

# Linguaggi di Programmazione

DaveRhapsody

Jacopo De Angelis

DlcGold

30 Settembre 2019

# Indice

<b>1</b>	<b>Introduzione al corso</b>	<b>3</b>
1.1	Programma del corso . . . . .	3
1.1.1	Logica Matematica e Linguaggi logici (Prolog) . . . . .	3
1.1.2	Linguaggi funzionali e Lisp (et al.) . . . . .	3
1.1.3	Linguaggi imperativi . . . . .	3
1.2	Modalità d'esame . . . . .	3
1.2.1	subsection name . . . . .	3
1.2.2	Prove parziali . . . . .	3
1.3	Appelli regolari . . . . .	4
<b>2</b>	<b>Il paradigma</b>	<b>5</b>
2.1	Cos'è? . . . . .	5
2.1.1	Storicamente . . . . .	5
2.1.2	L'effetto collaterale . . . . .	5
2.2	Logica del primo ordine . . . . .	5
2.3	Linguaggi funzionali . . . . .	5
2.4	Paradigma imperativo . . . . .	6
2.4.1	Il concetto di variabile . . . . .	6
2.5	Modello di Von Neumann . . . . .	6
2.6	Stile prescrittivo . . . . .	6
2.7	Concetto di programma . . . . .	6
2.8	Perchè utilizzare paradigmi diversi? . . . . .	7
2.9	Paradigma logico . . . . .	7
2.10	Esempio di un programma Prolog . . . . .	7
2.10.1	Esempio: . . . . .	8
2.10.2	Esempio dell'ordine di una lista . . . . .	8
2.11	Paradigma funzionale . . . . .	8
2.11.1	Composizione di funzioni + ricorsione . . . . .	9
2.12	LISP . . . . .	9
2.12.1	Esempio di programma LISP . . . . .	9
2.13	Ambienti RunTime di linguaggi logici funzionali e non . . . . .	10
2.13.1	Activation frame . . . . .	11
2.14	Activation Frame di una funzione . . . . .	11
2.15	Heap e Garbage Collector . . . . .	11
<b>3</b>	<b>Logica e ragionamento</b>	<b>12</b>
3.0.1	Regole di inferenza . . . . .	12
3.1	Dimostrazione . . . . .	13

3.2	Logica Proposizionale . . . . .	13
3.3	Principio di risoluzione . . . . .	14
3.4	Unit Resolution . . . . .	14
3.4.1	Esempio di dimostrazione per assurdo . . . . .	15
3.5	Ricapitolando . . . . .	15
3.6	L'assioma . . . . .	16
3.6.1	L'esempio dell'unicorno . . . . .	16
3.6.2	Tautologie e modelli . . . . .	17
3.7	La logica del primo ordine . . . . .	17
3.7.1	Le Formule Ben Formate nella logica del primo ordine . . . . .	18
3.7.2	Calcoli logici . . . . .	19
3.8	Altre regole in Logica del Primo Ordine . . . . .	19
<b>4</b>	<b>Prolog</b>	<b>21</b>
4.1	Programmazione logica . . . . .	21
4.1.1	Logica Matematica . . . . .	21
4.2	Cos'è ProLog . . . . .	22
4.2.1	Caratteristiche di Prolog . . . . .	22
4.3	Formule Ben Formate e Forma Normale . . . . .	22
4.3.1	Forma normale congiuntiva . . . . .	22
4.3.2	Linguaggio dichiarativo . . . . .	22
4.4	Sintassi Prolog . . . . .	23

# Capitolo 1

## Introduzione al corso

### 1.1 Programma del corso

Il corso è volto ad insegnare dei paradigmi di programmazione dei seguenti tipi:

#### 1.1.1 Logica Matematica e Linguaggi logici (Prolog)

Termini, fatti(predicati), regole, unificazione, procedura di risoluzione

#### 1.1.2 Linguaggi funzionali e Lisp (et al.)

Atomi, liste, funzioni e ricorsione

#### 1.1.3 Linguaggi imperativi

Memoria, stato, assegnamenti, puntatori

---

Il concetto è che con questo corso si vanno a studiare paradigmi più evoluti, usati tutt'ora e comunque aventi un ampio approccio logico, oltretutto LISP è usato nelle pagine web (Si userà moltissimo la ricorsione, A I U T O)

### 1.2 Modalità d'esame

#### 1.2.1 subsection name

- il voto finale sarà una media pesata dei voti conseguiti nell'esame relativo alla parte teorica e nell'esame del progetto
  - Occhio, il peso è a discrezione dei prof

#### 1.2.2 Prove parziali

Le prove d'esame sono costituite da uno scritto di 6-10 domande, e da un progetto da consegnare entro una data prefissata

### 1.3 Appelli regolari

Gli appelli regolari sono composti da un progetto ed un esame scritto, che può essere seguito da un esame orale a discrezione del docente basato sui temi trattati durante il corso

**NON C'E' POSSIBILITA' DI RECUPERI**, infatti scritto, orale e progetto vanno sostenuti **NELLO STESSO APPELLO**

Progetto e scritto sono corretti separatamente

**NON CI SARANNO ECCEZIONI** Lo avete già letto nel passaggio precedente, ma lo ripeto lo stesso perchè deve essere chiaro che **N O N S I F A N N O E C C E Z I O N I**.

# Capitolo 2

## Il paradigma

### 2.1 Cos'è?

E' il metodo di soluzione ad un determinato problema, a seconda dei paradigmi si hanno diversi tipi di linguaggi di programmazione

#### 2.1.1 Storicamente

Il primo paradigma è l'imperativo, cioè il paradigma basato sui tre costrutti di selezione, iterazione e sequenza.

Inoltre si mantiene il concetto di assegnamento di un valore ad una determinata variabile

#### 2.1.2 L'effetto collaterale

Viene definito effetto collaterale quando, a seguito dell'esecuzione di un qualsiasi codice, il contenuto di un'area di memoria viene cambiato; per intenderci, anche solo l'istruzione "x += 1" genera un effetto collaterale, poichè nell'area di memoria di x viene cambiato il valore.

Perchè è importante tutto ciò, direte. Semplice: il paradigma puro funzionale si basa proprio sul fatto che un programma non generi mai, mai, *M A I*, effetti collaterali. Successivamente vedremo che in Prolog ci saranno parecchi problemi se provassimo ad assegnare direttamente un valore ad una variabile

### 2.2 Logica del primo ordine

Prolog è costituito da una serie di clausole derivanti dalla logica del primo ordine

### 2.3 Linguaggi funzionali

Questi si basano proprio sui concetti matematici di funzione, ad esempio si ragiona sui domini, sui codomini, sull'insiemistica, solite cose. La loro caratteristica è che ogni funzione, dato sempre lo stesso input, restituisce sempre lo stesso risultato. Cosa vuol dire questo? Che non dipende da variabili esterne (e da qui l'importanza degli effetti collaterali, evitati nei linguaggi funzionali).

## 2.4 Paradigma imperativo

Le caratteristiche essenziali dei linguaggi imperativi sono legate all'architettura di Von Neumann, costituita dai famosi due componenti **Memoria (componente passiva)** e **Processore (componente attiva)**

In pratica la principale attività che ha la cpu è quella di eseguire calcoli ed assegnare valori alle variabili, che sono delle celle di memoria.

**Va considerato** Il concetto di variabile è un'astrazione di una cella di memoria, per dire se giochi su assembly vai a toccare i veri e propri registri, mentre su C o Assembly si ragiona per nome di variabile, non vai di indirizzamento fisico

### 2.4.1 Il concetto di variabile

In Prolog e LISP cambia completamente il concetto di variabile, ma per come saranno presentati vedremo che non c'entra niente.

In matematica abbiamo il concetto di variabile? Sì, quella che sta dentro una funzione, in informatica è diciamo diverso, non è un'astrazione, ma lo vedremo in seguito

## 2.5 Modello di Von Neumann

Per manipolare la memoria utilizzo la variabile, simbolo che indica la cella di memoria, nei linguaggi funzionali sarà possibile usare il concetto di variabile matematica.

Alla fine il modello di Neumann è composto da I/O, Memoria e CPU con i suoi cicli di clock

## 2.6 Stile prescrittivo

Un programma scritto in un linguaggio imperativo prescrive le operazioni che la CPU deve eseguire per modificare lo stato di un sistema

Le istruzioni sono eseguite nell'ordine in cui queste appaiono, ad eccezione delle strutture di controllo

**Realizzati** sia attraverso interpretazione che compilazione, nati più per manipolazione numerica che simbolica.

## 2.7 Concetto di programma

Un programma è intendibile come un insieme di algoritmi e di strutture dati ma la struttura di un programma consiste in

- Una parte dichiarativa in cui son presenti le dichiarazioni di tutte le variabili del programma e del loro tipo

- Una parte che descrive l'algoritmo risolutivo utilizzato, mediante istruzioni del linguaggio

## 2.8 Perchè utilizzare paradigmi diversi?

Per esempio l'intelligenza artificiale si sviluppa su linguaggi di programmazione specifici, bisogna usare linguaggi che operino in un determinato modo, considerati tipo di Altissimo super mega galatticissifantastico livello infatti, utilizzabili pure da non programatori

Infatti son generati per manipolazione simbolica non numerica

## 2.9 Paradigma logico

Concetto primitivo: Deduzione logica, avente una base di logica formale e un obbiettivo, che è intendibile come formalizzazione del ragionamento

**Programmare infatti significa** descrivere il problema con frasi (Formule logiche) del linguaggio,

Interrogare il sistema, che effettua deduzioni in base alla "conoscenza rappresentata"

**Ai lettori** Mi rendo conto che non si capisca un cazzo, voi immaginatevi come mi stia sentendo al momento io mentre prendo appunti.. Perdonatemi

Prolog è un insieme di formule ben formate, ragiona con il linguaggio logico, con una descrizione della realtà di interesse, di fatto è una dimostrazione in un linguaggio logico che costituisce un programma. Più semplicemente ho una frase da dare al mio interprete, Prolog icchè fa? Semplicemente la realizza sotto forma di dimostrazione.

## 2.10 Esempio di un programma Prolog

Ci sono fondamentalmente:

- Asserzioni incondizionate (**fatti**) A.
- Asserzioni condizionate (**regole**)  $A :- B, C, D, \dots, Z$ .
  - A è la conclusione o conseguente (deve avere una sola clausola)
  - B, C, D, ..., Z sono le premesse o antecedenti
- Un'interrogazione ha la forma:  $:- K, L, M, \dots, P$ .

Ovviamente A, B, C, \*TUTTE LE ALTRE\*, sono semplicemente predicati  
 MI RACCOMANDO MASSIMA ATTENZIONE ALLA SINTASSI, ogni clausola Prolog termina con un punto.

La ',' si legge come AND



### 2.10.1 Esempio:

Due individui sono colleghi se lavorano per la stessa ditta/azienda **Regole** **Fatti** Interrogazione

```
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).
```

Programmare Prolog non è come scrivere in un linguaggio di programmazione, non si scrive un algoritmo, in questo caso abbiamo le famose clausole, (regole e fatti),

**ATTENZIONE** l'interrogazione non è una clausola, occhio a non confondersi

### 2.10.2 Esempio dell'ordine di una lista

- **Ordine prescrittivo:** Controlla se la lista è vuota, e dà come risultato la lista vuota stessa, altrimenti calcola una permutazione della lista e controlla se è ordinata, dando come risultato  $L_1$  altrimenti fa una permutazione su  $L$  etc.

Il programmatore deve specificare le istruzioni che generano la sequenza di permutazioni della lista  $L$   $\left\{ \begin{array}{l} \text{il risultato dell'ordinamento di una lista vuota è la lista vuota} \\ \text{Il risultato dell'ordinamento di una lista } L \text{ e } L_1 \end{array} \right.$

Quindi, lo stile prescrittivo presuppone che OGNI SINGOLO CASO venga considerato e programmato. non esiste il "ma è ovvio che debba fare questo", ogni singolo caso è tua responsabilità. Sì, tua, proprio tu che stai leggendo. Prolog si basa su questo tipo di ordine.

- **Stile Dichiarativo:** L'ambiente si fa carico di generare possibili permutazioni della lista  $L$ , secondo deduzione matematica

## 2.11 Paradigma funzionale

- Si basa sul concetto di funzione matematica, ossia una associazione tra due insiemi che relaziona ad ogni elemento di un insieme (dominio) un solo elemento di un altro insieme (codominio)
- La definizione di una funzione specifica dominio, codominio, e regola di associazione

- **ESEMPIO:**  
 $\text{Incr}: \mathbb{N} \rightarrow \mathbb{N}$   
 $\text{Incr}(x) = x + 1$
- Dopo aver dato definizione, una funzione è applicabile ad un elemento del dominio (argomento) per restituire l'elemento del codominio ad esso associato (valutazione)
- $\text{incr}(3) \rightarrow 4$  L'unica operazione utilizzata nel funzionale è l'applicazione di funzioni
- Il ruolo dell'esecutore di un linguaggio funzionale si esaurisce nel calcolare l'applicazione di una funzione (il programma) e produrre un valore
- Nel paradigma funzionale puro il valore di una funzione è determinato dagli 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
- Oggettivamente si ha l'assenza di effetti collaterali

**Attenzione** Il concetto di variabile che utilizziamo è quello di "costante" matematica, in cui i valori NON sono mutabili, non ho nessun assegnamento

L'essenza della programmazione funzionale consiste nel combinare funzioni mediante composizione e uso della ricorsione

### 2.11.1 Composizione di funzioni + ricorsione

La struttura di un programma consiste nella definizione di un insieme di funzioni ricorsive mutualmente

L'esecuzione del programma consiste nella VALUTAZIONE dell'APPLICAZIONE di una funzione principale a una serie di argomenti

## 2.12 LISP

LISt Processing,

Il progetto originale era di creare un linguaggio funzionale puro, infatti nel corso degli anni sono stati sviluppati molti ambienti di sviluppo lisp di cui terremo in considerazione Common Lisp e Scheme, oltre che emacs etc.

### 2.12.1 Esempio di programma LISP

Controlla un elemento se appartiene ad 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))
```

Dopo una parentesi tonda ci va per forza una funzione, è fondamentale, defun definisce una funzione infatti, dopo c'è il nome di tale funzione. In LISP la tabulazione è ESSENZIALE, AUGURI A DISTINGUERE DOVE PORTI LA QUINTA PARENTESI DELLE DODICI CHE HAI SCRITTO PIANGENDO SUL CODICE ALLE 2 DI NOTTE.

Gli elementi si separano con lo spazio NON con la virgola, mi raccomando.

La terza riga è la più ostica e dice: è uguale T al primo elemento della lista? E' molto incastrato ma si riesce a capire, associamo per esempio ad item = 2 (E' un esempio) e list = [1,2]

Noi ci arriviamo con la logica che c'è, ma in realtà cosa faremo? Ragioniamo per gradi

1. null list? **false**
2. è 2 = al primo elemento della lista? **False**
3. è 2 = al secondo elemento della lista? **True**

Se LISP trova T è come se scrivessimo **true**

## 2.13 Ambienti RunTime di linguaggi logici funzionali e non

- Richiami di nozioni di architettura e programmazione
- Per eseguire un programma di qualsiasi linguaggio il sistema operativo deve mettere a disposizione l'ambiente runtime che dia almeno due funzioni
  - Mantenimento dello stato della computazione(pc, limiti di memoria)
  - Gestione memoria disponibile (fisica e virtuale)
- L'ambiente runtime può essere una vera e propria macchina virtuale tipo la JVM di java
- In particolare la gestione di memoria avviene usando due aree concettualmente ben distinte con funzioni diverse
  - Lo stack, ambiente dell'ambiente runtime che serve per la gestione delle chiamate, a procedure metodi etc
  - L'heap dell'ambiente runtime serve per gestire strutture dinamiche
    - \* Alberi
    - \* Liste etc

### 2.13.1 Activation frame

- 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 (si passa tutto per valore, non esistono effetti collaterali)
- E' un altro modo di chiamare i record di attivazione, via
- Il corpo della procedura viene valutato (ricorsivamente) tenendo questi legami in maniera statica

cioè il concetto è che bisogna capire cosa accade con variabili che risultino libere in una sottoespressione

## 2.14 Activation Frame di una funzione

Contiene:

- Return address
- Registri
- Static / Dynamic link (lo statico punta alle variabili globali)
- Argomenti
- Local definitions (RV)

Se si ha in mente come funziona oggettivamente lo stack (con i record di attivazione) è la stessa cosa

All'esame si potrebbe chiedere cos'è l'activation frame e a icchè serve

## 2.15 Heap e Garbage Collector

L'heap è l'area di memoria destinata alla memorizzazione delle strutture per i dati dinamiche, mentre invece il garbage collector ha il compito di accumulare lo schifo che si accumula tra variabili non deallocate etc, e le dealloca appunto.

# Capitolo 3

## Logica e ragionamento

Partiamo con le cose semplici, bisogna passare da quello che è un linguaggio parlato a una stesura di condizioni

Prendiamo un triangolo, vogliamo dimostrare che se due triangoli hanno i due lati uguali allora è isoscele

$$AB = BC \vdash \angle A \angle C$$

1.  $AB = BC$  per ipotesi
2.  $\angle ABH = \angle HBC$  per (3)
3. Il triangolo  $HBC$  è uguale al triangolo  $HBC$   $ABH$  per (2)
4.  $\angle A = \angle C$  per (1)

### 3.0.1 Regole di inferenza

Esempio di una regola di inferenza:

$$\frac{F_1, F_2, F_3, \dots, F_n}{R}$$

1. Introduzione della congiunzione (L'AND)
2. Modus Ponens
3. Eliminazione della congiunzione

Come lavora il **Modus Ponens**:

E' semplice manipolazione sintattica, osservando la formula che ci vien data possiamo riscriverla scrivendo come base di conoscenza il conseguente, cioè in pratica prendo e sostituisco con il conseguente.

Per far si che le mie formule siano vere, se avessi un  $A$  or  $B$  può esser vera in ben tre casi diversi, non posso eliminare i due casi disgiunti, se voglio mantenere una solidità non posso, e quindi questa regola (disgiunzione) non esiste.

**Attenzione** in una dimostrazione non si può dare nulla per scontato, tutto ciò che noi diamo per assodato, un pc non lo dà, dobbiamo essere molto precisi nelle indicazioni, bisogna lavorare in un'ottica più precisa

cerchiamo ora di tradurre tutto in un linguaggio più formale SE  $AB = BC$  E  $BH = BH$  e  $ABH = HBC$  allora il triangolo  $ABH$  è uguale a  $HBC$  ed abbiamo trasformato (1) in

SE triangolo  $ABH$  è uguale al triangolo  $HBC$  ALLORA  $AB = BC$  e  $BH = BH$  e  $AH = HC$ , E  $ABH = HBC$  E  $AHB = CHB$  E  $A = C$

Da un punto di vista formale noi partiamo da un'ipotesi, noi vogliamo dimostrare che  $A = C$ .

La dimostrazione è un processo sintattico, non ragiono in termini di verità, perchè NON si sta parlando di interpretazione ma manipolazione delle formula

Ogni passo deve corrispondere ad una formula, e subito dopo le etichette

**Differenza tra assiomi e ipotesi** Gli assiomi sono conoscenza pregressa del dominio, mentre le ipotesi sono solo supposizioni iniziali, uno si specifica, l'altro no

### 3.1 Dimostrazione

E' una sequenza di passi dove il finale è la formula da dimostrare e abbiamo un insieme di passi intermedi che possono essere presi dalle conoscenze pregresse, oppure applicando regole di inferenza ai passi PRECEDENTI, solo precedenti mi raccomando.

Le regole di inferenza sono applicabili solo ai passi precedenti rispetto ad una formula

### 3.2 Logica Proposizionale

Nella logica proposizionale ci si occupa delle conclusioni che possiamo trarre da un insieme di proposizioni, abbiamo infatti un insieme  $P$  di proposizioni

Si introduce il concetto di **interpretazione** di un insieme di proposizioni, infatti all'insieme  $P$  si associa una funzione di verità (True e False)

Questa funzione associa un valore di verità ad ogni elemento di  $P$ , ad ogni proposizione. La valutazione è il ponte tra sintassi e semantica di un linguaggio

Chiaramente posso legare tra loro le proposizioni con  $\vee$  e  $\wedge$  e  $\neg$  Una formula ben formata è un insieme di espressioni sintatticamente corrette di un linguaggio

In prolog le formule atomiche le chiameremo letterali, che possono esser positivi e negativi, e qui si richiamano i concetti di fondamenti della tabella di verità

Negli esercizi potremo usare

$$\frac{F_1, F_2, \dots, F_K}{R}$$

E' la forma generale di una regola, sopra hai l'insieme delle formule vere tra le formule ben formate e R è la formula generata da "inserire" in FBF.

L'esempio di inferenza che si usa di solito è il Modus Ponens:

$$\frac{p \rightarrow q, p}{p}$$

Se il conseguente appare come formula negata, si negherà anche la formula originaria

### Esempio:

$\frac{p \vee \neg q}{vero}$  Terzo escluso

$\frac{\neg \neg q}{q}$  eliminazione  $\neg$

$\frac{p \wedge vero}{p}$  eliminazione  $\vee$

$\frac{p \wedge \neg p}{q}$  contraddizione

## 3.3 Principio di risoluzione

E' una regola di inferenza generalizzata semplice e facile da utilizzare ed implementare, in pratica opera su formule ben formate trasformate in forme normali congiunte, ed ognuno dei congiunti vien detto **clausola**

L'osservazione fondamentale alla base del principio di risoluzione è un'estensione della nozione di rimozione dell'implicazione su base della contraddizione

In pratica si usa per le dimostrazioni per assurdo.

## 3.4 Unit Resolution

E' un caso particolare di principio di soluzione;

Da un lato ho una formula ben formata disgiuntiva e dall'altro ho un letterale (o asserito), e una di queste formule è costituita da un solo letterale, per questo si chiama unit, è sintassi, non ci sto capendo più un cazzo pure. io, non so davvero che dirvi..

**Spiegazione fornita da** Jacopo De Angelis

Eccomi, arrivo da autore esterno a spiegare: prendiamo prima lo schemino semplice semplice

$P < - - A \wedge B \wedge C \wedge D...$

Questo vuol dire che P è vera solo se tutte A, B, C e D sono vere. Ora, prendiamo nuovamente il codice prolog visto all'inizio:

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

Cosa cambia dal dire questo o la regola sopra? La corrispondenza è semplicemente:

```
P è collega(X, Y) :-
    A è lavora(X, Z),
    B è lavora(Y, Z),
    C è diverso(X, Y).
```

Questo vuol dire che la nostra unit, P, è vera solo se sono vere le altre. Fine del mio intervento, la linea di nuovo a Dave.

### Esempio:

- <non piove>, <piove o c'è il sole>
- <c'è il sole>

La dimostrazione per assurdo di fatto funziona assumendo che la formula negata sia vera, se combinandola con le proposizioni in fbf ottengo una contraddizione, allora si può concludere con la verità della proposizione.

### 3.4.1 Esempio di dimostrazione per assurdo

Abbiamo una proposizione  $\lambda$  e dobbiamo dimostrare che essa sia vera:

Per dimostrarlo occorre porre per ipotesi che  $\neg\lambda$  sia vera e SE FBF  $U \neg\lambda$  genera una contraddizione, ALLORA  $\lambda$  è vera!

## 3.5 Ricapitolando

Quello che noi definiamo come "Calcolo Logico" delle proposizioni va a toccare

- Dal punto di vista della sintassi
  - Un insieme di proposizioni che chiameremo **P**
  - Un insieme di **FBF**, tale che  $P \subseteq \mathbf{FBF}$
  - Un sottoinsieme di assiomi  $A \subseteq \mathbf{FBF}$
  - Un insieme di regole di inferenza che ci permettono di incrementare **FBF**
- Dal punto di vista della Semantica
  - Una funzione di verità che consente di distinguere **true** e **false** rispetto alle tavole di verità o funzioni di interpretazione



## 3.6 L'assioma

Un assioma è una conoscenza che si dà per assodata, qualcosa di sicuramente vero, se vogliamo anche "scontato"

### 3.6.1 L'esempio dell'unicorno

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.

L'unicorno:

- E' mitico?
- E' magico?
- E' cornuto?

#### Procedimento

1. Esprimere il problema in forma di logica delle proposizioni
2. Individuare i teoremi da dimostrare
3. Dimostrare i teoremi

Cioè concettualmente devo cercare di individuare i predicati, e porli in modo più logico (Tipo mitico(x), magico(x) e cornuto(x)), MA MI RACCOMANDO ATTENZIONE.

**NON CONFONDERE** I termini a disposizione, se qui hai mortale o "Immortale" stai parlando di qualcosa che è Mortale  $\vee \neg$  Mortale.

Ora andiamo a risolvere questo esercizio, e si comincia con il dare un nome ai nostri predicati, che siccome sarebbero lunghi, saranno abbreviati in massimo 5 predicati

- UM = Mitico
- UI = Immortale
- UMag = Magico
- UC = Cornuto

Ritrascriviamo quindi la nostra frase iniziale:

$UM \rightarrow UI$ ,

$\neg UM \rightarrow \neg UI$ ,

$\neg UI \vee UI \rightarrow UC$

$UC \rightarrow UMag$

Mentre per quanto riguarda le domande poste:

1.  $S \vdash UM$  ?
2.  $S \vdash UMag$  ?

3.  $S \vdash UC$ ?

Risolvendo l'esercizio otteniamo che

Ora, il metodo per risolvere le  $\vdash$  potrebbe essere tramite negazione della nostra ipotesi. In che senso? Proviamo con il caso  $S \vdash UC$

- P1:  $\neg UI \vee UI \rightarrow UC$  (Da S)
- P2:  $\neg UI \vee UI$  (Che era stato dimostrato in un passaggio precedente con la dimostrazione di  $A \rightarrow (B \rightarrow C)$ )
- P3: UC (Da P1, P2 e modus ponens)

Proviamo ora a verificare per esempio se l'unicorno è magico. Riscriviamoci l'insieme  $S = \{UM \Rightarrow UI, \neg UM \Rightarrow \neg UI, \neg UI \vee UI \Rightarrow UC, UC \Rightarrow U_{mag}\}$

### 3.6.2 Tautologie e modelli

Una fbf che si verifica in ogni caso, è detta **Tautologia** (Fondamenti docet)

Una particolare **Interpretazione**  $V$  che rende vere tutte le formule in  $S$  viene detta **modello** di  $S$

## 3.7 La logica del primo ordine

Se la logica proposizionale si dimostra utile, avente caratteristiche di computazione che da questo punto di vista sono chiare, c'è da dire che la semantica è chiara allo stesso modo, però purtroppo non ci permette di fare asserzioni in merito ad insieme di elementi in maniera concisa.

**Ai lettori** Cioè io sto capendo logica adesso e non l'ho capita a Fondamenti.. Meditiamo ragazzuoli, meditiamo.

Sono Jacopo, vi parlo dal terzo anno. Fondamenti si capisce SOLO tramite gli altri corsi.

---

Con la **Logica del primo Ordine** introduciamo modi diversi di esprimere le proposizioni, prendiamo per esempio Socrate (Ricordate Palmonari? Ecco)

- Tutti gli uomini sono mortali
- Socrate è un uomo
- dalle precedenti ipotesi si deduce che Socrate è mortale

C'è un problemino di fondo, non si può esprimere in alcun modo qualcosa del tipo "Tutti gli uomini sono mortali"

Un linguaggio logico del primo ordine è costituito da **termini** che in pratica si costruiscono con

- V: insieme di simboli di variabili
- C: insieme di simboli di costante
- R: insieme di simboli di relazione o predicati (di qualsiasi arietà)
- F: insieme di simboli di funzione (di qualsiasi arietà)

**Ah, giusto** L'arietà sarebbe il numero di argomenti di una relazione o predicato. (Chiaramente anche funzioni, in quanto caso particolare di relazione)

Inoltre si hanno i connettivi logici, ovvero  $\forall$  (Per ogni) e  $\exists$  (Esiste).

Con la logica di primo ordina si ragiona diversamente, infatti il linguaggio è costruito ricorsivamente, ed i termini minimi sono detti **PREDICATI**

### 3.7.1 Le Formule Ben Formate nella logica del primo ordine

Qui si fa menzione della definizione **Ricorsiva** di Formula Ben Formata cioè:

$$FBF = \{t_j, r_{t_1, \dots, t_k}\}$$

Dove  $t_j$  è un termine elemento di C, di V, oppure un'applicazione di una funzione  $f(t_1, \dots, t_s)$  mentre

$$r(t_1, \dots, t_k)$$

Che sarebbe un termine costituito da un predicato (dove le t derivano dai termini appartenenti alle FBF)

Contiamo che diversi elementi di FBF connessi dai connettivi  $\forall, \exists, \neg, \rightarrow$  appartengono ad FBF

**Si denota**  $t(t_1, \dots, t_s)$  tale combinazione di termini

Grazie alle definizioni precedenti possiamo andare a risolvere l'esempio di Socrate.

Iniziamo dalle cose semplici, definiamo chi sono le costanti (insieme C)  $\rightarrow$

Socrate è una costante, pertanto apparterrà all'insieme C:  $C = \{\text{Socrate}\}$ , e poi possiamo definire quello che sono i predicati, che sono in questo caso uomo e mortale.

I predicati appartengono all'insieme R;  $R = \{\text{uomo. mortale}\}$

Benissimo, proviamo ora a realizzare la frase "Tutti gli uomini sono mortali":

Se tutti gli uomini son mortali, significa che per ogni elemento x tale che esso sia un uomo, si implica che x sia mortale.

**Osservazione** Quando dico "Sia un uomo" e "Sia mortale", intendo che la funzione  $uomo(x)$  e  $mortale(x)$  siano tendenzialmente delle booleane

Infatti ritrascrivendolo vien fuori:  $\forall x, (uomo(x) \rightarrow mortale(x))$ , tenendo conto che Socrate è un uomo, pertanto si dirà  $uomo(Socrate)$ , che se pensate al booleano, sì, darà **true**.

### 3.7.2 Calcoli logici

Per ottenere il risultato è necessario che si utilizzino delle regole di calcolo

#### Regola di eliminazione del quantificatore universale

Il discorso è: Come si risolve il  $\forall$ ?

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

Si ok, ma come lo realizziamo?

Con questa super mega iper formula possiamo finalmente derivare la nostra conclusione a partire dalle asserzioni iniziali.

1.  $uomo(Socrate)$
2.  $\forall x, uomo(x) \rightarrow mortale(x)$
3.  $mortale(Socrate)$

Andiamo a scrivercela sostituendo alla formula  $\rightarrow$

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

Ora andiamo a togliere il quantificatore universale ( $\forall$  detto in modo figo)

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

Cosa notiamo? Il "denominatore" della prima formula è diventato il "numeratore" della seconda, infatti vedete come abbiamo  $uomo(Socrate) \rightarrow mortale(Socrate)$ ?

A questo punto bisogna togliere anche l'implicazione, o meglio risolverla. Riflettendoci  $A \rightarrow B$  a cosa è uguale?  $\neg A \wedge \neg B$ , in questo modo abbiamo risolto l'implicazione.

## 3.8 Altre regole in Logica del Primo Ordine

Abbiamo risolto il  $\forall$ , ok, ma l' $\exists$ ? Introduciamo quindi il quantificatore esistenziale:

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

Per completezza possiamo dedurre il fatto che

$$\begin{cases} Se \exists x, \neg T(..., x, ...) = \neg \forall x, T(..., x, ...) \\ Se \forall x, \neg T(..., x, ...) = \neg \exists x, T(..., x, ...) \end{cases}$$

Spiegato peggio, se dico che esiste un elemento, per cui non si verifica una proprietà, allora non per ogni elemento essa si verifica.

Spiegato ancora peggio, se uno solo non si verifica, allora significa che non tutte si verificano

Nel secondo caso invece se dico che per ogni  $x$  una proprietà non si verifica, allora non esiste alcun  $x$  per cui si verifica

**Precisazione** In effetti negare il  $\forall$  si tradurrebbe in "non tutte le  $x$ ", mentre è molto più semplice per l' $\exists$  in cui si dice "non esiste", lo specifico perchè io mi ci confondevo spesso.

# Capitolo 4

## Prolog

Data questa confusissima introduzione giungiamo a trattare più nello specifico il linguaggio Prolog. Quando parliamo di Prolog, infatti, smettiamo di ragionare in modo **imperativo** per passare al paradigma di **programmazione logica**.

**Premessa:** C'è (come lo è stato per Prog 1) da scaricarsi l'interprete di Prolog, si consiglia SWI-Prolog, che si appoggia ad Emacs come editor di testo.

(Sia per la stesura di questi appunti che per gli esercizi io ho sempre usato Sublime Text MA all'esame molto probabilmente dovremo usare quel che ci impongono loro, pertanto, meglio abituarsi da subito ad Emacs.)

### 4.1 Programmazione logica

Se prolog utilizza il paradigma di **programmazione logica**, quest'ultima non è rappresentata solo da Prolog, ci sono ovviamente altri linguaggi che lo fanno. Perché scegliere Prolog?

- Il formalismo è più semplice
- E' un linguaggio ad alto livello
- La semantica è comprensibile

Quello che è il nostro programma divente un **insieme di formule** ed ha un enorme potere espressivo, mantenendo come chiave il fatto che la computazione effettiva è costruzione di una dimostrazione di una affermazione (Definita anche come goal, obbiettivo, meta).

#### 4.1.1 Logica Matematica

**Definizione Davis e Putnam:** Per logica matematica si intende la dimostrazione automatica di teoremi, secondo Davis e Putnam la logica matematica implica la dimostrazione dei teoremi

**Definizione Kowalski:** Interpretazione procedurale di formule. Quindi in pratica qua si entra già più nel concetto di un linguaggio di programmazione.

## 4.2 Cos'è ProLog

Nel titolo di questa sezione la L maiuscola non è a caso, perchè Pro-Log sarebbe acronimo di **PRO**gramming **LOG**ic, ed è un linguaggio che si basa su una restrizione della logica del primo ordine.

### 4.2.1 Caratteristiche di Prolog

- Si basa su una restrizione della logica del primo ordine
- Ha uno stile dichiarativo
- E' usato per determinare quando una affermazione è vera e quali vincoli abbiano fatto da discriminante tipo i vincoli sui valori da dare alle variabili che han generato la risposta

## 4.3 Formule Ben Formate e Forma Normale

Qualsiasi formula ben formata può essere riscritta in forma normale a clausole. Esistono due forme normali a clausole:

1. Forma congiunta:

La formula è una congiunzione di disgiunzioni di predicati o negazioni di predicati (letterali positivi e letterali negativi)

$$\bigwedge_i (\bigvee_j L_{ij})$$

2. Forma disgiunta:

E' una disgiunzione di congiunzioni di predicati o negazioni di predicati (letterali positivi e letterali negativi)

$$\bigvee_j (\bigwedge_i L_{ij})$$

### 4.3.1 Forma normale congiuntiva

La clausole che hanno al più un solo letterale positivo (sia con che senza letterali negativi) si chiamano cause di **Horn**. Però occhio: Non tutte le formula ben formate si riescono a far diventare un insieme di , che per la cronaca compongono i programmi Prolog o meglio, **I programmi prolog son collezioni di clausole di Horn**

### 4.3.2 Linguaggio dichiarativo

Uno dei punti di forza di Prolog è il fatto che sia un linguaggio dichiarativo, quindi è pressochè esente dall'avere istruzioni, contiene solo fatti e regole, che dal passaggio precedente sappiamo essere delle **clausole di Horn** .

**Ricordiamo che:** Un fatto è una asserzione vera nel contesto che si descrive, tipo assioma, mentre la regola è qualcosa che serve per dedurre dei nuovi fatti **partendo** da quelli esistenti.

Un programma scritto in Prolog ci dà informazioni su un sistema, e vien chiamato **base di conoscenza** (Il programma eh, non il sistema.)

Inoltre, (*già detto in precedenza ma va ripetuto perchè sicuramente anche io stesso quando andrò a studiare da qua mi sarò già dimenticato*) un programma Prolog **non si esegue**, ma si **interroga**.

Quali sono le possibili domande da fargli? Ad esempio:

- Questa serie di fatti è vera?  
E la risposta sarà Si o **No**, True o **False**, 1 o **0**, "Si broh" o "**No frate**".

## 4.4 Sintassi Prolog

- Fatto/Asserzione: *\*nomefatto\**.
- Regole:  $c :- b_1, \dots, b_n$ .
- Goal/Query:  $?-q_1, \dots, q_n$ .

Ogni lettera avente un pedice tipo  $p_\lambda$  o  $q_\kappa$  sono tutti termini composti, notare che in molte implementazioni il prompt Prolog è anche un operatore che chiede al sistema di valutare il **goal**

Ogni espressione Prolog diventa **TERMINE**, ne abbiamo diversi esempi:

- Atomi:  
E' una semplice sequenza di caratteri che inizia con carattere **Minuscolo** e può avere il '\_', oppure è un numero, o qualcosa racchiusa tra apici ('')
- Variabili
- COmposizioni di altri termini (Da qui termine composto)

**Precisazione** Ogni istruzione finisce con il punto, avete presente quando la vostra fidanzata capite che è innervosita con voi e alla fine delle frasi ci mette un '.' che vi fa gelare il sangue? Ecco, se qui non mettete il punto dopo ogni istruzione, vedrete come vi gireranno i *\*Censura\* ;))))*