	
Argomenti	**x : 42**
Local definitions, RV	

infine vengono risolti a partire da *doppio*, salvando ogni volta i vari risultati nelle *local definitions*:



Global frame	
Return address:	0
Registri	…
Static link	0
Dynamic link	0
Argomenti	…
Local definitions, RV	**doppio** **doppio_42**

Doppio_42	
Return address:	#x000....
Registri	…
Static link	
Dynamic link	
Argomenti	**z : 42**
Local definitions, RV	**84**

Global frame	
Return address:	0
Registri	…
Static link	0
Dynamic link	0
Argomenti	…
Local definitions, RV	**doppio** **doppio_42 42**

L'area di memoria per la manipolazione di strutture dati dinamiche verrà spiegata meglio al momento dell'introduzione delle primitive per la costruzione delle liste in Prolog e Lisp.
Il componente software che si occupa della gestione automatica della memoria in *Lisp, Prolog, Java, C#*, ed altri linguaggi è il *garbage collector* (il servizio di raccolta rifiuti). *C* e *C++* non hanno un garbage collector standard.

2.2 Logica

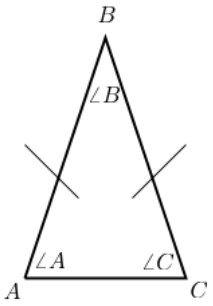
Si ricordano le seguenti due definizioni:

1. **Implicazione**, indicata dal simbolo $\rightarrow$, è un connettivo logico che si usa per costruire forme logiche complesse. In una frase il *se* comporta un'implicazione logica.

2. **Inferenza** è un meccanismo per il ragionamento. Detta anche *modus ponens*, è una manipolazione sintattica (quindi non rappresentabile con tabelle di verità)

Partiamo con l'esempio di una semplice dimostrazione geometrica effettuata con le regole della logica:

Teorema 1. *Dato un triangolo isoscele (con $\overline{AB} = \overline{BC}$) si ha che $\angle A$, ovvero l'angolo in A, e $\angle C$, ovvero l'angolo in C, sono uguali.*

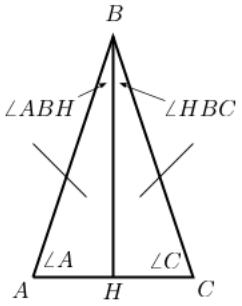


Dimostrazione. Si comincia la dimostrazione con l'elenco delle conoscenze pregresse:

1. se due triangoli sono uguali essi hanno lati e angoli uguali

2. se due triangoli hanno due lati e l'angolo sotteso uguale allora i due triangoli sono uguali

3. se viene definita la bisettrice di $\angle B$, $\overline{BH}$, si ha che $\angle ABH = \angle HBC$



Procediamo ora coi passi della dimostrazione:

1. $\overline{AB} = \overline{BC}$ per ipotesi
2. $\angle ABH = \angle HBC$ per la terza conoscenza pregressa
3. $\triangle HBC = \triangle ABH$ per la seconda conoscenza pregressa (dal secondo passo si ha che l'angolo tra la bisettrice, per forza uguale tra i due triangoli, e i due lati obliqui, uguali per ipotesi, è uguale per entrambi)
4. $\angle A = \angle C$ per la prima conoscenza pregressa (essendo uguali i triangoli per il passo precedente si ha che anche gli angoli sono uguali)

Siamo così giunti alla fine della dimostrazione. Bisogna ora rappresentarla con gli strumenti della logica.
Sono stati svolti i seguenti passaggi:

- si è trasformata la seconda conoscenza pregressa in:
  se $\overline{AB} = \overline{BC}$ **e** $\overline{BH} = \overline{BH}$ **e** $\angle ABH = \angle HBC$ **allora** $\triangle ABH = \triangle HBC$
- si è trasformata la prima conoscenza pregressa in:
  se $\triangle ABH = \triangle HBC$ **allora** $\overline{AB} = \overline{BC}$ **e** $\overline{BH} = \overline{BH}$ **e** $\overline{AH} = \overline{HC}$ **e** $\angle ABH = \angle HBC$ **e** $\angle AHB = \angle CBH$ **e** $\angle A = \angle C$

Possiamo ora procedere col processo di **formalizzazione**, ovvero razionalizzare il processo per poter affermare:

$$\overline{AB} = \overline{BC} \vdash \angle A = \angle C$$

con $\vdash$ che è il simbolo di *derivazione logica* e significa "consegue", "allora" etc...
Quindi per ottenere:

$$\overline{AB} = \overline{BC} \vdash \angle A = \angle C$$

abbiamo assunto:

$$P = \{\overline{AB} = \overline{BC}, \angle ABH = \angle HBC, \overline{BH} = \overline{BH}\}$$

con queste conoscenze pregresse:

1. $\overline{AB} = \overline{BC} \wedge \overline{BH} = \overline{BH} \wedge \angle ABH = \angle HBC \rightarrow \triangle ABH = \triangle HBC$
2. $\triangle ABH = \triangle HBC \rightarrow \overline{AB} = \overline{BC} \wedge \overline{BH} = \overline{BH} \wedge \overline{AH} = \overline{HC} \wedge \angle ABH = \angle HBC \wedge \angle AHB = \angle CBH \wedge \angle A = \angle C$

Si ha quindi una catena di formule (**P** sono le ipotesi iniziali):

1. **P1:** $\overline{AB} = \overline{BC}$ preso da **P**
2. **P2:** $\angle ABH = \angle HBC$ preso da **P**
3. **P3:** $\overline{BH} = \overline{BH}$ preso da **P**
4. **P4:** $\overline{AB} = \overline{BC} \wedge \overline{BH} = \overline{BH} \wedge \angle ABH = \angle HBC$ preso da **P1, P2, P3** e dalla regola di inferenza detta **introduzione della congiunzione**
5. **P5:** $\triangle ABH = \triangle HBC$ da **P4**, dalla **regola 2** (*se due triangoli hanno due lati e l'angolo sotteso uguale allora i due triangoli sono uguali*) e dalla regola di inferenza detta **modus ponens**
6. **P6:** $\overline{AB} = \overline{BC} \wedge \overline{BH} = \overline{BH} \wedge \overline{AH} = \overline{HC} \wedge \angle ABH = \angle HBC \wedge \angle AHB = \angle CBH \wedge \angle A = \angle C$ da **P5**, dalla **regola 1** (*se due triangoli sono uguali essi hanno lati e angoli uguali*) e dalla regola di inferenza detta **modus ponens**
7. **P7:** $\angle A = \angle C$ da **P6** e dalla regola d'inferenza detta *eliminazione della congiunzione* (date più asserzioni collegate da $\wedge$ posso eliderne un numero a piacere)

Abbiamo così dimostrato tutto anche per mezzo dei costrutti della logica $\square$

Definizione 1. *Una dimostrazione del tipo F è conseguenza di S,* ***dim****, si indica con:*

$$S \vdash F$$

ed è una sequenza:

$$dim =< P_1, P_2, ..., P_n >$$

con:

- $P_n = F$
- $P_i \in S$ *o con* P_i *ottenibile dalle* $P_1, ..., P_{i-1}$ *applicando una regola di inferenza*

Un insieme di regole di inferenza costituisce la base di un calcolo logico e diversi insiemi di regole danno vita a diversi calcoli logici. Un calcolo logico ha lo scopo di manipolare le formule in modo unicamente sintattico, stabilendo una connessione tra un'insieme di formule di partenza, dette *assiomi*, e un insieme di conclusioni.

2.2.1 Logica Proporzionale

La logica proposizionale si occupa delle conclusioni che si possono trarre da un insieme di proposizioni, che definiscono *sintatticamente* la logica proposizionale (l'ultima parte della dimostrazione sopra è fatta da costrutti della logica proposizionale).
All'insieme **P** è associata una funzione di *verità*, o di *valutazione*, **V** (spesso indicata con **T** o **I**):

$$V : P \to \{vero,\ falso\}$$

che associa un valore di verità ad ogni eleemnto di **P**. La funzione di valutazione è il ponte di connessione tra sintassi e semantica in un linguaggio logico.
Le proposizione si combinano con i seguenti connettivi:

- **congiunzione:** $\wedge$
- **disgiunzione:** $\vee$
- **negazione:** $\neg$
- **implicazione:** $\to$

L'insieme di tutte le formule formate dagli elementi di **P** e dalle loro combinazioni è detto: **Formule Ben Formate (FBF)**.
Le formule atomiche e i loro negativi vengono detti **letterali**. Il valore di verità di proposizione dipende dalla funzione di verità **V** e questa definizione può essere estesa sul dominio **FBF**:

- $V(\neg s) = non\, V(s)$
- $V(a \wedge b) = V(a)\ \ e\ V(b)$
- $V(a \vee b) = V(a)\ \ o\ V(b)$
- $V(a \to b) = (non\, V(a))\ \ o\ V(b)$

ovvero, secondo la tavola di verità (con $1 = vero$ e $0 = falso$):

a	b	$\neg a$	$\neg b$	$a \wedge b$	$a \vee b$	$a \to b$
1	1	0	0	1	1	1
1	0	0	1	0	1	1
0	1	1	0	0	1	0
0	0	1	1	0	0	1

La tavola di verità costituisce la semantica di un insieme di proposizioni mentre un calcolo logico dice come generare nuove formule logiche, ovvero espressioni sintattiche, a partire dagli assiomi. Questo calcolo deve garantire che le nuove formule generate siano vere se gli assiomi sono veri. Questo processo di generazione si chiama **dimostrazione**.
Per ottenere nuove formule dagli assiomi si usa il calcolo proposizionale, che si base su regole di inferenza. Una regola di inferenza ha la seguente forma:

$$\frac{F_1, F_2, ..., F_N}{R} \text{ [nome regola]}$$

con F_i formula vera in **FBF** e R è la formula vera generata da inserire in **FBF**. Vediamo qualche esempio:

Esempio 2 (Modus Ponens)**.**

$$\frac{a \to b,\ a}{b}$$

ovvero:

- *Se piove, la strada è bagnata*
- *Piove*
- *Allora la strada è bagnata*

Ovvero, la regola sintattica del modus ponens ci permette di aggiungere le conclusioni di una "regola" al nostro insieme di formule ben formate "vere"

Esempio 3 (Modus Tollens)**.**

$$\frac{a \to b, \neg b}{\neg a}$$

ovvero:

- *Se piove, la strada è bagnata*
- *La strada non è bagnata*
- *Allora non piove*

Ovvero, la regola sintattica del modus tollens ci permette di aggiungere la premessa negata di una "regola" al nostro insieme di formule ben formate "vere"

Esempio 4 (Eliminazione e Introduzione di $\wedge$)**.**

$$\frac{P_1 \wedge P_2 \wedge ... \wedge P_n}{P_i} \; [Eliminazione \; di \; \wedge]$$

ovvero:

- *Piove e la strada è bagnata*
- *Piove*

$$\frac{P_1, P_2, ..., P_n}{P_1 \wedge P_2 \wedge ... \wedge P_n} \; [Introduzione \; di \; \wedge]$$

ovvero:

- *Piove*
- *La strada è bagnata*
- *Piove e la strada è bagnata*

Ovvero, la regola sintattica dell'eliminazione della congiunzione ci permette di aggiungere all'insieme ***FBF*** *i singoli componenti di una formula complessa.*
Queste regole si chiamano anche, rispettivamente, di congiunzione e di disgiunzione

Esempio 5 (Introduzione di $\vee$)**.**

$$\frac{a}{a \vee b}$$

ovvero:

- *Piove*
- *Piove o c'è vita su Marte*

Ovvero, la regola sintattica dell'introduzione della disgiunzione ci permette di aggiungere i singoli componenti di una formula.
Questa regola è detta anche di addizione complessa

Esempio 6. *Ecco altre regole utili:*

- ***Terzo Escluso:***

$$\frac{a \vee \neg a}{vero}$$

- ***Eliminazione di*** $\neg$***:***
$$\frac{\neg\neg a}{a}$$
- ***Eliminazione di*** $\wedge$***:***
$$\frac{a \wedge vero}{a}$$
- ***Contraddizione:***
$$\frac{a \wedge \neg a}{b}$$
ovvero da una contraddizione posso trarre qualsiasi conseguenza

Queste regole di inferenza fanno parte del *calcolo naturale*, detto anche *di Gentzen*, ovvero il loro formalizzatore. Questo tipo di calcolo consiste nel formalizzare i modi di derivare conclusioni a partire dalle premesse, ovvero di derivare direttamente un FBF mediante una sequenza di passi ben codificati. La regola del *modus ponens* (ovvero l'eliminazione dell'implicazione), insieme al principio del terzo escluso, posso essere usati anche *procedendo per assurdo* alla dimostrazione di una data formula; questa procedura è detta *principio di risoluzione*. Il *principio di risoluzione* è una regola di inferenza generalizzata semplice e facile da utilizzare e implementare. Questo principio lavora su FBF trasformate in *forma normale congiunta*, dove ogni congiunto è detto *clausola*. L'osservazione alla base del principio è un'estensione della rimozione dell'implicazione sulla base del principio di contraddizione e si usa per dimostrazioni per assurdo:

$$\frac{p \vee r,\ s \vee r}{p \vee s} \qquad \frac{\neg r,\ r}{\bot}$$

dove:

- $p \vee s$ è la *clausola risolvente*
- $\bot$ è la *clausola vuota*, che corrisponde all'aver creato una contraddizione con delle FBF (posso infatti dedurre qualsiasi cosa, anche la clausola vuota)

vediamo un'altra regola di inferenza:

Esempio 7 ((unit) resolution)**.**

$$\frac{\neg p,\ q_1 \vee q_2 \vee ... \vee q_k \vee p}{q_1 \vee q_2 \vee ... \vee q_k}$$

o anche:

$$\frac{p,\ q_1 \vee q_2 \vee ... \vee q_k \vee \neg p}{q_1 \vee q_2 \vee ... \vee q_k}$$

è una regola di risoluzione molto generale e se una delle due clausole da risolvere è un letterale (come negli esempi) si parla di unit resolution. Come esempio di può avere:

- *non piove, piove e c'è il sole*
- *c'è il sole*

Vediamo ora come funzionano le dimostrazioni per assurdo.
Si supponga di avere un'insieme di FBF vere e di avere una proposizione p da dimostrare vera. Si ha il metodo *reductio ad absurdum*:

- assumo $\neg p$ vera
- se combinandola con le FBF ottengo una contraddizione allora p deve essere vera (o meglio derivabile)

Esempio 8 (provo che q è vera)**.** *assumo:*

- $FBF = \{p \to q,\ p,\ \neg w,\ e,\ r\}$
- *assumo vera* $\neg q$

si ha che $FBF \cup \{\neg q\}$ *genera una contraddizione infatti:*

- $p \to q \equiv \neg p \vee q$ *combinato con* p *produce* q *e quindi si ha* $q \vee \neg q$*, ovvero una contraddizione*
- *se applico a* q *e* $\neg q$ *il principio di risoluzione ottengo la clausola vuota* $\perp$

Torniamo ad analizzare gli assiomi. Si hanno delle proposizioni sempre vere:

- $a \to (b \to a)$
- $(a \to (b \to c)) \to ((a \to b) \to (a \to c))$

- $(\neg b \to \neg a) \to ((\neg b \to a) \to b)$
- $\neg(a \wedge \neg a)$ detto *principio di non contraddizione*
- $a \vee \neg a$ detto *principio* del terzo escluso

Vediamo un esempio:

Esempio 9. *Si ha la seguente affermazione:*
se l'unicorno è mitico, allora è immortale, ma se non è mitico allora è mortale. Se è mortale o immortale, allora è cornuto. L'unicorno è magico se è cornuto. Ho le seguenti domande:

1. *l'unicorno è mitico?*
2. *l'unicorno è magico?*
3. *l'unicorno è cornuto?*

Bisogna quindi esprimere il problema con la logica delle proposizioni, individuare teoremi da dimostrare e infine dimostrarli. Le proposizioni saranno quindi:

- UM = *unicordo è mitico*
- UI = *unicorno è immortale*
- $UMag$ = *unicorno è magico*
- UC = *unicorno è cornuto*

che trascritto in logica diventa, che sono le mie conoscenze pregresse:

$$UM \to UI$$
$$\neg UM \to \neg UI$$
$$\neg UI \vee UI \to UC$$
$$UC \to Umag$$

dimostro ora le domande rappresentandole in forma logica. Si ha:

$$S = \{UM \to UI,\ \neg UM \to \neg UI,\ \neg UI \vee UI \to UC,\ UC \to Umag\}$$

con le domande che diventano:

- $S \vdash UM$*?*
- $S \vdash Umag$*?*
- $S \vdash UC$*?*

Inizio a dimostrare la terza:

$$P1 : \neg UI \vee UI \rightarrow UC \quad da\ S$$

$$P2 : \neg UI \vee UI \ da\ A5 \textit{(tautologia)}$$

$$P3 : UC \ da P2,\ P2\ e\ ModusPonens$$

$P3$ si può scrivere già perché $P2$ è una tautologia, quindi posso utilizzare il modus ponens.
Ecco la seconda:

$$P4 : UC \rightarrow Umag \ da\ S$$

$$P5 : Umag \ da\ P3,\ P4\ e\ ModusPonens$$

Ed ecco la prima:
non è dimostrabile con queste conoscenze pregresse, non posso usare nessuna regola di inferenza

Si hanno differenze tra *sintassi* e *semantica*. L'operazione di derivazione $\vdash$ è un operatore sintattico e il calcolo logico fornisce una manipolazione sintattica e simbolica.
La semantica dipende dalla funzione di interpretazione, o funzione di valutazione o funzione di verità, V, che si basa sulle tabelle di verità. Si ha l'operatore di *conseguenza logica* $\models$. Si ha, data una particolare logica che:

$$S \vdash f \text{ se e solo se } S \models f$$

Con S insieme di formule iniziale e f una FBF, il tutto in dipendenza da una funzione di verità V.
Una FBF vera indipendentemente dai valori dei letterali è detta *tautologia*.
Una particolare interpretazione V che rende vere tutte le formule di S è detta *modello* di S.

2.2.2 Logica del primo ordine

Di cui fanno parte i sillogismi (tutti gli uomini sono mortali, Socrate è mortale quindi Socrate è mortale). Si ha la *logica proposizionale*, computazionalmente e semanticamente più chiara, ma limitata nella difficoltà di rappresentare asserzioni circa insiemi di elementi in maniera concisa. Nel sillogismo sopra la prima frase non è esprimibile in logica proposizionale. Si hanno le seguenti nozioni nella logica del primo ordine (LPO):

- variabile
- costante
- relazione (o predicato), che è un sottoinsieme di un prodotto cartesiano
- funzione
- quantificatore

quindi un linguaggio del primo ordine è costituito da *termini* costruiti a partire da:

- V insieme di simboli di variabili
- C insieme di simboli di costante
- R insieme di simboli di relazione o predicati di varia arità
- F insieme fi funzioni di varia arità
- connettivi logici e simboli di quantificazione universale, $\forall$ (universale, "per ogni") e $\exists$(esistenziale, "esiste")

La costruzione di un linguaggio di primo ordine è *ricorsiva*. I termini più semplici sono predicati

$$r \subseteq C_0 \times C_1 \times \cdots \times C_n$$

ovvero le relazioni cartesiane su C, scritte come $r(c_1, ..., c_n)$.
Le funzioni sono definite con dominio e codominio:

$$f : C_0 \times C_1 \times \cdots \times C_n \to C$$

e una funzione si scrive come $f(c_1, ..., c_n)$.
Si hanno FBF costruite ricorsivamente da:

- un *termine*, ovvero un elemento di C, V o l'applicazione di una funzione $f(t_1, t_2, ...t_n)$
- un termine costruito da un predicato $r(t_1, t_2, ...t_n)$, con t_i termini, appartiene ad una FBF
- diversi elementi di FBF connessi da connettivi logici standard($\vee$, $\wedge$, $\neg$, $\rightarrow$) appartengono a FBF, tali combinazioni sono dette $t(t_1, t_2, ...t_n)$
- le formule $\forall x$, $t(t_1, t_2, ...t_n)$ e $\exists x$, $t(t_1, t_2, ...t_n)$ appartengono a FBF

Rivediamo ora il sillogismo *tutti gli uomini sono mortali, Socrate è mortale quindi Socrate è mortale.* Ho:

- l'insieme delle costanti individuali $C = \{Sorate, Platone, ...\}$
- l'insieme dei predicati $P = \{uomo, mortale\}$

traduco quindi le asserzioni principali del sillogismo:

$$\forall x, (uomo(x) \rightarrow mortale(x))$$

$$uomo(Socrate)$$

e traduco la conclusione:

$$mortale(socrate)$$

Bisogna giustificare la semantica con nuove regole.
Si aggiunge una nuova regola d'inferenza per la logica dei predicati, l'eliminazione del quantificatore universale $\forall$ (C insieme delle costanti):

$$\frac{\forall x, T(..., x, ...), c \in C}{T(..., c, ...)}$$

Quindi risolvo il sillogismo con questa regola:

$$\frac{(\forall x, uomo(x) \rightarrow mortale(x)), Socrate \in C}{uomo(Socrate) \rightarrow mortale(Socrate)}$$

poi applico il modus ponens, detto, per la logica proposizionale, anche *eliminazione dell'implica* $\rightarrow$:

$$\frac{uomo(Socrate), uomo(Socrate) \rightarrow mortale(Socrate)}{mortale(Socrate)}$$

Abbiamo altre regole di inferenza per il quantificatore esistenziale:

- Introduzione del quantificatore esistenziale $\exists$:

$$\frac{T(..., c, ...), c \in C}{\exists x, T(..., x, ...)}$$

- si hanno le seguente identità:

$$\exists x, \neg T(..., x, ...) \equiv \neg \forall x, T)..., x, ...)$$

$$\forall x, \neg T(..., x, ...) \equiv \neg \exists x, T(..., x, ...)$$

Capitolo 3

Prolog

Il prolog è un linguaggio logico rappresentato da una semplicità del formalismo, è di alto livello e ha una semantica chiara. Un programma è un insieme di formule e si ha come fine la dimostrazione di un'affermazione. La base formale è data dalle *clausole di Horn* e dal meccanismo di *risoluzione*. Prolog viene da *PROgramming LOGic* e si basa su una restrizione della logica del primo ordine e usa uno stile dichiarativo. Si vuole determinare la veridicità di una affermazione. Ogni FBF può essere in *forma normale a clausola*:

- *formula normale congiunta:* congiunzione di disgiunzioni o di negazione di predicati (sia positivi che negativi):

  $$\wedge_i(\vee_i L_{ij})$$

  Esempio 10. *ecco degli esempi:*

  - $(p(x) \vee q(x,y) \vee \neg t(z)) \wedge (p(w) \vee \neg s((u) \vee \neg r(v))$
  - $(\neg t(z)) \vee (p(w) \vee \neg s(u)) \wedge (p(x) \vee s(x) \vee q(y))$

  che sono riscrivibili come:

  $$t(z) \rightarrow p(x) \vee q(x,y)$$

  e

  $$s(u) \wedge r(v) \rightarrow p(w)$$

- *forma normale disgiunta*: disgiunzione di congiunzioni o di negazione di predicati (sia positivi che negativi)

  $$\vee_i(\wedge_i L_{ij})$$

Le clausole con un solo letterale positivo solo le *clausole di Horn* e un programma in prolog è una collezione di *clausole di Horn*. Non tutte le FBF possono essere *clausole di Horn*.
Prolog non ha istruzioni. Si hanno *fatti* (asserzioni vere nel contesto descritto) e *regole*.
Parliamo di sintassi:

- ":-" è l'implicazione
- tutto termina con un punto "."
- la "," è l'and logico
- il ";" è lor logico
- *a.* è un fatto o asserzione
- *b :- c, d, ...* è una regola (c, d, ... termini composti)
- *? :- x, y, ...* è un goal o una query (x, y, ... termini composti)
- non si dichiarano variabili e si ha solo la lista come struttura dati. SI hanno vari termini (atomi variabili etc...):
  - atomi (numeri, ...) ovvero caratteri alfanumerici che inizia col carattere minuscolo (puo esserci _ ma quindi non un numero), qualsiasi cosa dentro apici "" o numeri.
  - variabili è un carattere che inizia con la maiuscola (e quindi non con un numero) o col carattere _ (che indica una variabile anonima o *indifferenza*). Le variabili si istanziano (non con un assegnamento) col procedere del programma e della dimostrazione
  - termine composto (col simbolo +). Un *funtore* è un simbolo di funzione o predicato che applica ad argomenti, messi tra parentesi tonde (da non spaziare dal funtore). Il funtore è minuscolo e non è un numero
- un *fatto* è un nome di predicato che inizia con la maiuscola
- una *regola* esprime la dipendenza di un fatto da un insieme di altri fatti. Possono esprimere anche definizioni
- una regola ha una *testa (head)*, ed è il conseguente e un *corpo (body)*, l'antecedente. Si ha quindi un solo termine come conseguente (*regola di Horn*). Si ha quindi: *un pesce è un animale e ha le squame*:

```
pesce(x) :- animale(x), ha_le_squame(x).
```

- una relazione si può esprimere con più regole:

```
genitore(x, y) :- padre(x,y).
genitore(x, y) :- madre(x,y).
```

- si ha la ricorsione per definire una relazione. La definizione richiede almeno due proposizioni, *caso base* e *caso generale*:

```
antenato(x, y) :- genitore(x, y).
antenato(x, y) :- genitore(z, y), antenato(x, z).
```

- i commenti in linea si hanno con % e su più linee con /* e */

```
%commento
antenato(x, y) :- genitore(x, y).
antenato(x, y) :- genitore(z, y), antenato(x, z).
/*commento
su più righe*/
```

- il prompt ci chiede una FBF di Horn con "?-" (senza variabili) e prende l'input e come output si avrà *yes*. Interrogare il programma è richiedere una dimostrazione. Se si chiede una formula presente direttamente si avrà subito *yes* (se il goal è una clausola (un fatto) quindi si applica il principio di risoluzione e si ha la dimostrazione effettuata). In un'interrogazione si possono avere *variabili esistenziali*. Tutte le variabili istanziate sono mostrate in risposta

- l'operatore di istanziazione di variabili durante la prova di un predicato è il risultato di una procedura particolare, detta *unificazione*

- dati due termini la procedura di unificazione crea un insieme di sostituzioni delle variabili. Questo insieme permette di rendere uguali i due termini. Tradizionalmente si ha il *most general unifier (Mgu)*. Una sostituzione è indicata come una sequenza ordinata di coppie *variabile/valore*:

```
Mgu(42, 42) % ->{} non serve nessuna sostituzione
Mgu(42, X) % ->{X/42} ovvero X deve essere 42
Mgu(X,42) % ->{X/42} ovvero X deve essere 42
Mgu(foo(bar, 42), foo(bar, X)) % ->{X/42} ovvero X deve essere 42
Mgu(foo(X, 42), foo(bar, X)) % ->{Y/bar, X/42} ovvero X deve essere 42
```

```
Mgu(foo(bar(42), baz), foo(X, Y))  /* ->{X/bar (42), Y/baz}
 ovvero X deve essere  bar(42) e Y baz*/
Mgu(foo(X), foo(bar(Y)))
Mgu(foo(bar(42), baz), foo(X, Y))  /* ->{X/bar (y), Y_:G001}
 ovvero non si ha soluzione */
```

L'Mgu non è altro che il risultato finale della procedura di valutazione del Prolog. Il modo più semplice per vedere se l'unificazione funziona è usare "=", ovvero, ricollegandoci al codice sopra:

```
?- 42 = 42.
Yes

?- 42 = X.
X = 42 % per rendere vera l'affermazione serve x = 42
Yes

?- foo(bar, 42) = foo(bar, X).
X = 42
Yes

?- foo(Y, 42) = foo(bar, X).
Y = bar
X = 42
Yes

?- foo(bar(42), baz) = foo(X, Y).
X = bar(42)
Y = baz
Yes

?- foo(X) = foo(bar(Y)).
X = bar(Y)
Y = _G001
Yes

?- foo(42, bar(X), trillian) = foo(Y, bar(Y), X).
No
```

Esempio 11. *Si descrive un insieme di fatti riguardanti i corsi offerti: Tutte le informazioni sono concentrate in una relazione a 6 campi:*

```
corso(linguaggi, lunedì, '9:30', 'U4', 3, antoniotti).
corso(biologia_computazionale, lunedì, '14:30', 'U14', t023, antoniotti).
```

a partire da qui costruisco altri predicati:

```
aula(Corso, Edificio, Aula) :-
^^Icorso(Corso, _, _, Edificio, Aula, _).
docente(Corso, Docente) :-
^^Icorso(Corso, _, _, _, _, Docente).
```

oppure posso concentrare le informazioni in una relazione con 4 campi; le informazioni sono concentrate in termini funzionali che rappresentano le informazioni raggruppate logicamente;

```
corso(linguaggi, orario(lunedì, '9:30'), aula('U4', 3), antoniotti).
corso(biologia_computazionale,
^^Iorario(lunedì, '14:30'),
^^Iaula('U4', 3),
^^Iantoniotti).
```

posso quindi definire altri predicati:

```
aula(Corso, Edificio, Aula) :- corso(Corso, _, aula(Edificio, Aula), _).
docente(Corso, Docente) :- corso(Corso, _, _, Docente).

% oppure...

aula(Corso, Luogo) :- corso(Corso, _, Luogo, _).
```

oppure ancora i predicati che abbiamo definito a partire dalle relazioni con 6 o 4 campi possono essere ricodificate con predicati binari:

```
giorno(linguaggi, martedì).
orario(linguaggi, '9:30').
edificio(linguaggi, 'U4').
aula(linguaggi, 3).
docente(linguaggi, antoniotti).
```

La relazioni a 6 o 4 argomenti possono essere ricostruite in a partire da queste relazioni binarie. La costruzione di schemi RDF/XML (e, a volte, SQL) corrisponde a questa operazione di ri-rappresentazione

- Si definisce una lista in Prolog racchiudendo gli elementi (termini e/o variabili logiche) della lista tra parentesi quadre *[* e *]* e separandoli

da virgole. Gli elementi di una lista in Prolog possono essere termini qualsiasi o liste. La lista vuota si indica con *[]*. Una coda non vuota si può dividere in *testa* e *coda*:

- la testa è il primo elemento della lista
- la coda rappresenta tutto il resto ed è sempre una lista

```
[a, b, c] % a è la testa e [b, c] la coda
[a, b] % a è la testa e [b] la coda
[a] % a è la testa e [] la coda
[[a]] % [a] è la testa e [] la coda
[[a, b], c] % [a, b] è la testa e [c] la coda
[[a, b], [c], d] % [a, b] è la testa e [[c], d] la coda
```

Prolog possiede uno speciale operatore usato per distinguere tra l'inizio e la coda di una lista: l'operatore /:

```
?- [X | Ys] = [mia, vincent, jules, yolanda].
X = mia
Ys = [vincent, jules, yolanda]
Yes

?- [X, Y | Zs] = [the, answer, is, 42].
X = the
Y = answer
Zs = [is, 42]
Yes

?- [X, 42 | _] = [41, 42, 43, foo(bar)].
X = 41
Yes
```

La lista vuota *[]* in prolog è gestita come una lista speciale:

```
?- [X | Ys] = [].
No
```

- La base di conoscenza è nascosta ed è accessibile solo tramite opportuni comandi o tramite ambiente di programmazione. Bisogna poter caricare un insieme di fatti e regole. Per farlo si ha il comando *consult*, che appare come un predicato da valutare (un goal) e prende almeno un termine che denota un file come argomento, file che deve contenere fatti e regole

```
?- consult('guida-astrostoppista.pl').
Yes

?- consult('Projects/Lang/Prolog/Code/esempi-liste.pl').
Yes
```

- il predicato *consult* può essere usato anche per inserire fatti e regole direttamente alla console usando il termine speciale *user*:

```
?- reconsult('guida-astrostoppista.pl').
Yes
% A questo punto la base di dati Prolog contiene il
% nuovo contenuto del file.
```

il predicato *reconsult* deve invece essere usato quando si vuole ricaricare un file (ovvero un data o knowledge base) nell'ambiente Prolog. L'effetto è di prendere i predicati presenti nel file, rimuoverli completamente dal data base interno e di reinstallarli utilizzando le nuove definizioni:

```
?- reconsult(user). % Notare il sotto-prompt.
|- foo(42).
|- friends(zaphod, trillian).
|- ^D
Yes

% A questo punto la base di dati Prolog contiene i due
% fatti inseriti manualmente.
?- friends(zaphod, W).
W = trillian
Yes
```

- Prolog lavora anche su strutture ad albero e anche i programmi sono strutture dati manipolabili con predicati *extra-logici*. Si ha la ricorsione e non l'assegnamento. Un programma Prolog è un insieme di clausole di Horn che rappresentano:
  - fatti riguardanti gli oggetti in esame e le relazioni che intercorrono tra di loro
  - regole sugli oggetti e sulle relazioni (se ... allora ...)
  - interrogazioni (goals o queries; clausole senza testa), sulla base della conoscenza definita

- si ha l'implicazione:

  $$\neg B \vee A \text{ corrisponde a } B \rightarrow A$$

  data una clausola $A \vee ... \vee A_n \vee \neg B_1 \vee ... \vee \neg B_n$, usando De Morgan si ottiene:

  $$(A \vee ... \vee A_n) \vee \neg(B_1 \vee ... \vee B_n)$$
  $$\downarrow$$
  $$(B \vee ... \vee B_n) \rightarrow (A_1 \vee ... \vee A_n)$$

  si può fare anche l'inverso.

vediamo un programma che fa la somma tra due naturali:

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

con $S(n)$ interpretato come il successore ($0 = 0$, $s(0) = 1$, $s(s(0)) = 2, ...$).
Quando viene specificato un goal esso viene confrontato con tutte le clausole di programma, che usa il procedimento di negazione e di trasformazione in sintassi Prolog.
Vediamo i passi:

- interroghiamo il programma:

  $$\exists X \;\; sum(s(0),\, 0,\, X) \quad \{X \,/\, s(0)\}$$
  $$\exists W \;\; sum(s(s(0)),\, s(0),\, W) \quad \{W \,/\, s(s(s(0)))\}$$

  dove $\{X \,/\, s(0)\}$ e $\{W \,/\, s(s(s(0)))\}$ sono le sostituzioni che rappresentano il risultato

- usiamo il procedimento di negazione e trasformazione in sintassi:

  ```
  :- sum(s(0), 0, N).     %{N / s(0)}
  :- sum (s(s(0)), s(0), W).     %{W / s(s(s(0)))}
  ```

  con $\{N \,/\, s(0)\}$ e $\{W \,/\, s(s(s(0)))\}$ che sono le **sostituzioni**. Una sostituzione ci dice che valori (che possono essere altre variabili) si può sostituire le variabili di un termine. Una sostituzione si indica con $\sigma = \{X_1 \,/\, V_1/, ..., X_k \,/\, V_k\}$ e può essere considerata come una funzione $(\sigma : T \rightarrow T)$ applicabile ad un termine dell'insieme dei termini T.

  Esempio 12. *data la sostituzione* $\sigma = \{X \,/\, 42,\, Y \,/\, foo(s(0))\}$ *si ha:*

  $$\sigma(bar(X,\, Y)) = bar(42,\, foo(s(0)))$$

Una computazione corrisponde al tentativo di dimostrare, tramite la regola di risoluzione, che una formula segue logicamente da un programma (è un teorema). Si ha inoltre che si deve determinare una sostituzione per le variabili del goal (le *query*) per cui la query segue logicamente dal programma:

Esempio 13. *Sia dato il programma P e la query:*

```
:- p(t1, t2, ..., tm).
```

se ***X1,..., Xn*** *sono le variabili in* ***t1,..., tm****, il significato della query è:*

$$\exists X1,\ ...,\ Xn\,.\,p(t1, t2, ..., tm)$$

e si cerca una sostituzione:

$$s = \{X1\,/\,s1, ...,\ Xn\,/\,sn$$

con gli s i termini tali per cui:

$$P \vdash s[p(t1, t2, ..., tm)]$$

Dato un insieme di clausole di Horn è possibile derivare la clausola vuota solo se c'è almeno una clausola senza testa, ovvero se si ha una query G_0 da provare. Ovvero si deve dimostrare che da $P \cup G_0$ si può derivare la clausola vuota. Si dimostra per assurdo col principio di risoluzione.

Risoluzione ad Input Lineare (SLD)

Il sistema Prolog dimostra la veridicità di un goal eseguendo una sequenza di passi di risoluzione (l'ordine con cui vengono eseguiti questi passi rende i sistemi di prova basati sulla risoluzione più o meno efficienti). In Prolog la risoluzione avviene sempre fra l'ultimo goal derivato in ciascun passo e una *clausola di programma* e mai fra due clausole di programma o fra una clausola di programma ed un goal derivato in precedenza. Questa forma di risoluzione è detta*Risoluzione-SLD* (*Selection function for Linear and Definite sentences Resolution*), dove le sentenze lineari sono le clausole di Horn.

Esempio 14. *partiamo dal goal G_i:*

$$G_i \equiv\ ?-\ A_{i,1}, ..., A_{i,m}.$$

e dalla regola:

$$A_r\ \ :-\ B_{r,1}, ..., B_{r,k}.$$

se esiste un unificatore σ *tale che* $\sigma[A_r] = \sigma[A_{i,1}]$ *allora si ottiene il nuovo goal:*

$$G_{i+1} \equiv \ B'_{r,1}, ..., B'_{r,k}, A'_{i,1}, ..., A'_{i,m}.$$

che è un passo di risoluzione eseguito dal sistema Prolog (con $\sigma[A_{i,m}] = A'_{i,m}$ *4* $\sigma[B_{i,m}] = B'_{i,m}$*. La scelta di unificare il primo sottogoal di* G_i *è arbitraria (si sarebbe potuto scegliere, per esempio* $A_{i,m}$ *o* $A_{i,c}$ *con* $c \in [1, m]$ *)*

Esempio 15. *partiamo dal goal* G_i*:*

$$G_i \equiv \ ? - \ A_{i,1}, ..., A_{i,m}.$$

e dalla regola (ovvero dal ***fatto****):*

$$A_r.$$

se esiste un unificatore σ *tale che* $\sigma[A_r] = \sigma[A_{i,1}]$ *allora si ottiene il nuovo goal:*

$$G_{i+1} \equiv \ A'_{i,2}, ..., A'_{i,m}.$$

che ha dimensioni minori di G_i *avendo* $m - 1$ *sottogoal*

Nella risoluzione SLD, il passo di risoluzione avviene sempre fra l'ultimo goal e una clausola di programma. SI possono avere i seguenti risultati:

- **successo:** si genera la clausola vuota, ovvero se per n finito G_n è uguale alla clausola vuota $G_{\cdot}n \equiv \ : -$
- **insuccesso finito**: se per n finito G_n non è uguale alla clausola vuota $G_{\cdot}n \equiv \ : -$ e non è più possibile derivare un nuovo *risolvente* da G_n ed una clausola d programma
- **successo infinito:** se è sempre possibile derivare nuovi risolventi tutti diversi dalla clausola vuota

La **sostituzione di risposta** è la sequenza di unificatori usati; applicata alle variabili nei termini del goal iniziale dà la risposta finale.
Durante il processo di generazione di goal intermedi si costruiscono delle varianti dei letterali e delle clausole coinvolti mediante la *rinominazione di variabili*. Una variante per una clausola C é la clausola C' ottenuta da C rinominando le sue variabili, *renaming*:

Esempio 16. *esempio:*

```
p(X) :- q(X, g(Z)).
% è uguale alla clausola con variabili rinominate:
p(X1) :- q(X1, g(FooFrobboz)).
```

Possono esserci più clausole di programma utilizzabili per applicare la risoluzione con il goal corrente ed esistono diverse strategie di ricerca:

- **in profondità (Depth First)**: si sceglie una clausola e si mantiene fissa questa scelta, finché non si arriva alla clausola vuota o alla impossibilità di fare nuove risoluzioni; in questo ultimo caso si riconsiderano le scelte fatte precedentemente
- **in ampiezza (Breadth First)**: si considerano in parallelo tutte le possibili alternative

IL PROLOG ADOTTA UNA STRATEGIA DI RISOLUZIONE IN PROFONDITÀ CON *BACKTRACKING*.
Si ha così risparmio di memoria anche se non è una strategia completa per le clausole di Horn.

Alberi di Derivazione

Dato un programma logico P, un goal G_i e una regola di calcolo R si ha che un *albero SLD* per $P \cup G_i$ via R è definito sulla base del processo di prova visto precedentemente:

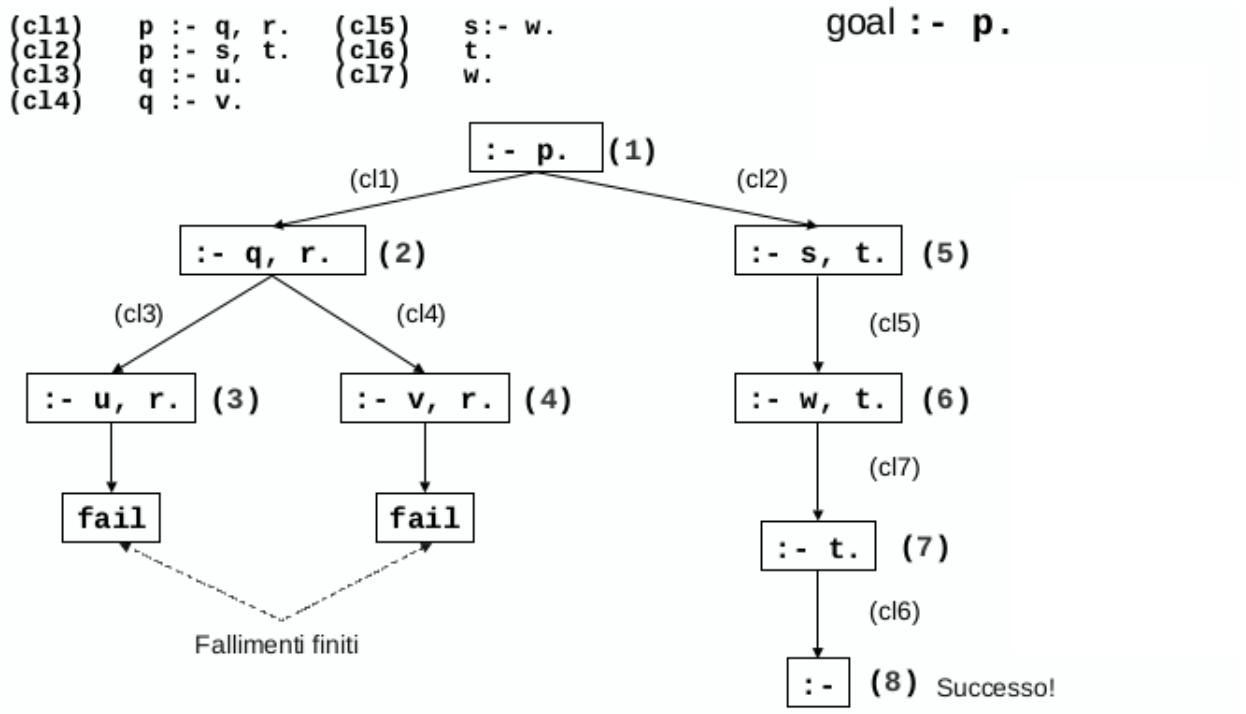
- ciascun **nodo** dell'albero è un goal (possibilmente vuoto)
- la **radice** dell'albero SLD è il goal G_0
- dato il nodo:
$$: - \ A_1, ..., A_{m-1}, A_m, A_{m+1}, ..., A_k$$
se A_m è il sottogoal selezionato dalla regola di calcolo R, allora questo nodo (genitore) ha un nodo figlio per ciascuna clausola del tipo:
$$C_i \equiv A_i \ : - \ B_{i,1}, ..., B_{i,q}$$
$$C_k \equiv A_k$$
di P tale che A_i e A_M (A_K e A_m) sono unificabili attraverso la sostituzione più generale σ.
Il nodo figlio è etichettato con la clausola goal
$$: - \ \sigma[A_1, ..., A_{m-1}, B_{i,i}, ..., B_{i,q}, A_{m+1}, ..., A_k]$$
$$: - \ \sigma[A_1, ..., A_{m-1}, A_m, A_{m+1}, ..., A_k]$$
e il ramo dal nodo padre al figlio è etichettato dalla sostituzione σ e dalla clausola selezionata C_i o C_k.
Il nuovo nodo $: -$ non ha figli.

La regola R è variabile. Si ha:

- **regola Left-most**: ovvero si ha la scelta del sottogoal più a sinistra
- **regola Right-most**: ovvero si ha la scelta del sottogoal più a destra
- si può avere la scelta di un sottogoal a caso
- se si ha un modo per decider il miglior sottogoal

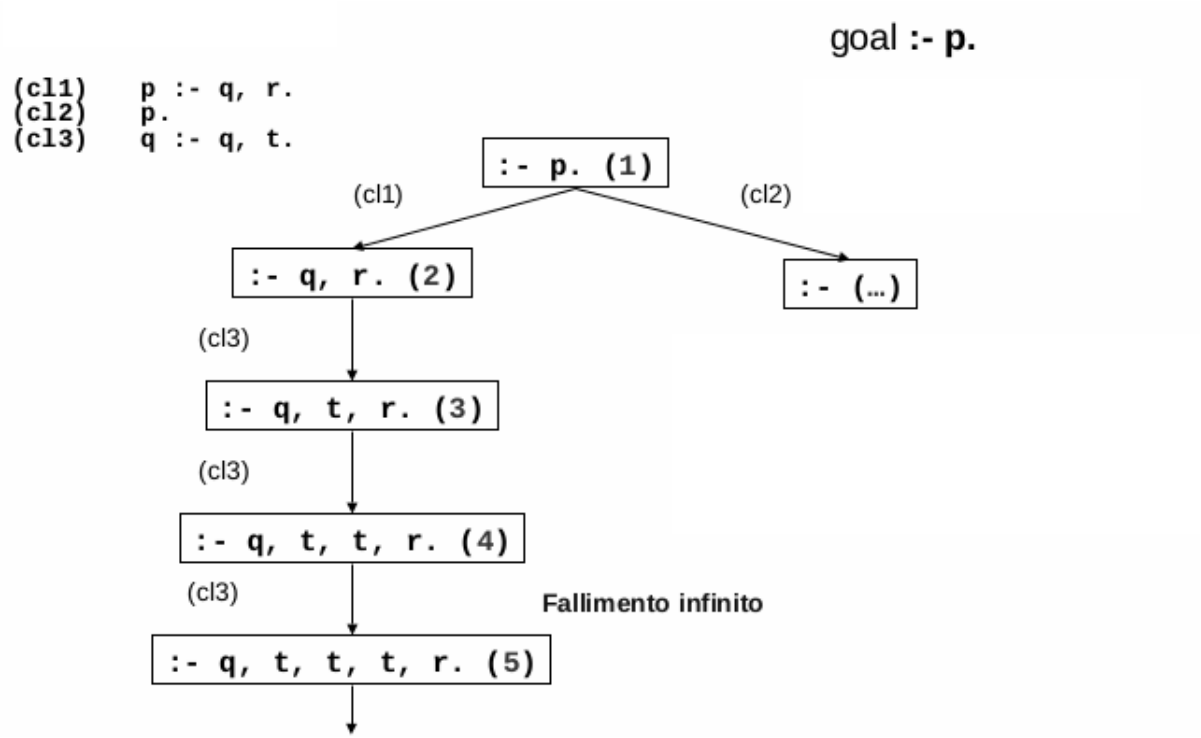
IL PROLOG ADOTTA UNA REGOLA LEFT-MOST
L'albero SLD (implicito!) generato dal sistema Prolog ordina i figli di un nodo secondo l'ordine dall'alto verso il basso delle regole e dei fatti del programma P.

Esempio 17. *Ecco un esempio:*



Ad ogni ramo di un albero SLD corrisponde una derivazione SLD e o gni ramo che termina con il nodo vuoto : − rappresenta una derivazione SLD di successo. La regola di calcolo influisce sulla struttura dell'albero per quanto riguarda sia l'ampiezza sia la profondità tuttavia non influisce su correttezza e completezza; quindi, qualunque sia R, il numero di cammini di successo (se in numero finito) è lo stesso in tutti gli alberi SLD costruibili per $P \cup \{G_0\}$ e R influenza solo il numero di cammini di fallimento (finiti ed infiniti).

Esempio 18. *Ecco un altro esempio, con fallimento infinito, dove la clausola vuota può essere generata ma il Prolog non è in grado di trovare questa soluzione dato che la sua strategia di percorrimento dell'albero (implicito) di soluzioni è depth-first con backtracking :*



Esempio 19.

3.0.1 Cut e Backtracking

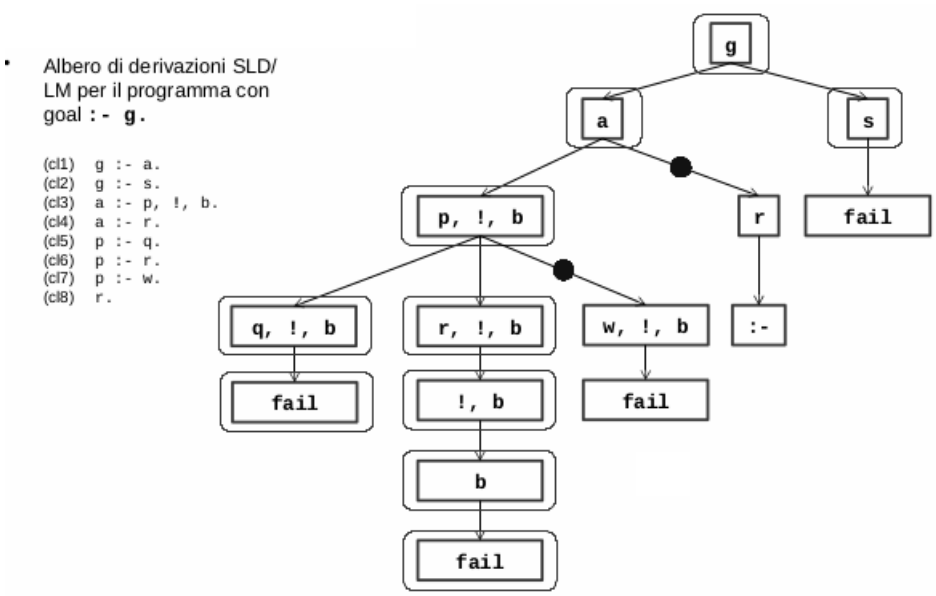
Permette di controllare il backtracking, ovvero si tagliano certe possibilità di backtracking. Si indica con ! ed effettua un'interpretazione procedurale.
Come abbiamo intuito, le clausole nel data base di un programma Prolog vengono considerate "da sinistra, verso destra" e "dall'alto al basso". Se un (sotto)goal fallisce, allora il dimostratore Prolog, sceglie un'alternativa, scandendo "dall'alto" verso "il basso" la lista delle clausole. Questo può essere controllato dal *cut*. Un esempio:

```
a :- b1, b2, ..., bk, !, ..., bn.
```

Questo è l'effetto del cut:

- se il goal corrente G unifica con a e $b_1, ..., b_k$ hanno successo, allora il dimostratore si impegna inderogabilmente alla scelta di C per dimostrare G
- ogni clausola alternativa (successiva, in basso) per a che unifica con G viene ignorata
- se un qualche b_j con $j > k$ fallisse, il backtracking si fermerebbe al cut e le altre scelte sono rimosse dall'albero di derivazione
- quando il backtracking raggiunge il cut, allora il cut fallisce e la ricerca procede dall'ultimo punto di scelta prima che G scegliesse C

vediamo un albero di derivazione in caso di cut:



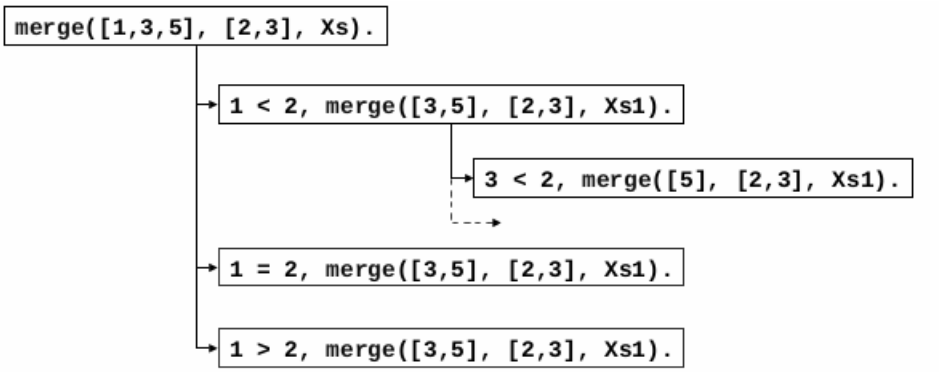
Si hanno due tipi di cut:

- **green cut:** utili per esprimere "determinismo" (e quindi per rendere più efficiente il programma)
- **red cut:** usati per soli scopi di efficienza, hanno per caratteristica principale quella di omettere alcune condizioni esplicite in un programma e, soprattutto, quella di modificare la semantica del programma equivalente senza cuts. Sono indesiderabili ma utili

consideriamo il seguente codice:

```
/* merge di due liste ordinate*/

merge([X | Xs], [Y | Ys], [X | Zs]) :-
^^IX < Y,
^^Imerge(Xs, [Y | Ys], Zs).
merge([X | Xs], [Y | Ys], [X, Y | Zs]) :-
^^IX = Y,
^^Imerge(Xs, Ys, Zs).
merge([X | Xs], [Y | Ys], [Y | Zs]) :-
^^IX > Y,
^^Imerge([X | Xs], Ys, Zs).
merge([], Ys, Ys).
merge(Xs, [], Xs).

/+ minimo tra due numeri */

minimum(X, Y, X) :- X =< Y.
minimum(X, Y, Y) :- Y < X.
```

vediamo cosa succede:



Solo la prima clausola ha successo, le altre due falliscono al momento del confronto numerico; ciononostante tutte e tre le clausole vengono considerate

consideriamo la seguente query:

```
?- merge([], [], Xs).
Xs = [];
Xs = [];
False.
```

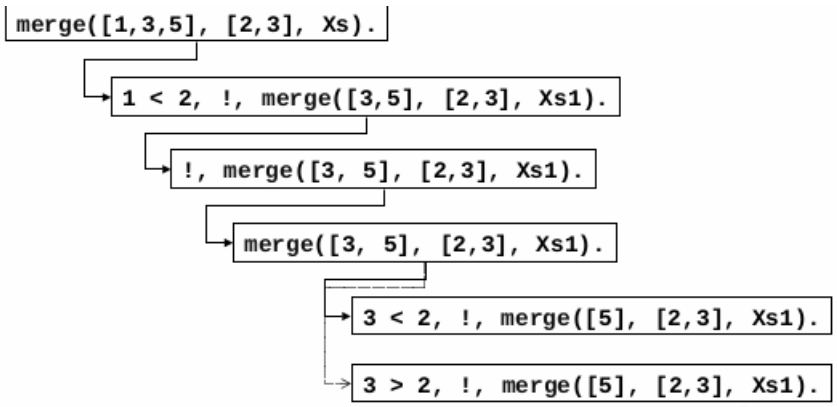
si ha una soluzione di troppo.
Un programma Prolog si dice **deterministico** quando una sola delle clausole serve (o si vorrebbe servisse) per provare un dato goal. Si usano quindi i green cuts:

```
merge([X | Xs], [Y | Ys], [X | Zs]) :-
^^IX < Y, !,
^^Imerge(Xs, [Y | Ys], Zs).
merge([X | Xs], [Y | Ys], [X, Y | Zs]) :-
^^IX = Y, !,
^^Imerge(Xs, Ys, Zs).
merge([X | Xs], [Y | Ys], [Y | Zs]) :-
^^IX > Y, !,
^^Imerge([X | Xs], Ys, Zs).
merge([], Ys, Ys) :- !.
merge(Xs, [], Xs) :- !.
```

ottenendo:

```
?- merge([], [], Xs).
Xs = [];
False.
```

ovvero:



Solo la prima clausola ha successo, ed il cut fa si che la seconda e la terza clausola **non** vengano considerate

il programma del minimo diventa:

```
minimum(X, Y, X) :- X =< Y, !.
minimum(X, Y, Y) :- Y < X, !.
```

qui il secondo cut è ridondante ma viene messo per simmetria.
Una volta che il programma ha fallito la prima clausola (ovvero il test X =< Y) al sistema Prolog non rimane che controllare la clausola seguente. Riscrivo in maniera non simmetrica:

```
minimum(X, Y, X) :- X =< Y, !.
minimum(X, Y, Y).
```

Qui si ha un red cut dato che taglia solo delle soluzioni, portando anche a risultati errati.

3.0.2 Predicati Meta-Logici

Vediamo il predicato:

```
celsius_fahrenheit(C, F) :- C is 5/9 * (F - 32).
```

Questo predicato non è invertibile. Si deve decidere qual è l'input e qual è l'output. Per questi problemi si hanno i *predicati meta-logici*. La ragione di questo effetto (della non invertibilità) sta nell'uso che abbiamo fatto di vari predicati aritmetici nel corpo dei predicati (>, <, =<, is, etc), per poter usare i predicati aritmetici che usano direttamente l'hardware abbiamo sacrificato la semantica dei nostri programmi.
I predicati meta-logici principali trattano le variabili come oggetti del linguaggio e ci permettono di riscrivere molti programmi che usano i predicati aritmetici di sistema come predicati dalla semantica "corretta" ed dal comportamento invertibile. Si hanno due predicati importanti:

- *var(X):* vero se X è una variabile logica
- *nonvar(X):* vero se X non è una variabile logica

ovvero:

```
?- var(foo).
No
?- var(X).
Yes
?- nonvar(42).
Yes
```

il nostro programma dei gradi diventa:

```
celsius_fahrenheit(C, F) :-
  var(C), nonvar(F), C is 5/9 * (F - 32).
celsius_fahrenheit(C, F) :-
  var(F), nonvar(C), F is (9/5 * C) + 32
```

con var decido che clausola usare. L'uso di questi predicati ci permette di scrivere programmi efficienti e semanticamente corretti.

3.0.3 Ispezione dei termini

Posso chiedere a prolog se ho a che fare con termini atomici o scomposti, con i seguenti predicati:

- *atomic(X):* vero se X è un numero od una costante
- *compound(X):* vero se non *atomic(X)*

Ho anche per manipolare un termine *Term*:

- *functor(Term, F, Arity)*, vero se *Term* è un termine, con *Arity* argomenti, il cui funtore (simbolo di funzione o di predicato) è F
- *arg(N, Term, Arg)*, vero se l'N-esimo argomento di *Term* è *Arg*
- *Term =.. L* questo predicato, =..,viene chiamato (per motivi storici) *univ*; è vero quando L è una lista il cui primo elemento è il funtore di *Term* ed i rimanenti elementi sono i suoi argomenti

ovvero:

```
?- functor(foo(24), foo, 1).
YES
?- functor(node(x, _, [], []), F, 4).
F = node
Yes
?- functor(Term, bar, 2).
Term = bar(_0,_1)
Yes
?- arg(3, node(x, _, [], []), X).
X = []
Yes
?- arg(1, father(X, lot), haran).
```

```
X = haran
Yes
?- father(haran, lot) =.. Ts.
Ts = [father, haran, lot]
Yes
?- father(X, lot) =.. [father, haran, lot].
X = haran
Yes
```

3.0.4 Programmazione di ordine superiore

Quando si formula una domanda per il sistema Prolog, ci si aspetta una risposta che è un'istanza (individuale) derivabile dalla knowledge base. Col backtracking ne abbiamo una alla folta, se le voglio tutte o voglio altri modi per acceder agli insiemi ho:

- *findall(Template, Goal, Set):*
  - Vero se Set contiene tutte le istanze di Template che soddisfano Goal
  - *Le istanze di Template vengono ottenute mediante backtracking*
- *bagof(Template, Goal, Bag):*
  - Vero se Bag contiene tutte le alternative di Template che soddisfano Goal
  - Le alternative vengono costruite facendo backtracking solo se vi sono delle variabili libere in Goal che non appaiono in Template
  - È possibile dichiarare quali variabili non vanno considerate libere al fine del backtracking grazie alla sintassi Var^G come Goal; In questo caso Var viene pensata come una variabile esistenziale
- *setof(Template, Goal, Set)*, che si comporta come bagof, ma Set non contiene soluzioni duplicate

ovvero:

```
/* findall */

?- findall(C, father(X, C), Kids).
C = _0
X = _1
Kids = [abraham, nachor, haran, isaac, lot, milcah, yiscah]
Yes.


/* bagof */

?- bagof(C, father(X, C), Kids).
C = _0
X = terach
KIDS = [abraham, haran, nachor];
C = _0
X = haran
KIDS = [lot, yiscah, milcah];
C = _0
X = abraham
KIDS = [isaac];
flase.

/* bagof con variabile esistenziale */

?- bagof(C, X^father(X, C), Kids).
C = _0
X = _1
Kids = [abraham, haran, lot, yiscah, nachor, isaac,...];
false.
```

3.0.5 Predicati di ordine superiore e meta variabili

Buona parte di questi predicati funziona grazie al meccanismo delle meta-variabili, ovvero variabili interpretabili come goals. Per esempio si ha il predicato *call*:

```
call(G) :- G.
```

Possiamo quindi definire il predicato *applica* che valuta una query composta da un funtore e da una lista di argomenti:

```
applica(P, Argomenti) :-
  P =.. PL, append(PL, Argomenti, GL), Goal =.. GL, call(Goal).
```

```
?- applica(father, [X, C]).
X = terach
C = abraham;
X = terach
C = nachor;
false

?- applica(father(terach), [C]).
C = abraham;
C = nachor;
false
```

3.0.6 Manipolazine della base di Dati

Un programma Prolog è costituito da una base di dati (o knowledge base) che contiene **fatti** e **regole**. Il Prolog però mette a disposizione anche altri predicati che servono a manipolare direttamente la base di dati. Ovviamente, questi predicati vanno usati con molta attenzione, dato che modificano dinamicamente lo stato del programma:

- *listing*
- *assert, asserta, assertz*
- *retract*
- *abolish*

se si ha una knowledge base vuota si avrà:

```
?- listing.
Yes
```

ovvero solo yes e il *listing* è vuoto. Aggiungo qualcosa alla base dati con:

```
?- assert(happy(maya)).
true
```

che risponderà sempre *true*, ora si avrà:

```
?- listing.
happy(maya).
true
```

e la base dati non sarà più vuota. Aggiungo altro alla base dati:

```
?- assert(happy(vincent)).
true

?- assert(happy(marcellus)).
true

?- assert(happy(butch)).
true

?- assert(happy(vincent)).
true
```

il *listing* ora darà:

```
?- listing.
happy(mia).
happy(vincent).
happy(marcellus).
happy(butch).
happy(vincent).
true
```

si nota come un dato si possa inserire più volte.
si possono anche asserire regole usando *assert* e mettendo la regola tra parentesi:

```
?- assert( (naive(X) :- happy(X)) ).
true

?- listing.
happy(mia).
happy(vincent).
happy(marcellus).
happy(butch).
happy(vincent).
naive(A) :-
happy(A).
true
```

possiamo anche rimuovere fatti e regole con *retract*, riprendendo dagli esempi sopra:

```
?- retract(happy(marcellus)).
true

?- listing.
happy(mia).
happy(vincent).
happy(butch).
```

```
happy(vincent).
naive(A) :-
happy(A).
true
```

inoltre si ha che *retract* rimuove solo la prima occorrenza:

```
?- listing.
happy(mia).
happy(vincent).
happy(butch).
happy(vincent).
naive(A) :-
happy(A).
true

?- retract(happy(vincent)).
true

?- listing.
happy(mia).
happy(butch).
happy(vincent).
naive(A) :-
happy(A).
true
```

Per rimuovere tutte le nostre asserzioni possiamo usare una variabile:

```
?- retract(happy(X)).
X = mia;
X = butch;
X = vincent;
false
```

```
?- listing.
naive(A) :-
happy(A)
true
```

Per avere più controllo su dove vengono aggiunti fatti e regole possiamo usare le due varianti di assert:

1. *assertz:* inserisce l'asserzione alla fine della knowledge base
2. *asserta:* inserisce l'asserzione all'inizio della knowledge base

ovvero, partendo da una base dari vuota:

```
?- assert(p(b)), assertz(p(c)), asserta(p(a)).
true

?- listing.
p(a).
p(b).
p(c).
true
```

manipolarei dati in prolog pè utile ma rischioso!
La manipolazione dati può essere usata per memorizzare i risultati intermedi di varie computazioni, in modo da non dover rifare delle queries dispendiose in futuro: semplicemente si ricerca direttamente il fatto appena asserito. Questa tecnica si chiama **memorization** o **caching**.

Esempio 20. *Creiamo una tavola di addizioni manipolando la knowledge base:*

```
addition_table(A) :-
  member(B, A),
  member(C, A),
  D is B + C,
  assert(sum(B, C, D)),
  fail.
```

dove member(X, Y) controlla che il predicato X appartenga alla lista Y.

```
?- addition_table([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).
false
```

La risposta è false; ma non è la risposta che ci interessa, bensì l'effetto (collaterale) che l'interrogazione ha sulla knowledge base:

```
?- listing(sum).
sum(0, 0, 0).
sum(0, 1, 1).
sum(0, 2, 2).
sum(0, 3, 3).
...
sum(9, 9, 18).
true
```

potremmo ora rimuovere tutti questi fatti non con il solito

```
?- retract(sum(X, Y, Z)).
```

che dovremmo ripetere ogni volta per ogni fatto ma con:

```
?- retract(sum(_, _, _)), fail.
false
```

Ancora una volta, lo scopo del fail è di forzare il backtracking. Il Prolog rimuove il primo fatto con funtore sum dalla base di dati e poi fallisce. Quindi fa bactrack e rimuove il fatto successivo e così via. Alla fine, dopo aver rimosso tutti i fatti con funtore sum, la query fallirà completamente ed il Prolog risponderà (correttamente) con un false. Ma anche in questo caso a noi interessa unicamente l'effetto collaterale sulla knowledge base.

3.0.7 Input e Output

I predicati primitivi principali per la gestione dell'I/O sono essenzialmente due, **read** e **write**, a cui si aggiungono i vari predicati per la gestione dei files e degli streams: **open, close, seek**, etc... *write* è l'equivalente del metodo *toString* JAva su un oggetto di classe "termine" mentre *read* invoca il parser prolog:

```
?- write(42).
42
true

?- foo(bar) = X, write(X).
foo(bar)
X = foo(bar)
?- read(What).
|: foo(42, Bar).
What = foo(42, _G270).

?- read(What), write('I just read: '), write(What).
|: read(What).
I just read: read(_G301)
What = read(_G301).

?- open'(some/file/here.'txt, write, Out),
write(Out, foo(bar)), put(Out, '0.), nl(Out),
close(Out).
true % But file "some/file/here."txt now contains the term 'foo
(bar').

?- open'(some/file/here.'txt, read, In),
read(In, What)
close(In).
What = foo(bar)
```

open e **close** servono per leggere e scrivere files; la versione più semplice di open ha con tre argomenti: un atomo che rappresenta il nome del file, una "modalità" con cui si apre il file ed un terzo argomento a cui si associa l'identificatore del file.

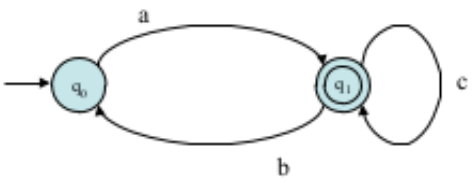
Esiste anche il predicato **put** che emette un carattere sullo stream ed il predicato nl che mette un 'newline' sullo stream.
Il Prolog usa la notazione *0'c* per rappresentare i caratteri come termini

3.0.8 Interpreti in Prolog

Il Prolog si presta benissimo alla costruzione di interpreti per la manipolazione di linguaggi specializzati (Domain Specific Languages (DSLs)). Vediamo qualche esempio:

- intepreti per Automi (a Stati Finiti, a Pila, Macchine di Turing)
- sistemi per la deduzione automatica
- sistemi per la manipolazione del Linguaggio Naturale (Natural Language Processing (NLP))

Vediamo un automa:



i nodi rappresentano degli stati. Si ha uno stato inziale (q_0) e uno finale, denotato con un doppio cerchio, (q_1). Gli archi sono le transizioni tra due stati e nel nostro caso si generano caratteri. Nell'esempio ho solo stringhe che iniziano con a. In q_1 posso fermarmi e generare solo la stringa a o produrre altri caratteri in numero variabile. Le stringhe qui prodotte saranno del tipo $ac^n(ba)^n$, con $n \geq 0$.

Per decidere se una certa sequenza di simboli è riconosciuta dall'automa possiamo costruire il seguente predicato:

```
/* definisco l'interprete */
accept([I | Is], S) :-
  delta(S, I, N),
  accept(Is, N).
accept([], Q) :- final(Q). /* è l'uscita, se ho una lista vuota
                              ho generato tutto */

/* richiesta di riconoscere un automa con stato iniziale S
    poi si usa l'accept con input che rappresenta la stringa
     considerando lo stato inziale S */
recognize(Input) :- initial(S), accept(Input, S).

initial(q0). % stato iniziale
final(q1). % stato finale

/* definisco le transizioni, con delta a tre argomenti */
delta(q0, a, q1).
delta(q1, b, q0).
delta(q1, c, q1).
```

```
?- recognize([a, b, a, c, c, b, a]).
Yes

?- recognize([a, b, a, c, b]).
No
```

vediamo cosa succede:
prendo la lista $[a, b, a, c, c, b, a]$, si hanno le seguenti sostituzioni: $I\backslash a$ e $S\backslash q0$. Avrò quindi:

```
delta(q0, a, N).
accept([b, ...], N).
```

ora si hanno le seguenti sostituzoni: $N\backslash q1$:

```
accept([b, ...], q1). % è il nuovo goal
```

unifico ancora con le sostituzioni $I2\backslash b$ e $Is\backslash[a, c, c, b, a]$:

```
delta(q1, b, N2).
accept([a, c, c, b, a], N2).
```

ad un certo punto arriveremo alla fine:

```
accept([], q1).
```

che unifica con la seconda clausola, sostituendo $q1\backslash Q$, arrivando a:

```
final(q1).
```

3.0.9 Meta-interpreti

Possiamo scrivere degli interpreti più complicati e/o specializzati se accettiamo di rappresentare i programmi con una sintassi leggermente diversa:

```
rule(append([], X, X)).
rule(append([X | Xs], Ys, [X | Zs]), [append(Xs, Ys, Zs)]).

solve(Goal) :- solve(Goal, []). /* lista goal e lista di appoggio
                                   dove mettere i goal restanti */

solve([], []). % clausola di uscita

solve([], [G | Goals]) :-
  solve(G, Goals).
solve([A | B], Goals) :-
  append(B, Goals, BGoals),
  solve(A, BGoals).
solve(A, Goals) :-
  rule(A),
  solve(Goals, []).
solve(A, Goals) :-
  rule(A, B),
  solve(B, Goals).
```

Il programma *solve* è un meta-interprete per i predicati *rule* che compongono il nostro sistema (o programma).

Si può avere incertezza nella conoscenza espressa. Si può avere unsa sorta di statistica:

```
solve_cf(true, 1) :- !.
solve_cf((A, B), C) :-
  !,
  solve_cf(A, CA),
  solve_cf(B, CB),
  minimum(CA, CB, C).
solve_cf(A, 1) :-
  builtin(A),
  !,
  call(A).
solve_cf(A, C) :-
  rule_cf(A, B, CR),
  solve_cf(B, CB),
  C is CR * CB.
```

Il programma *solve_cf* è un meta-interprete per stabilire se un goal G è vero e quanto siamo certi che sia vero.

Capitolo 4

Lisp

Usa il paradigma funzionale. I programmi computano combinando “valori” (rappresentati in memoria) trasformati da chiamate a funzioni. Si ha che i programmi sono funzioni matematiche, funzioni primitive o composte. Si ha il concetto matematico della **trasparenza referenziale** ovvero:

il significato del tutto si può determinare dal significato delle parti

questo concetto rende possibile la composizione di funzioni.
Ogni funzione denota un valore ottenuto tramite una mappa a partire dagli ”argomenti”. Nel paradigma funzionale vi sono oggetti di vario tipo e strutture di controllo, ma vengono raggruppati logicamente in modo diverso da come invece accade nel paradigma imperativo. In particolare è utile pensare in termini di:

- espressioni (funzioni primitive e non)
- modi di combinare tali espressioni per ottenerne di più complesse (composizione)
- modi e metodi di costruzione di “astrazioni” per poter far riferimento a gruppi di espressioni per “nome” e per trattarle come unità separate
- operatori speciali (condizionali ed altri ancora, che verranno introdotti in seguito)

ricordiamo che una funzione è una regola per associare gli elementi di un insieme (**dominio**) a quelli di un altro insieme (**codominio**). In un linguaggio funzionale l’argomento della funzione è mappato in particolari locazioni di memoria e non può essere modificato. Le composizioni di funzioni sono spesso organizzate in maniera ricorsiva. Le funzioni sono oggetti di prima

classe e possono essere puntatori a una funzione, una struttura dati e possono essere costruite durante l'esecuzione di un programma e ritornate come valore di un'altra funzione. Ilinguaggi funzionali consentono l'uso di funzioni di ordine superiore, cioè funzioni che prendono altre funzioni come argomenti e che possono restituirne come valore, in modo assolutamente generale. Nei linguaggi funzionali puri non esistono strutture di controllo iterative come while e for; questi sostituiti da ricorsione combinata con gli operatori speciali condizionali.
Lisp è una famiglia di linguaggi con due dialetti principali: **Common Lisp** e **Scheme**. Lo studio del LISP in una delle sue incarnazioni è importante dato che il linguaggio è uno dei più vecchi rappresentanti del paradigma di programmazione funzionale. Le versioni minimali di Lisp ammettono:

- funzioni primitive su **liste**
- un'operatore speciale **lambda** per creare funzioni
- un'operatore codizionale **cond**
- un piccolo insieme di predicati ed operatori speciali

Una prima cosa da notare è che in LISP ogni "espressione" denota un "valore".

```
prompt> 42
42

prompt> "Sapete che 'cose` ""42?"
Sapete che 'cose` ""42?
```

Si ha la seguente notazione:

$$/f\ x_1\ ...\ x_n)$$

Le parentesi iniziali e finale sono obbligatorie (vedremo più in là che ciò ha conseguenze importanti) e gli spazi (almeno uno) sono necessari per separare tra di loro la funzione e gli argomenti. le operazioni matematiche elementari sono funzioni:

```
prompt> (+ 40 2)
42

prompt> (- 84 42)
42

prompt> (* 2 3 7)
42
```

si possono avere più argomenti:

```
prompt> (+ 2 10 10 20)
42
```

e per le composte si ha che al posto di un valore posso avere la chiamata ad una funzione:

```
prompt> (+ 2 (* 2 10) 20)
42

prompt> (+ (* 3 (+ (* 2 4) (+ 3 2))) (+ (- 10 8) 1))
42
```

solitamente si allinea con gli argomenti (ovvero gli **operandi**) di una chiamata allineati verticalmente:

```
prompt> (+ (* 3
              (+ (* 2 4)
                 (+ 3 2)))
           (+ (- 10 8)
              1))
42
```

Notiamo come le funzioni aritmetiche elementari + e * in (Common) LISP rispettano i vincoli di "campo" algebrico:

```
prompt> (+)
0

prompt> (*)
1
```

Le funzioni aritmetiche elementari - e / in (Common) LISP richiedono almeno un argomento e rappresentano in questo caso il "reciproco", sempre in senso algebrico:

```
prompt> (- 42)
-42

prompt> (/ 42)
1/42
```

posso avere:

- numeri interi, numeri in virgola mobile (es. 3.5 o $6.02E+21$), numeri razionali (es. $-3/42$) o numeri complessi ($\#C(01)$)
- booleani T e NIL
- stringhe (es "*sono una stringa*")
- operatiori sui booleani *null*, *and*, *or*, *not*
- funzioni sui numeri *+ - / * mod sin cos sqrt tan atan plusp > <= zerop*

la valutazione delle funzioni va da sinistra a destra e produce i valori v_1 ... v_n a cui successivamente verrà applicata la funzione in maniera *inerentemente ricorsiva*. Questa valutazione non è seguita da alcuni operatori come *if, cond, defun, defparameter, quote, etc....*
Le variabili si definiscono con *defparameter*:

```
prompt> (defparameter quarantadue 42)
quarantadue
```

ora il simbolo quarantadue ha ora associato il valore 42.
Le funzioni si possono definire usando l'operatore speciale defun (nome (argomenti) (corpo funzione)):

```
prompt> (defun quadrato (x) (* x x))
quadrato
```

L'operatore defun associa il corpo della funzione al nome nell'ambiente globale del sistema Common Lisp e restituisce come valore il nome della funzione. Una volta definita, una funzione viene eseguita (o chiamata) usando la regola descritta precedentemente:

```
prompt> (quadrato quarantadue)
1764

prompt> (- (quadrato (+ 20 22)) 10)
1754

prompt> (defun somma-di-quadrati (x y)
          (+ (quadrato x) (quadrato y)))
somma-di-quadrati

prompt> (- (somma-di-quadrati 6 3) 3)
42
```

i nomi delle funzioni possono contenere il carattere "-" in quanto non si cra ambiguità con la funzione "-(...)", questo succede anche con *(,), #, ‘, ’, "* e con la virgola.
Per valutare le funzioni si ha la costruzione di un **activation frame**, i parametri formali di una funzione vengono associati ai valori (si passa tutto per valore senza effetti collaterali). Il corpo della funzione viene valutato (ricorsivamente) tenendo conto di questi legami in maniera statica, bisogna

tener presente cosa accade con variabili che risultano "libere" in una sotto-espressione. Ad ogni sotto-espressione del corpo si sostituisce il valore che essa denota (computa). l valore (valori) restituito dalla funzione è il valore del corpo della funzione (che non è altro che una sotto-espressione), Quando il valore finale viene ritornato i legami temporanei ai parametri formali spariscono (lo stack di sistema subisce una "pop" l'activation frame viene rimosso). Per esempio:

```
prompt> (defun doppio (n) (* 2 n))
doppio

prompt> (doppio 3)
6
```

- estende l'ambiente globale con il legame tra doppio e la sua definizione
- estende l'ambiente globale con quello locale che contiene i legami tra parametri formali e valori dei parametri attuali, un activation frame viene inserito in cima allo stack di valutazione
- valuta il corpo della funzione
- ripristina l'ambiente di partenza, l'activation frame viene rimosso dalla cima dello stack di valutazione

Per la valutazione di una funzione F, l'ambiente deve eseguire i seguenti sei passi:

1. mettere i parametri in un posto dove la procedura possa recuperarli
2. trasferire il controllo alla procedura
3. allocare le risorse (di memorizzazione dei dati) necessarie alla procedura
4. effettuare la computazione della procedura
5. mettere i risultati in un posto accessibile al chiamante
6. restituire il controllo al chiamante

Queste operazioni agiscono sui registri a disposizione e sullo "stack" utilizzato dal runtime (esecutore) del linguaggio.
Lo spazio richiesto per salvare (sullo stack) tutte le informazioni necessarie all'esecuzione di una funzione F ed al ripristino dello stato precedente alla chiamata è quindi costituito da:

- Spazio per i registri da salvare prima della chiamata di una sotto funzione
- Spazio per l'indirizzo di ritorno (nel codice del corpo della funzione)
- Spazio per le variabili, definizioni locali, e valori di ritorno
- Spazio per i valori degli argomenti
- Spazio per il riferimento statico (static link)
- Spazio per il riferimento dinamico (dynamic link)
- Altro spazio dipendente dal particolare linguaggio e/o politiche di allocazione del compilatore

La gestione dello static link in Common Lisp è in realtà più complicato, dato che il linguaggio ammette la creazione a runtime di funzioni che si devono ricordare il valore delle loro variabili libere al momento della loro creazione. Queste funzioni sono implementate con particolari strutture dati chiamate **chiusure (closures)**, che chiudiono i valori delle loro variabili libere. Si possono quindi creare funzioni anonime, le **lambda**:

$$(lambda\ (x_1\ ...\ x_n)\ <e>)$$

con $<e>$ espressione detta **lambda-expression**:

```
(lambda (x) (+ 2 x))

((lambda (x) (+ 2 x)) 40)
42
```

pensiamo ora alla creazione come quella per il valore assoluto che presenta più casistiche. Useremo il costrutto **cond**:

$$(cond\ (c_1\ e_1),\ ...\ (c_n\ e_n))$$

ovvero:

```
(defun valore-assoluto (x)
   (cond ((> x 0) x)
         ((= x 0) 0)
         ((< x 0) (- x))))
```

l'operatore cond verifica ogni coppia di espressioni e se il risultato è T ritorna il valore della seconda espressione altrimenti passa lal coppia successiva. Se non ci sono più coppie valutabili restituisce NIL

```
prompt> (valore-assoluto 3)
3

prompt> (valore-assoluto -42)
42
```

potevo anche definirlo come:

```
(defun valore-assoluto (x)
    (cond ((> x 0) x)
          (T (- x))))
```

e anche con:

```
(defun valore-assoluto (x)
   (if (> x 0) x (- x)))
```

si hanno ovviamente anche gli operatori booleani *and, or e not*:

```
prompt> (and (> 42 0) (< -42 0))
T

prompt> (not (> 42 0))
NIL

prompt> (and)
T

prompt> (or)
NIL
```

ecco una funzione per approssimare la radice quadrata di un numero:

```
(defun radice-quadrata (x)
    (ciclo-radice-quadrata x 1.0))

(defun va-bene? (x c)
    (< (valore-assoluto (- x (quadrato c)))
       0.001))

(defun media (x y) (/ (+ x y) 2.0))

(defun migliora (c x) (media c (/ x c)))

(defun ciclo-radice-quadrata (x c)
    (if (va-bene? x c)
        c
        (ciclo-radice-quadrata x (migliora c x))))
```

che esegue la seguente approssimazione:

$$1 \to \frac{2}{1} = 2 \to ((2+1)/2) = 1,5$$

$$1,5 \to \frac{2}{1,5} = 1,3333 \to ((1,3333+1,5)/2) = 1,4167$$

$$\dots$$

con un errore massimo di 0,001. La prima congettura è per il numero 1.
Vediamo una funzione ricoriva come il fattoriale:

```
(defun fattoriale (n)
    (if (= n 0)
        1
        (* n (fattoriale (- n 1)))))
```

Questa funzione calcola il fattoriale di un numero utilizzando una sequenza di valori intermedi che devono essere salvati da qualche parte (ovvero sullo "stack" di attivazione di ogni chiamata ricorsiva).
Vediamo anche Fibonacci:

```
(defun fib (n)
     (cond ((= n 0) 1)
           ((= n 1) 1)
           (T (+ (fib (- n 2)) (fib (- n 1))))
            ))
```

possiamo anche scrivere il fattoriale in una forma simile:

```
(defun fatt-ciclo (n acc)
      (if (= n 0)
          acc
          (fatt-ciclo (- n 1) (* n acc))))
(defun fattoriale (n)
    (fatt-ciclo n 1))
```

ovvero utilizzando uno degli argomenti come accumulatore. Con *fatt-ciclo* si ha un "ciclo in incognito" che è d'aiuto al compilatore che può effettuare una funzione di jump. Una funzione così è detta **tail-ricorsive**.
Un insieme di funzioni mutualmente ricorsive può rappresentare una macchina di Turing quindi, i linguaggi funzionali puri (senza assegnamenti e "salti") sono Turing-completi.
Creaiamo ora una libreria per fare conti coi razionali. Innanzitutto assumiamo di aver a disposizione una funzione che costruisce una rappresentazione di un numero razionale:

```
(crea-razionale n d) Þ <il razionale n/d>
```

Assumiamo anche di avere due funzioni numer and denom che estraggono rispettivamente il numeratore n ed il denominatore d dalla rappresentazione di un numero razionale <razionale n/d>:

```
(defun somma-raz (r1 r2)
    (crea-razionale (+ (* (numer r1) (denom r2))
                       (* (numer r2) (denom r1)))
                    (* (denom r1) (denom r2))))
(defun molt-raz (r1 r2)
    (crea-razionale (* (numer r1) (numer r2))
                    (* (denom r1) (denom r2)))
(defun =-raz (r1 r2)
    (= (* (numer r1) (denom r2))
       (* (numer r2) (denom r1))))
```

...

Una delle strutture più importanti del Lisp è la **Cons-Cell**, iovvero una coppia di puntatori a due elementi, *car* e *cdr*:

```
cons : <oggetto Lisp> x´<oggetto Lisp> -> <cons-cell>
```

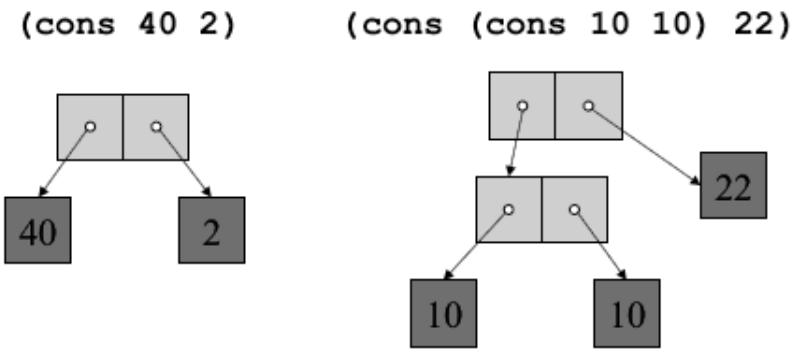
```
prompt> (defparameter c (cons 40 2))
c

prompt> (car c)
40

prompt> (cdr c)
2
```

con questo costrutto si semplifica la libreria per i razionali:

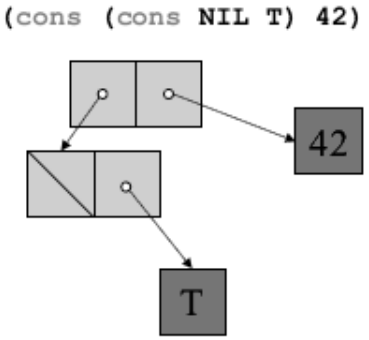
```
(defun crea-razionale (n d)
   (cons n d))

(defun numer (r) (car r))

(defun denom (r) (cdr r))
```

```
prompt> (denom (crea-razionale 42 7))
7
```

La funzione cons genera in memoria dei grafi di puntatori arbitrariamente complessi. Questi grafi vengono rappresentati in una tradizionale notazione detta *box-and-pointer*:



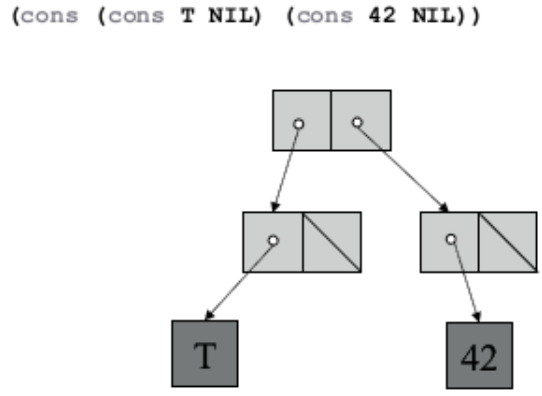
Posso inserire anche T e NIL (rappresentato da una sbarra:



bediamo qualche esempio:

```
prompt> (cons NIL T)
(NIL . T) ; Notazione "dotted-"pair
          ; "(coppia-"puntata). Gli spazi
          ; sono significativi.

prompt> (cons 4 2)
(4 . 2)

prompt> (cons
```

La notazione "dotted pair" è l'unica "irregolarità sintattica infissa" in Lisp.
Quando si parla di liste in Lisp ci si riferisce a particolari configurazioni di cons-cells dove l'ultimo puntatore è NIL; la costante NIL è equivalente alla lista vuota ():



Una lista Lisp corrispondente a questa configurazione di cons-cells viene rappresentata tipograficamente come *((T) 42)* che è uquivalente a *((T . NIL) . (42 . NIL))*. Quindi una cons-cell con NIL come secondo elemento (ovvero come cdr) viene stampata senza il punto ed il NIL:

```
prompt> (cons 42 nil)
(42)

prompt> (cons "foo "bar nil)"
(foo "bar)
```

inoltre una cons-cell con una cons-cell come secondo elemento è stampata in modo "abbreviato":

```
prompt> (cons 42 (cons 123 666))
(42 123 . 666)  ; Equivalente a (42 . (123 . 666))

prompt> (cons 42 (cons "foo "bar nil))
(42 "foo "bar)
;;; To`, una "lista.
```

cons si può usare anche per rappresentare una lista di oggetti:

```
(defparameter L (cons 1 (cons 2 (cons 3 (cons 4 NIL)))))
```

```
prompt> (car (cdr (cdr L)))
3
```

```
(defparameter L (list 1 2 3 4))
```

```
prompt> (car (cdr (cdr L)))
3
```

inoltre si ha la funzione list con un numero variabile di elementi:

```
prompt> (list -1 0 1 2 3)
(-1 0 1 2 3)
```

vediamo qualche manipolazione di una lista.
estraiamo l'n-esimo elemento:

```
(defun list-ref (n list)
    (if (<= n 0)
        (car list)
        (list-ref (- n 1) (cdr list)))
```

calocliamo la lunghezza di una lista:

```
(defun lunghezza (l)
    (if (null l)
        0
        (+ 1 (lunghezza (cdr l)))))
```

Si ha la funzione **nth**. Inoltre funzione cdr ritorna di fatto il "resto" di una lista e lo standard Common Lisp fornisce anche l'ovvio sinonimo: rest e la funzione car ritorna il "primo" elemento di una lista; lo standard Common Lisp fornisce anche il sinonimo first.
per fare l'append:

```
(defun appendi (l1 l2)
    (if (null l1)
        l2
        (cons (car l1) (appendi (cdr l1) l2))))
```

La funzione è ricorsiva e presenta una tipica forma di ricorsione "strutturale" sul resto di una lista (detta anche "cdr recursion" o "rest-recursion").
In LISP si possono manipolare dati simbolici, costruendo liste come:

```
(a b c d e f g h I)
((gino 10) (ugo 10) (maria 2) (ettore 20))
```

o anche:

```
(+ quaranta 2)
(defun media (x y) (/ (+ x y) 2.0))
```

Si ha l'operatore *quote* che dice al LISP di non procedere alla valutazione di una (sotto)espressione:

```
(quote <e>)
```

con <e> che viene ritornato letteralmente

```
prompt> (quote 42)
42

prompt> (quote A)
A

prompt> (quote (1 B 3))
(1 B 3)

prompt> (quote (1 2 (3 4) (5 6 ""foo)))
(1 2 (3 4) (5 6 ""foo))
```

il quote ha anche una forma abbreviata: '<e>:

```
prompt> '42
42

prompt> 'A
A

prompt> '(1 B 3)
(1 B 3)

prompt> '(1 2 (3 4) (5 6 ""foo))
(1 2 (3 4) (5 6 ""foo))
```

il quote non agisce su numeri e stringhe che sono autovalutanti, ovvero essi sono espressioni il cui valore ha la stessa rappresentazione tipografica dell'espressione tipografica che li denota. I simboli (o identificatori) denotano invece I valori che sono loro associati (ad esempio tramite defparameter). Inoltre le liste, se non quotate, rappresentano invece delle espressioni da valutare, ergo la necessità di avere l'operatore di quote:

```
prompt> 42
42

prompt> '42
42

prompt> '"io sono una "stringa"
io sono una "stringa

prompt> "io sono una "stringa"
io sono una "stringa

prompt> (list 'A 'B)
(A B)

prompt> '(A B)
(A B)

prompt> '(defun media (x y) (/ (+ x y) 2.0))
(DEFUN MEDIA (X Y) (/ (+ X Y) 2.0))

prompt> (defun media (x y) (/ (+ x y) 2.0))
MEDIA

prompt> (defparameter m'
(defun media (x y) (/ (+ x y) 2.0))
M

prompt> (third m) ; (car (cdr (cdr m)))
(X Y)
```

e quindi si ha che **in LISP i programmi e i dati sono esattamente la stessa cosa**
In LISP si ha una ripartizione degli oggetti in due categorie:

- **atomi:** che sono simboli e numeri (ma non solo; ad esempio le stringhe):

```
prompt> (atom 3)
T
prompt> (atom 'un-simbolo)
T
prompt> (atom "una "stringa)
T
prompt> (atom (cons -42 42))
NIL
prompt> (atom (list 1 2 3))
NIL
```

- **cons-cell:** ovvero le liste

cons-cell più numeri, simboli, stringhe etc... costituiscono le **Symbolic Expression (sexp's)**
Dato che programmi e sexp's in Lisp sono equivalenti, possiamo dare le seguenti regole di valutazione (ed implementarle nella funzione *eval*). data un sexp's:

- se è un atomo:
  - se è un numero ritorna il valore
  - se è una stringa ritornala così com'è
  - se è un simbolo ne estrae il valore dall'ambiente corrente e ne restituisce il valore, se non ha un valore associato da errore
- se è una cons-cell si ha:
  - Se O è un operatore speciale, allora la lista $(0\, A_1 \ldots A_n)$ viene valutata in modo speciale
  - Se O è un simbolo che denota una funzione nell'ambiente corrente, allora questa funzione viene applicata *apply* alla lista $(V A_1 \ldots V A_n)$ che raccoglie i valori delle valutazioni delle espressioni $A_1 \ldots A_n$
  - Se O è una Lambda Expression la si applica alla lista che $(V A_1 \ldots V A_n)$ che raccoglie i valori delle valutazioni delle espressioni $A_1 \ldots A_n$
  - altrimenti segnala errore

definisco il fattoriale:

```
(defun fatt (n)
    (if (zerop n) 1 (* n (fatt (- n 1)))))
```

```
prompt> (fatt 3)
n diventa associato a 3 “(bound ”to 3):
(eval ‘(fatt 3))
-> (eval ‘(if (zerop 3) 1 (* 3 (fatt (- 3 1)))))
      ... -> (eval ‘(* 3 (fatt 2)))
          ... -> (eval ‘(*3 (* 2 (fatt 1))))
          ... -> (* 3 (* 2 (* 1 (fatt 0))))
      ... -> (* 3 (* 2 (* 1 1)))
... -> ... 6
```

eval applica la funzione al suo argomento, definendo i legami di N “in cascata” e costruendo una espressione che alla fine verrà valutata, a partire dalla sotto-espressione più annidata.
Questa struttura delle liste si presta bene alla ricorsione.
Vediamo la lunghezza di una lista:

```
(defun lung (l)
   (if (null l)
       0
       (+ 1 (lung (rest l)))))
```

last restituisce l’ultimo elemento di una lista:

```
(defun last-l (l)
    (cond ((null l) nil)
          ((atom l) ; Notare questo caso.
           (error "L'argomento non e` una lista"))
          ((null (rest l)) (first l))
          (t (last-l (rest l)))))
```

si ha la ricorsione semplice, **ricorsione cdr** che è definita sul resto di una lista.
Vediamo come contare gli atomi di una sexp.
al caso base si ha che NIL, ”()” conta 0, un atomo conta 1. l’ipotesi ricorsiva effettua il conta sul rest della lista e il caso passo analizza il first della lista e chiama la ricorsione.
Esiste anche la ricorsione **car-cdr** che effettua la ricorsione sul car:

```
(defun count-atoms(x) ; 'x' e` una sexp.
  (cond ((null x) 0)
        ((atom x) 1)
        (T (+ (count-atoms (first x))
              (count-atoms (rest x))))))
```

Si definisce **profondità massima di annidamento** di una sexp come il numero di parentesi sospese (max è definito in Common Lisp:

```
(defun prof (x)
   (cond ((null x) 1)
         ((atom x) 0)
         (t (max (+ 1 (prof (first x)))
                 (prof (rest x))))))
```

vediamo la funzione flatten per appiattire una lista:

```
(defun flatten (x) ; x e` s-espressione
    (cond ((null x) x) ; NB: (atom nil) = T
          ((atom x) (list x)) ; serve per usare
                    ; append
          (T (append (flatten (first x))
                     (flatten (rest x))))))
```

```
prompt> (flatten '((((a (b (c d)) e) f))
(a b c d e f)

prompt> (flatten 'a)
(a)
```

per ottenere un'immagine speculare di una sexp:

```
(defun mirror (x) ; x e` una sexp
    (if (atom x)
        x
    (append (mirror (rest x))
            (list (mirror (first x))))))
```

```
prompt> (flatten '((((a (b (c d)) e) f))
(f (e ((d c) b) a)
```

la funzione per invertire:

```
(defun inverti (x)
    (cond ((null x) x)
          ((atom x) x) ; anche Sexp...
          (T (cons-end (first x)
                       (inverti (rest x))))))
```

funzione circulate:

```
(defun circulate (lst direction)
    (cond ((atom lst) lst)
          ((null lst) nil)
          ((eq direction 'left)
           (append (cdr lst) (list (car lst))))
          ((eq direction 'right)
           (cons (last-l lst) (but-last lst)))
          (T lst) ))
```

```
prompt> (circulate '(1 2 3 4) 'left)
(2 3 4 1)

prompt> (circulate '(1 2 3 4) 'right)
(4 1 2 3)
```

Un problema preliminare che si pone in Lisp è quello di controllare se due oggetti sono uguali; il Common Lisp mette a disposizione una pletora di predicati di uguaglianza con semantica diversa. Si hanno due soluzioni principali:

- **eql:** viene usato per controllare l'uguaglianza di simboli, numeri interi e puntatori:

```
prompt> (eql 42 42)
T

prompt> (eql 42 3)
NIL
```

```
prompt> (eql 'quarantadue 'quarantadue)
T

prompt> (eql 'quarantadue quarantadue)
NIL

prompt> (eql 'quarantadue 'fattoriale)
NIL

prompt> (eql '(42) '(42))
NIL
```

- **equal:** si comporta come *eql*, ma è in grado di controllare se due liste sono uguali; in pratica non fa altro che applicare *eql* ricorsivamente a tutti gli atomi di una lista: se un'applicazione di *eql* ritorna il valore NIL allora *equal* fa lo stesso, altrimenti viene ritornato il valore T:

```
prompt> (equal 42 42)
T

prompt> (equal 42 3)
NIL

prompt> (equal 'quarantadue 'quarantadue)
T

prompt> (equal '(1 2 3 (a s d) 4)'
                 (1 2 3 (a s d) 4))
T

prompt> (equal '(1 due tre (a s d) 4)'
                 (1 2 3 (a s d)4))
NIL
```

Supponiamo di voler moltiplicare tutti gli elementi di una lista definita per un certo valore e di ritornare una nuova lista con I nuovi elementi; questa funzione è semplicemente:

```
(defparameter pari (list 2 4 6 8 10))

(defun scala-lista (l fattore)
       (if (null l)
          nil
          (cons (* fattore (car l))
                (scala-lista (cdr l) fattore))))
```

Si noti che la funzione scala-lista ripete la struttura di appendi e si può astrarre se astraiamo il concetto di valore funzionale. L'astrazione "applica la funzione f a tutti gli elementi della lista L e ritorna una lista dei valori" è nota come "map"; in Common Lisp la funzione *mapcar* svolge questo compito. Questa funzione è già definita ma sarebbe:

```
(defun mapcar* (funzione lista)
        (if (null lista)
            nil
            (cons (funcall funzione (car lista))
                  (mapcar* funzione (cdr lista)))))
```

Dove il nome *mapcar** viene usato per evitare errori nell'ambiente Common Lisp. La funzione *funcall* serve invece chiamare una funzione con un certo argomento.
Definisco queste funzioni:

```
(defun scala-4 (x) (* x 4))
(defun scala-10 (x) (* x 10))
(defun scala-pi (x) (* x pi))
```

si ha:

```
prompt> (defun scala-lista-10 (lista)
          (mapcar 'scala-10 lista))
scala-lista-10

prompt> (scala-lista-10 '(1 2 3 4 5))
(10 20 30 40 50)
```

possiamo quindi definire:

```
(defun scala-lista (lista funzione-scalante)
   (mapcar funzione-scalante lista)
```

Il parametro funzione-scalante è associato alla funzione che ci interessa; ad esempio per scalare di un fattore 10 ora possiamo scrivere:

```
prompt> (scala-lista (list 1 2 3) 'scala-10)
(10 20 30)
```

torniamo quindi alle Lambda con cui possiamo creare tutte le funzioni che vogliamo senza assegnare loro un nome:

```
prompt> (lambda (x) (+ x 42))
#<funzione>

prompt> ((lambda (x) (+ x 42)) 42)
84

prompt> (scala-lista '(1 2 3)
                     (lambda (x) (* x 3)))
(3 6 9)
```

Possiamo anche creare funzioni che costruiscono delle funzioni e le ritornano come valori:

```
prompt> (defun adder-x (x)
        (lambda (y) (+ x y)))
adder-x

prompt> (defparameter adder-42 (adder-x 42))
adder-42

prompt> adder-42
#<function>

prompt> (funcall adder-42 42)
84
```

riscriviamo quindi lo scala-lista:

```
(defun scala-lista (lista fattore)
   (mapcar (lambda (e) (* e fattore)) lista))
```

```
prompt> (scala-lista (list 1 0 1 0 1) 42)
(42 0 42 0 42)
```

considero la funzione:

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1-y)(1+xy)$$

con la lambda diventa:

```
(defun f (x y)
   ((lambda (a b)
      (+ (* x (quadrato a))
         (* y b)
         (* a b)))
      (+ 1 (* x y))
          (- 1 y)))
```

Questo tipo di chiamate a funzioni anonime è così utile da essere stato ricodificato con un nuovo operatore speciale: *let*:

```
(defun f (x y)
   (let ((a (+ 1 (* x y)))
         (b (-1 y))
         )
    (+ (* x (quadrato a))
       (* y b)
       (* a b))))
```

Ovvero, l'operatore let ci permette di introdurre dei nuovi nomi (variabili) locali da poter riutilizzare all'interno di una procedura; la sua sintassi è la seguente:

```
(let ((n 1 e 1 ) (n 2 e 2 ) ... (n k e k )) espressione)
```

```
prompt> (let ((a 40) (b (+ 1 1))) (+ a b))
42
```

Le funzioni che prendono una (o più) funzioni come argomenti sono dette funzioni di ordine superiore e la loro esistenza è fondamentale nel paradigma funzionale.
La funzione *compose* corrisponde alla nozione matematica di composizione di funzioni.
La semantica della funzione è la seguente: date due funzioni (di un solo argomento) f e g come argomenti, ritorna una nuova funzione che corrisponde alla composizione f(g(x)):

```
(defun compose (f g)
   (lambda (x)
    (funcall f (funcall g x))))
```

```
prompt> (funcall (compose 'first 'rest)'
                 (1 2 3 4 5))
2
```

La funzione *filter* rimuove gli elementi della lista che non soddisfano il predicato:

```
(defun filter (predicato lista)
   (cond ((null lista) nil)
         ((funcall predicato (car lista))
            (cons (car lista)
                  (filter predicato (cdr lista)))
          (T (filter predicato (cdr lista))))))
```

```
prompt> (filter 'oddp '(1 2 3 4 5))
(1 3 5)
```

a funzione accumula (detta anche fold o reduce) applica una funzione ad un elemento di una lista ed al risultato (ricorsivo) dell'applicazione di accumula al resto della lista:

```
(defun accumula (f iniziale lista)
    (if (null lista)
        iniziale
        (funcall f (car lista)
             (accumula f iniziale (cdr lista)))))
```

```
prompt> (accumula '+ 0 '(1 2 3))
6

prompt> (accumula '* 1 '(1 2 3 4))
24

prompt> (accumula 'cons NIL '(1 2 3))
(1 2 3)
```

I simboli i cui nomi iniziano con un due punti ":" sono detti keywords ed hanno se stessi come valore:

```
cl-prompt> :foo
:foo

cl-prompt> :forty-two
:forty-two
```

Le keywords sono usate estensivamente in Common Lisp e ci servono essenzialmente per definire delle funzioni con una sintassi di chiamata più interessante di quella semplice.
per ottenere dei risultati come i seguenti:

```
(list 1 2 3 4) -> (1 2 3 4)
(+ 1 1 1 1 1 1) -> 6
(< 1 2 3 4 55 66 77 100) -> T
```

posso usare le *lambda-list*:

```
(defun foo (a b c &rest l) (append l (list a b c)))
```

inoltre si hanno anche:

```
(subseq "qwerty" 2) -> "erty"
(subseq "qwerty" 1 4) -> "wer"
```

definibili con:

```
(defun foo (a &optional o) (cons a o))
```

I parametri opzionali possono essere inizializzati con un valore di default:

```
(defun fattoriale (n &optional (acc 1))
  (if (zerop n) acc (fattoriale (- n 1) (* n acc))))
```

```
cl-prompt> (fattoriale 4)
24

cl-prompt> (fattoriale 4 2)
48
```

n Common Lisp si possono definire delle funzioni che utilizzano i loro parametri associandoli a dei nomi, ovvero delle keywords:

```
cl-prompt> (find 2 '(1 2 3 4 5 6))
2

cl-prompt> (find 2 '(1 2 3 4 5 6) :start 3)
NIL

cl-prompt> (find 2 '(1 2 3 4 5 6) :end 3 :start 1)
2

cl-prompt> (sort '((a 1) (q 2) (d 3) (f 4)) 'string>
                 :key 'first)
((Q 2) (F 4) (D 3) (A 1))
```

Ovviamente si possono definire delle funzioni che accettano parametri a "chiave": basta usare la seguente sintassi nella lista di argomenti con cui si definisce una funzione:

```
(defun make-point (&key x y)
   (list x y))
```

```
cl-prompt> (make-point)
(nil nil)
```

```
cl-prompt> (make-point :y 42)
(nil 42)

cl-prompt> (make-point :y 42 :x -123)
(-123 42)
```

Parametri opzionali, a chiave e variabili vanno sempre dichiarati dopo quelli "obbligatori".

4.0.1 Input/output

Le due funzioni principale del Common Lisp per la gestione dell'I/O sono **READ** e **PRINT** e ad esse va associata la gestione di files e streams di I/O:

```
prompt> (read)
(foo 41 (42) 43 45) ; READ aspetta un input.
(FOO 41 (42) 43 45) ; Valore ritornato da READ

prompt> (third (read))
(foo 41 (42) 43 45) ; READ aspetta un input.
(42
```

Ovvero READ, legge un intero oggetto Lisp, riconoscendone la sintassi. La funzione PRINT stampa un oggetto Lisp rispettandone la sintassi.' Il valore ritornato da PRINT è il valore dell'oggetto la cui rappresentazione tipografica è appena stata stampata:

```
prompt> (print 42)
42 ; PRINT stampa la rappresentazione di 42
   ; (preceduta da un a-capo)

42 ; 'Lambiente Common Lisp stampa a video il
   ; valore 'dellapplicazione della funzione
```

```
    ; PRINT al valore 42

promtp> (print "HELLO WORLD"!)"
HELLO WORLD"!"
HELLO WORLD"!

prompt> (print (sqrt -1))
#C(0.0 1.0)
#C(0.0 1.0)
```

Il Common Lisp mette a disposizione la funzione FORMAT (simile alla fprintf C/C++, ed ai metodi format di varie classi Java):

```
prompt> (format t "Il fattoriale di ~D"~% 3 (fact 3))
Il fattoriale di 3 e` 6
NIL
```

la prima linea è la stampa della stringa "formattata"; il NIL è il valore ritornato da FORMAT

```
prompt> (format t "~S + ~S = ~S"~% '(1) '(2) (append '(1) '(2)))
(1) + (2) = (1 2)
NIL
```

Le direttive nella stringa da formattare sono introdotte dal carattere ~ (il carattere tilde), la direttiva D stampa numeri interi, la direttiva % va a capo, la direttiva S stampa un oggetto Lisp secondo la sua sintassi standard e la direttiva A stampa un oggetto Lisp secondo una sintassi esteticamente "piacevole":

```
prompt> (format t "~S e` una stringa"!~% ""foo)""
foo e` una stringa!
NIL
```

```
prompt> (format t "~A forse non e` una stringa"!~% ""foo)
foo forse non e` una stringa!
NIL
```

il T che appare come primo argomento a format è l'indicazione di "dove" andare a stampare; nella fattispecie su "standard output".
Si hanno a disposizione tre tipi di stream associati a tre variabili:

- standard input associato a *standard-input*
- standard output associato a *standard-output*
- standard error associato a *error-output*

```
prompt> (format *standard-output* "~S e`
                non una stringa"!~% ""foo)""
foo non e` una stringa!
NIL
```

Il T passato al posto di *standard-output* è una comodità.
Le funzioni READ, PRINT e FORMAT accettano un numero variabile di argomenti e uno di questi è uno stream (di output per FORMAT e PRINT e di input per READ):

```
prompt> (read *standard-input*)
qwerty
QWERTY

prompt> (print '(42 + 2 = 44 gatti) *error-output*)
(42 + 2 = 44 gatti)
(42 + 2 = 44 gatti)
```

Per leggere e scrivere da e su un file si usa la macro with-open-file (le macros in (Common) Lisp sono un utile strumento che permette di "estendere" il linguaggio; defun è solitamente implementata come una macro):

```
(with-open-file (<var> <file> :direction :input) <codice>)
(with-open-file (<var> <file> :direction :output) <codice>)
```

La variabile <var> viene associata allo stream aperto sul <file> e può venire utilizzata all'interno di <codice>. La macro with-open-file si preoccupa di chiudere sempre e comunque lo stream associato a <var> anche in presenza di errori (cfr., i meccanismi try ... catch () ... finally ... in Java, C++, etc.
Vediamo un esempio:

```
(with-open-file (out "foo.lisp"
                     :direction :output
                     :if-exists :supersede
                     :if-does-not-exist :create)
  (mapcar (lambda (e)
            (format out "~S" e))
          '((1 . A) (2 . B) (42 . QD) (3 . D))))

(with-open-file (in "foo.lisp"
                    :direction :input
                    :if-does-not-exist :error)
  (read-list-from in))

(defun read-list-from (input-stream)
  (let ((e (read input-stream nil 'eof)))
    (unless (eq e 'eof)
      (cons e (read-list-from input-stream)))))
```

Il secondo argomento a READ stabilisce che nessun errore debba essere generato quando si incontra la fine del file; in quel caso va invece ritornato il valore passato come terzo elemento (ovvero il simbolo EOF):

```
CL-USER 13 > (defun read-list-from (input-stream)
               (let ((e (read input-stream nil 'eof)))
                 (unless (eq e 'eof)
                   (cons e (read-list-from input-stream)))))
READ-LIST-FROM
```

```
CL-USER 14 > (with-open-file (out "foo.lisp
                                  :direction :output
                                  :if-exists :supersede
                                  :if-does-not-exist :create)
              (mapcar (lambda (e)
                        (format out "~S e))
                      '((1 . A) (2 . B) (42 . QD) (3 . D))))
(NIL NIL NIL NIL)

CL-USER 15 > (with-open-file (in "foo.lisp
                                 :direction :input
                                 :if-does-not-exist :error)
               (read-list-from in))
((1 . A) (2 . B) (42 . QD) (3 . D))
```

L'ambiente Lisp, o meglio la sua command-line, esegue tre operazioni fondamentali, ed ora che sappiamo qualcosa in più su input ed output possiamo esplicitarle:

- legge (READ) ciò che viene presentato in input, ciò che viene letto viene rappresentato internamente in strutture dati appropriate (numeri, caratteri, simboli, stringhe, cons-cells, ed altro ancora...)
- la rappresentazione interna viene valutata (EVAL) al fine di produrre un valore (o più valori)
- il valore così ottenuto viene stampato (PRINT)
- questo è il READ-EVAL-PRINT Loop (REPL)