

Linguaggi di Programmazione

UniShare

Davide Cozzi
@dlcgold

Gabriele De Rosa
@derogab

Federica Di Lauro
@f_dila

Indice

1	Introduzione	2
2	I linguaggi	3
2.1	Richiami di Architettura e Programmazione	7
2.2	Logica	12
2.2.1	Logica Proporzionale	15

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/dlccgold/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'architettura 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:

$$\textit{Programma} = \textit{Algoritmi} + \textit{Strutture Dati}$$

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 obiettivo 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:

$$\text{Programma} = \text{Conoscenza} + \text{Controllo}$$

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:

$A :- B, C, D, \dots, Z.$

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, \dots, Z implicano A ovvero $B, C, D, \dots, Z \rightarrow A$

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

$:- K, L, M, \dots, P.$

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 $X = \text{ciro}$ e $Z = \text{ibm}$. 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)
  (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:

Return address
Registri . .
Static link
Dynamic link
Argomenti . .
Variabili/definizioni locali, valori di ritorno . .

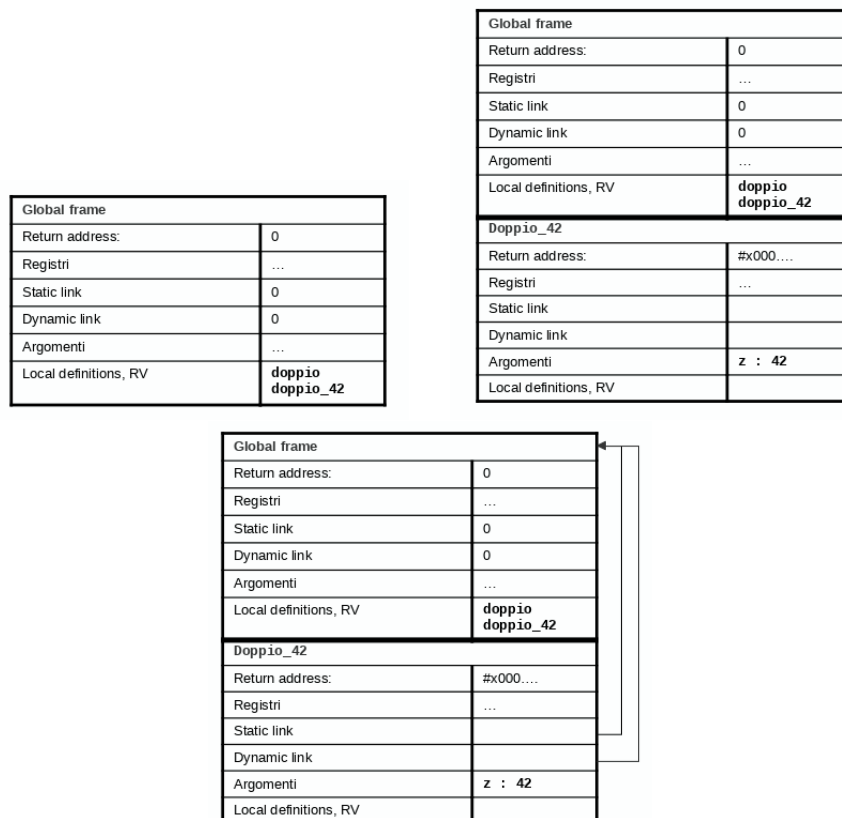
Partiamo ora dal seguente codice in *C*, e analizziamo i passaggi che avvengono nello stack:

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

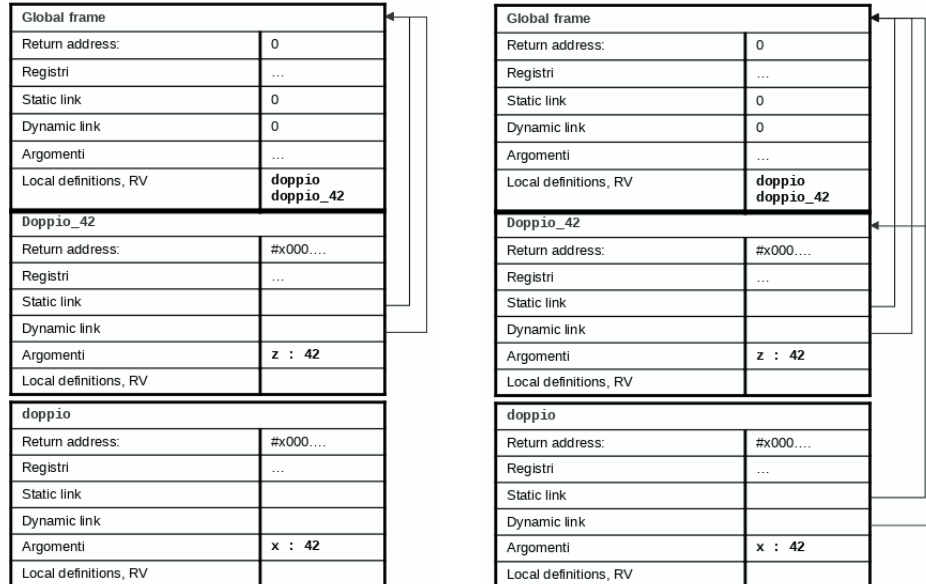
/* seconda procedura, che chiama la prima */
int doppio_42(z) {
    return doppio(z) - 42;
}

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

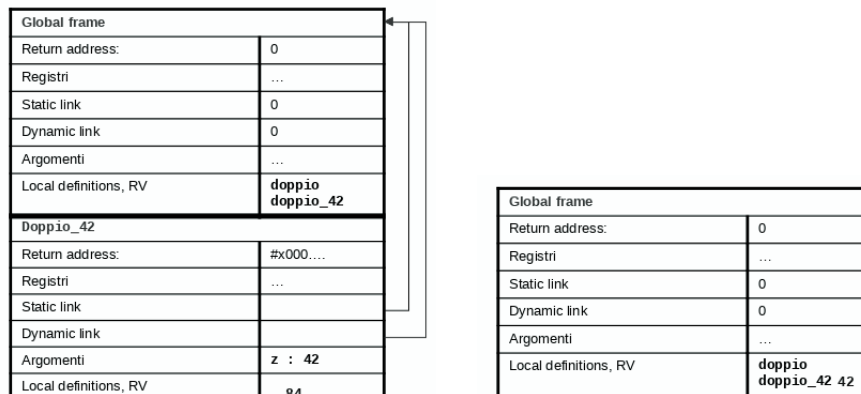
vediamo cosa accade nello stack: si definiscono in ordine *global frame* e *doppio_42*, con i collegamenti di static link e dynamic link:



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



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



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.

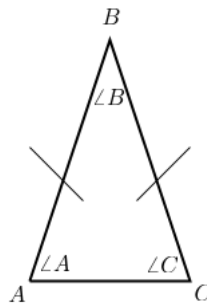
2.2 Logica

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)

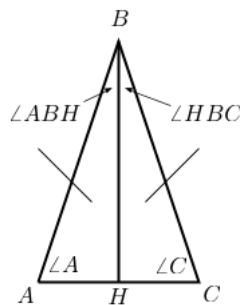
Partiamo con l'esempio di una semplice dimostrazione geometrica effettuata con le regole della logica:

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:

$$dim = \langle P_1, P_2, \dots, P_n \rangle$$

con:

- $P_n = F$
- $P_i \in S$ o con P_i ottenibile dalle P_1, \dots, 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 \mathbf{P} è associata una funzione di *verità*, o di *valutazione*, \mathbf{V} (spesso indicata con \mathbf{T} o \mathbf{I}):

$$V : P \rightarrow \{\text{vero}, \text{falso}\}$$

che associa un valore di verità ad ogni elemento di \mathbf{P} . La funzione di valutazione è il ponte di connessione tra sintassi e semantica in un linguaggio logico.

Le proposizioni si combinano con i seguenti connettivi:

- **coniunzione:** \wedge
- **disgiunzione:** \vee
- **negazione:** \neg
- **implicazione:** \rightarrow

L'insieme di tutte le formule formate dagli elementi di \mathbf{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 \mathbf{V} e questa definizione può essere estesa sul dominio **FBF**:

- $V(\neg s) = \text{non } V(s)$
- $V(a \wedge b) = V(a) \text{ e } V(b)$
- $V(a \vee b) = V(a) \text{ o } V(b)$
- $V(a \rightarrow b) = (\text{non } V(a)) \text{ o } V(b)$

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

a	b	$\neg a$	$\neg b$	$a \wedge b$	$a \vee b$	$a \rightarrow b$
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 basa su regole di inferenza. Una regola di inferenza ha la seguente forma:

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

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).

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

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).

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

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).

$$\frac{P_1 \wedge P_2 \wedge \dots \wedge P_n}{P_i} \text{ [Eliminazione di } \wedge]$$

ovvero:

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

$$\frac{P_1, P_2, \dots, P_n}{P_1 \wedge P_2 \wedge \dots \wedge P_n} \text{ [Introduzione di } \wedge]$$

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:**

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

- **Eliminazione di \neg :**

$$\frac{\neg \neg a}{a}$$

- **Eliminazione di \vee :**

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

- **Contraddizione:**

$$\frac{a \wedge \neg a}{b}$$

ovvero da una contraddizione posso trarre qualsiasi conseguenza