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:

$$Programma = Composizione\ di\ Funzioni + Ricorsione$$

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)
        (cond((null list) nil)
                ((equal item (first list)) T)
                (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) {
        return 2 * x;
}

/* main con la chiamata al metodo*/
        int 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:

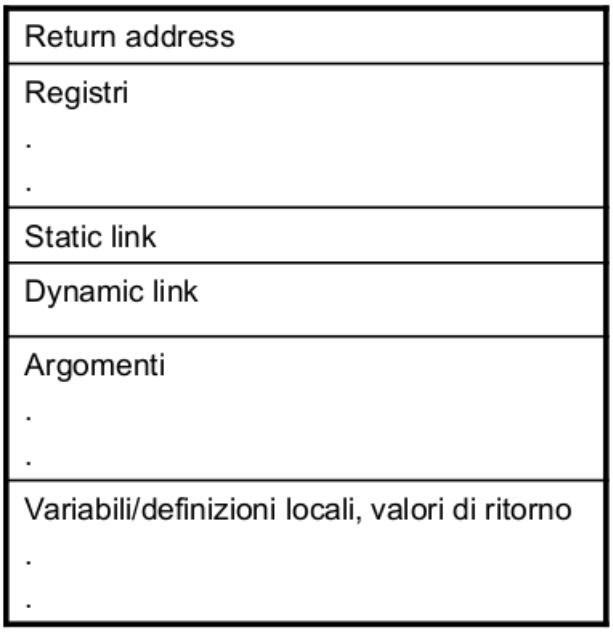
1. mettere i parametri in un posto dove la procedura possa recuperarli
2. trasferire il controllo alla procedura
3. allocare le risorse (di memorizzazione dei dati) necessarie alla procedura
4. effettuare la computazione della procedura
5. mettere i risultati in un posto accessibile al chiamante

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

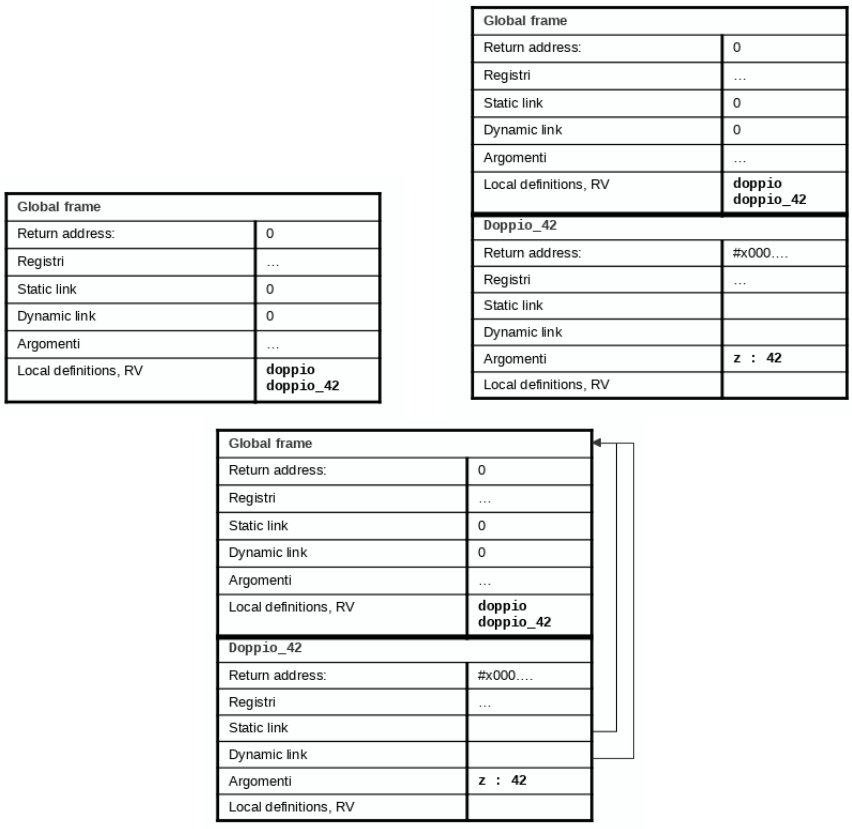
Analizziamo meglio l'*activation frame* di una procedura mediante la seguente immagine:



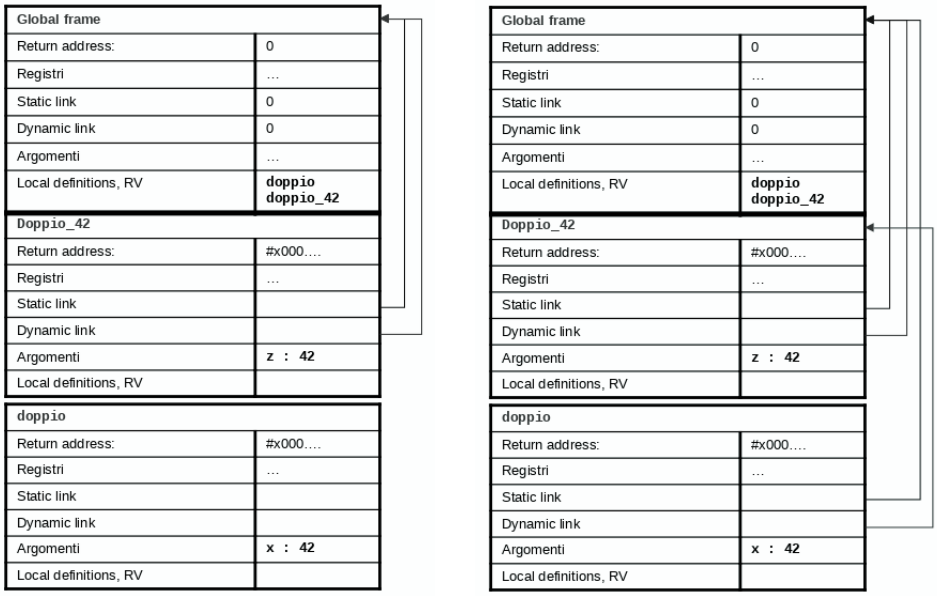
Partiamo ora dal seguente codice in *C*, e analizziamo i passaggi che avvengono nello stack:

```c
/* prima procedura */
int doppio(x) {
        return 2 * x;
}

/* seconda procedura, che chiama la prima */
int doppio_42(z) {
        return 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	

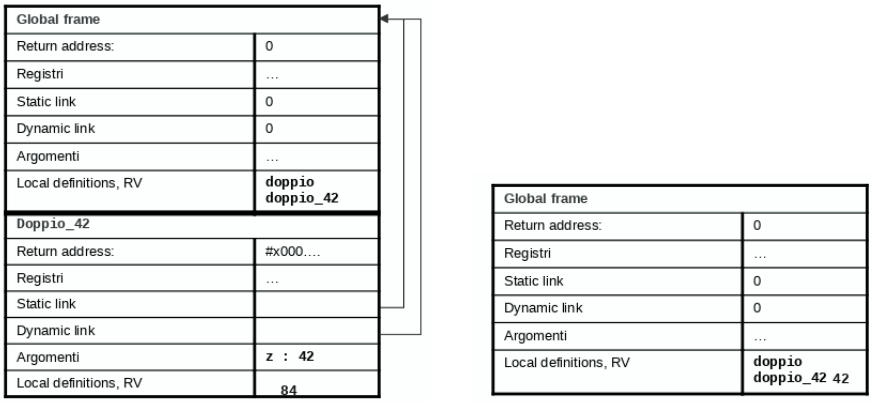
doppio	
Return address:	#x000….
Registri	…
Static link	
Dynamic link	
Argomenti	**x : 42**
Local definitions, RV	

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

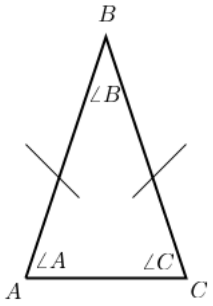
Si ricordano le seguenti due definizioni:

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.*

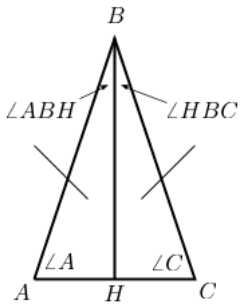


Dimostrazione. Si comincia la dimostrazione con l'elenco delle conoscenze pregresse:

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*** $\vee$*:*

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

- ***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) :-
        corso(Corso, _, _, Edificio, Aula, _).
docente(Corso, Docente) :-
        corso(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,
        orario(lunedì, '14:30'),
        aula('U4', 3),
        antoniotti).
```

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.

Capitolo 4

Laboratorio

4.1 Laboratorio 1

Esercizio 1. *Esempio base:*

```
lavora_per(bill, google). %questo è un fatto
lavora_per(mario, oracle).
lavora_per(steve, apple).  % con ; scorro le opzioni
lavora_per(jane, google).

collega(X, Y) :- lavora_per(X, Z),
                 lavora_per(Y, Z),
                 X \= Y. /* questa è una regola e con \=
                            evita che X = X sia un risultato
```

si possono avere le seguenti richieste con i seguenti output:

```
?- lavora_per(X, google).
X = bill ;
X = jane.

?- lavora_per(bill, X).
X = google.

?- collega(bill, jane).
true.
```

Esercizio 2. *esercizio 2, logica dei Naturali:*

```
/* definisco i naturali: */
nat(0). % 0 è naturale e lo sono anche i successori di un naturale
nat(s(N)) :- nat(N). /* s(N) indica il successore,
                        in compilatore sarà s(s(0)) etc...*/

/* definisco la somma come n + m = (n+1) + (m-1) se m>0 n + 0 = n */

succ(N, s(N)). % defisco il successore
sum(N, 0, N). % il terzo è il risultato
sum(N, s(M), s) :- sum(N, M, T),
                   succ(T, s).

/* si può anche fare così; */
summ(N, 0, N).
summ(N, s(M), s(T)) :- summ(N, M, T).
summ(N, M, S) :-  summ(s(N), P, S),
                  succ(P, M).
```

si hanno i seguenti output:

```
?- nat(0).
true.

?- nat(s(0)).
true.

?- sum(s(s(0)), 0, X).
X = s(s(0)) ;
false.

?- summ(s(0), 0, X).
X = s(0).
```

Esercizio 3. *Calcolo il fattoriale:*

```
fact(0, 1). % fattoriale di 0 è 1, caso base
fact(N, M) :- N>0,
              N1 is N-1,
              fact(N1, M1),
              M is N * M1. /* ogni volta salva in M1 il
                              risultato parziale, N>0
                               porterà a non fare il caso
                                base*/
```

```
?- fact(3, X).
X = 6 ;
false.
```

Esercizio 4. *Primi esercizi sulle liste:*

```
/* trovo primo elemento della lista */
intesta(X, [X|_]). /* | separa testa e coda (che non
                      interessa e indico con _,
                       senza non si sa che la lista può
                        continuare. Interrogo con, per
                         esempio  intesta(2, [2,3]). */

/* trovo ennesimo elemento*/
nth(0, [X|_], X). % se cerco lo 0, caso base
nth(N, [_|T], X) :- N1 is N-1,
                    nth(N1, T, X). /* se non è all'inizio
                                      cerco nella coda ad
                                       ogni passo la head
                                        aumenta di uno
                                        tengo solo T che è
                                         la coda, ovvero
                                         tutto tranne il
                                          primo*/

/*lista contiene l'elemento X?*/
contains(X, [X|_]). % controllo testa
contains(X,[_|T]) :- contains(X, T). % controllo ricorsivcamente la coda
```

Si hanno i seguenti output:

```
?- intesta(2, [2, 3]).
true.

?- intesta(2, [1, 3]).
false.

?- nth(0, [1,2,3], X).
X = 1 ;
false.

?- nth(2, [1,2,3], X).
X = 3 ;
false.

?- nth(18, X, 1).
X = [_594, _600, _606, _612, _618, _624|...] ;

?- contains(5, [5, 6, 7]).
true ;
false.

?- contains(4, [5, 6, 7]).
false.
```

dove l'ultimo risultato di nth ci dice che non può fare nulla con l'istruzione data, cerca all'infinito una risposta senza trovarla. Il primo di contains da false ad una seconda richiesta di risultato in quanto prova ad usare un'altra regola il compilatore di prolog

Esercizio 5. *Ancora sulle liste:*

```
/* funzione append per concatenare */
append([], L, L). % vuota più L = L
append([H|T], L, [H|X]) :- append(T, L, X).
/* l'inizio della lista finale è H e la fine è la fine
    delle liste concatenate */

/* chiedo se una lista  è ordinata */

sorted([]). % lista vuota ordinata
sorted([_]). % lista di un elemento è ordinata
sorted([X, Y| T]) :- X =< Y,
                     sorted([Y| T]). /* confronto sempre l'elemento col
                     resto della tail */

/* chiedo ultimo elemento */

last([X], X). %lista di un elemento ha come ultimo quell'elemento
last([_|T], X) :- last(T, X). /*controllo ricorsivamente
                                 la coda della lista finché
                                  non ho solo T e posso usare
                                   il caso base */
/* posso anche chieder e una lista che finisca con un certo N
                                 per esempio 4  last(X, 4). */

/* tolgo tutte le occorrenze */

remove_all([], _, []). % la lista vuota non ha nulla da rimuovere */
remove_all([X|T], X, L) :- remove_all(T, X, L).
/* se la testa è quel numero rimuovo tutte
    le occorrenze... ma se la testa \= X non va*/
remove_all([H|T], X, [H|L]) :- H \= X,
                               remove_all(T, X, L).
                               /* se H\=X  faccio il
                                   controllo senza H */

/* somma elementi lista */

somma_lista([], 0).
somma_lista([H|T], X) :-  somma_lista(T, N),
```

```
                                X is H + N.
/* sommo tutte le tail ricorsivamente e poi ci sommo la H */

/* ricordiamo le basi delle code */
coda([_|T], T).

/* duplico lista [1,2]->[1,1,2,2] */

duplico([], []).
duplico([H|T], [H, H|X]) :- duplico(T, X).
/* a priori duplico H riscivendo nel risultato
    poi duplico il tail, dove di volta in volta
     ogni head verrà duplicato. Se inverto e metto
      ( x, [lista]) mi toglie i duplicati*/
```

si hanno i seguenti output:

```
?- contains(5, [5, 6, 7]).
true ;
false.

?- contains(4, [5, 6, 7]).
false.

?- append([4], [5, 6, 7], X).
X = [4, 5, 6, 7].

?- append([1, 2, 3], X, [1, 2, 3]).
X = [].

?- append([1, 2, 3], X, [1, 2, 3, 6]).
X = [6].

?- append(X, Y, [1, 2, 3, 6]).
X = [],
Y = [1, 2, 3, 6] ;
X = [1],
Y = [2, 3, 6] ;
X = [1, 2],
```

```
Y = [3, 6] ;
X = [1, 2, 3],
Y = [6] ;
X = [1, 2, 3, 6],
Y = [] ;
false.

- sorted([1,2,1]).
false.

?- sorted([1, 2, 1]).
false.

?- sorted([1, 2, 1]).
false.

?- last([1, 2, 3], 3).
true.

?- last([1, 2, 3], X).
X = 3.

?- remove_all([1, 2, 1], X, L).
X = 1,
L = [2] ;
false.

?- somma_lista([1, 2, 3, 4], X).
X = 10.

coda([1,2,3], X).
X = [2,3]

?- duplico([1, 2, 3], X).
X = [1, 1, 2, 2, 3, 3].
```