# 1 - Introduzione ai linguaggi di programmazione ed alla teoria dei linguaggi formali

La differenza tra un LDP e la programmazione è la possibilità di far conoscere più linguaggi di programmazione di diverso tipo al programmatore con un alta velocità, per far ciò non basta solamente apprendere molti linguaggi differenti ma comprendere le **fondamenta** dietro questi linguaggi.
E' complicato conoscere un nuovo linguaggio in modo approfondito, ma è possibile conoscere i meccanismi che ne ispirano l'**implementazione**.
Per implementazione di un linguaggio di programmazione si intende un concetto strettamente collegato al concetto di **macchina astratta**.

## Macchina astratta

Il classico calcolatore è definito come macchina **fisica**, esso ci permette di:

- **Eseguire** algoritmi che siano comprensibili all'esecutore
- **Formalizzare**, ovvero la codifica di algoritmi in un determinato linguaggio definito da una specifica sintassi
- La **sintassi** di un *determinato* linguaggio permette di utilizzare determinati *costrutti* per comporre programmi in quel *determinato* linguaggio

La macchina **astratta** invece è l'**astrazione** della macchina fisica;
Comprende ed è utilizzata per un certo linguaggio $L$, essa si rappresenta con la lettera $Ml$ composta da:

- una **memoria** per immagazzinare i dati ed il programma con le sue istruzioni
- un **interprete** che esegue le istruzioni contenute nei programmi
  - L'interprete gestisce le operazioni per l'**elaborazione** dei dati primitivi
  - L'interprete gestisce le operazioni per il trasferimento dati, gestendo il **trasferimento** dalla memoria all'interprete e viceversa, inoltre gli interpreti possono far uso delle strutture dati ausiliari
  - L'interprete gestisce le operazioni e strutture dati per la gestione della **memoria** ovvero le relative azioni di all'allocazione e de-allocazione di memoria per i programmi

**"Data una macchina astratta $Ml$, il linguaggio $L$ *compreso* dall'interprete di $Ml$ è detto linguaggio macchina di $Ml$."**
La macchina a basso livello avrà delle istruzioni meno complesse di una macchina astratta che produce operazioni ad alto livello e per questo più complesse.

## Realizzazione macchina astratta

Per utilizzare una macchina astratta (quindi operazioni di alto livello per ogni livello superiore) si dovrà utilizzare un dispositivo **fisico** (versione hardware), quindi gli algoritmi implementati della macchina astratta dovranno essere realizzati su dispositivi fisici per poter essere funzionanti ed utili.

Possibili realizzazioni, per far comprendere tutto ciò dai livelli alti a quelli più bassi, è usare livelli **intermedi** tra $Ml$ ed il dispositivo fisico, attraverso **simulazioni mediante i software** o un emulazione mediante firmware (definiamo inizialmente i firmware come microprogrammi in linguaggi di basso livello).
Per esempio nel linguaggio C si utilizzano le **librerie** per poter implementare funzioni di un livello più alto del C (quindi più complesse) per poterle farle utilizzare al C che è di più basso livello rispetto all'idea iniziale.

## Realizzazione mediante software

$Ml$ è realizzata mediante un altro linguaggio detto $L'$ che permette di simulare le funzionalità di $Ml$, quindi la macchina astratta di un certo livello che comprende un determinato livello è composta da un altra macchina di un livello più basso, definita $M'l'$, quest'ultimo definito **macchina ospite** denotata come $MO_{Lo}$.
L'implementazione di un linguaggio $L$ sulla macchina ospite avviene tramite una **traduzione** di $L$ in $Lo$.
A seconda di quest'ultima fase di traduzione tra il linguaggio definito inizialmente e il linguaggio della macchina ospite si parla di:

- Implementazione **interpretativa**
- Implementazione **compilativa**

A seconda del linguaggio di programmazione scelto, si detiene utilizzare un programma ben spiegato nella sua programmazione e privo di errori morfologici, poiché potrebbe essere complesso comprenderlo nella sua sintassi e semantica, anche se lo scopo del linguaggio di programmazione è quello di essere più comprensivo di quello naturale, data la sua struttura, ove però sia mal composto potremmo ricadere nello stesso risultato di difficoltà, privando il vantaggio di comprensione dei linguaggi di programmazione.

# Funzioni parziali

Una funzione parziale viene definita come:

$$f : A \to B$$

Una funzione parziale è una corrispondenza tra elementi dell'insieme A ed elementi dell'insieme B.
Essa può essere **non definita** per qualche elemento di A.
Dato un elemento $a \in A$, **se esiste** un elemento corrispondente in B, esso è denotato con $f(a)$.

I programmi definiscono delle funzioni parziali, come queste

```
read(x);
if x==1 then print(x) else
      while (x<>1) do skip;
```

$$f(x) = \begin{cases} 1 \text{ se } x=1 \\ \text{indefinita altrimenti} \end{cases}$$

# Definizione di interprete

In generale un programma scritto in linguaggio $L$ si può vedere come una funzione parziale, creando l'operazione

$$P^L : D \to D$$

tradotta anche in

$$P^L(\text{Input}) = \text{Output}$$

Il programma $P$ scritto in linguaggio $L$ prende determinati dati $D$ in input e li trasforma in dati $D$ in output

$$I_{\mathcal{L}}^{\mathcal{L}o} : (\mathcal{P}rog^{\mathcal{L}} \times \mathcal{D}) \to \mathcal{D} \qquad \text{tale che}$$

$$I_{\mathcal{L}}^{\mathcal{L}o}(\mathcal{P}^{\mathcal{L}}, Input) = \mathcal{P}^{\mathcal{L}}(Input)$$

Non vi è quindi una traduzione esplicita, ma solo un procedimento di decodifica, l'interprete fa corrispondere ad una istruzione di $L$, un insieme di istruzioni di $L_O$.
La decodifica non corrisponde ad una traduzione esplicita poiché il codice di $L_O$ è eseguito direttamente, senza produrlo in uscita.

- **Vantaggi**
  1. Maggiore flessibilità: permette un debugging molto più semplice
- **Svantaggi**
  1. Scarsa efficienza: la decodifica viene fatto in runtime e viene eseguita ogni volta che viene eseguita un istruzione

# Definizione di compilatore

Il compilatore corrisponde alla funzione

$$C_{L,L_O} : Prog^L \to Prog^{L_O}$$

$C_{L,L_O}(P^L) = Pc^{Lo}$ con $P^L(\text{Input}) = Pc^{Lo}\text{Input})$

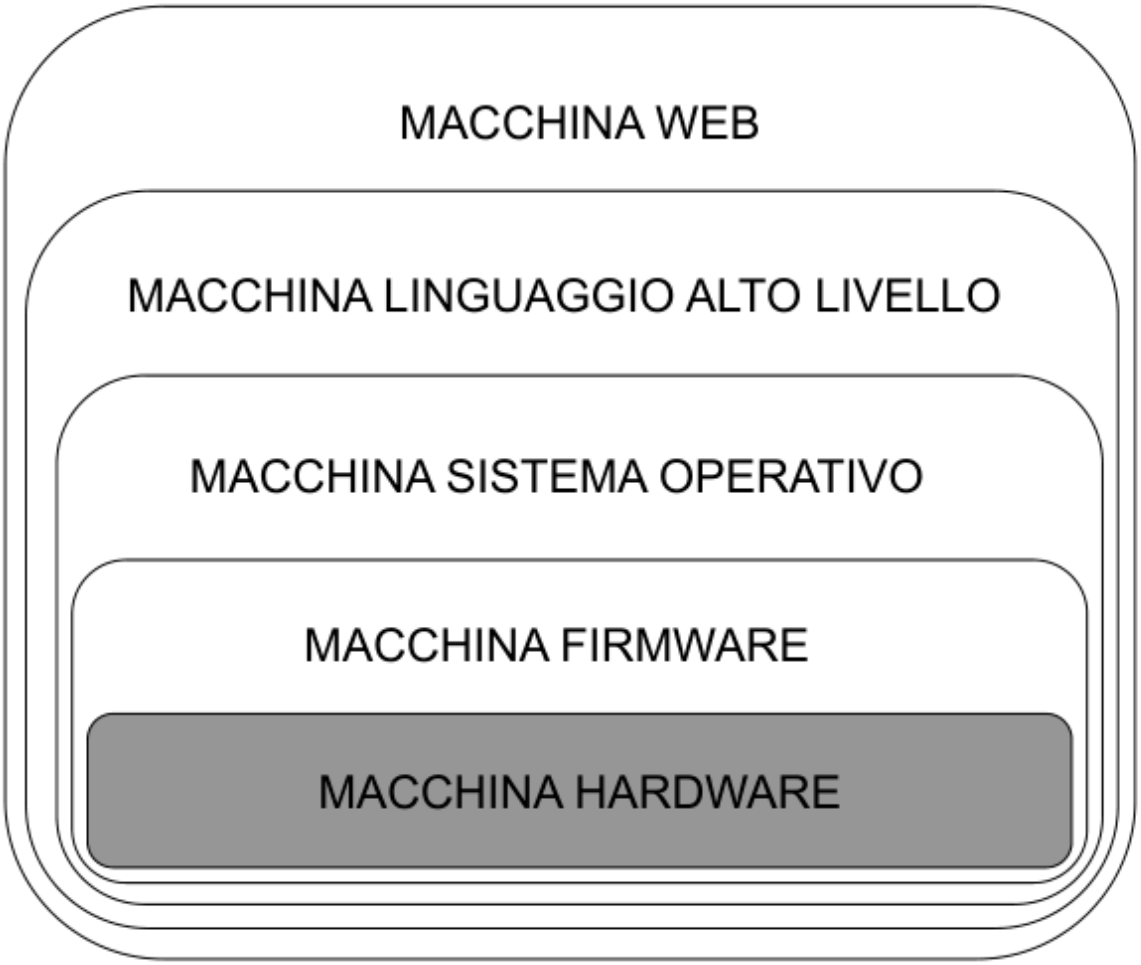
Un compilatore esegue una **traduzione esplicita** dei programmi scritti in un linguaggio $L$ in un altro linguaggio $L_o$, il risultato sarà un programma compilato $Pc^{L_O}$ scritto in $LO$, eseguito su $MoLo$ con i dati di input.

- **Vantaggi**
  1. Maggiore efficienza: la decodifica di un istruzione viene eseguita una sola volta indipendentemente da quante volte è eseguita
- **Svantaggi**
  1. Minore flessibilità: esiste una perdita di informazioni rispetto al programma sorgente (più difficile da tracciare gli errori, rendendo il debugging più difficile)

# Gerarchie di macchine astratte

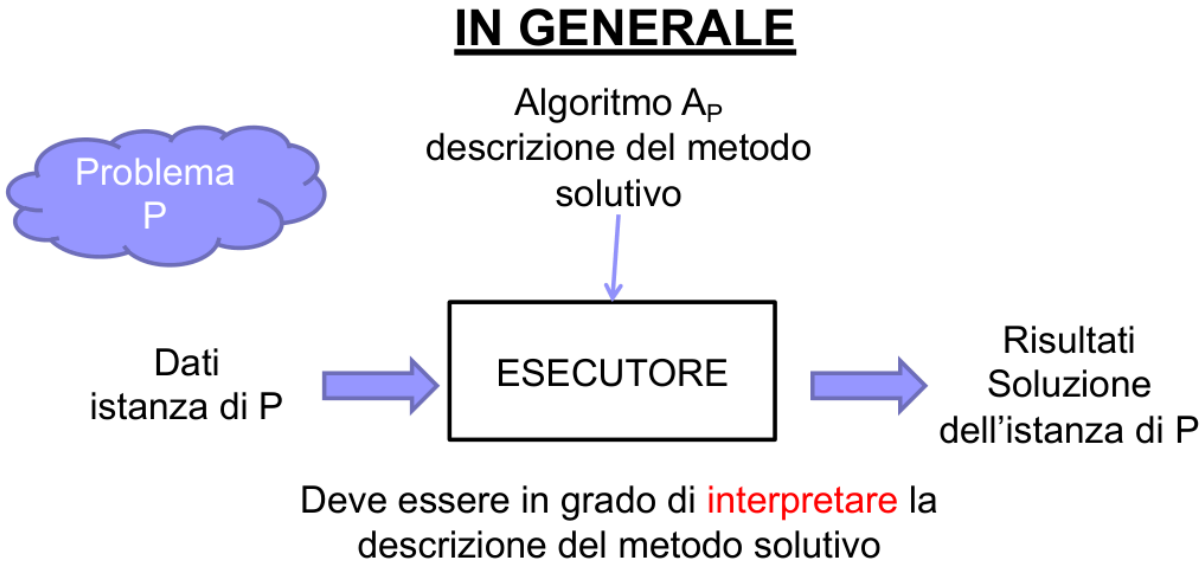
Si può pensare ad una gerarchia di macchine astratte $M_{Lo}$ , $M_{L_1}$ etc.
Ogni macchina astratta si basa su una macchina astratta più basse, e, a sua volta, è la base per una macchina astratta di più alto livello.
I livelli tra di loro sono indipendenti, in quanto un cambio alle funzionalità di una macchina

astratta non interferisce con le altre macchine astratte.




# Linguaggi di programmazione, Problemi, Interpreti

I linguaggi di programmazione non sono altro che formalismi per portare al livello di macchina fisica l'algoritmo $A_P$ per un certo problema $P$




Però **esistono** problemi a cui non c'è una soluzione, cosa che non dipende dal linguaggio.

# Funzione calcolabile parziale

Una **funzione parziale** $f : A \to B$ è una funzione definita solo per un sottoinsieme del dominio $A$. In altre parole, esistono elementi in $A$ per i quali $f(a)$ non è definita.

Una funzione parziale è detta **calcolabile** se esiste un algoritmo che, dato un input $a \in A$:

- **Termina** producendo l'output $f(a)$ quando $f(a)$ è definita.
- **Non termina** (entra in un ciclo infinito) quando $f(a)$ non è definita.

Esistono problemi per i quali **non esiste alcun algoritmo** che possa sempre fornire una soluzione in un tempo finito.

## Problema della fermata

Il problema della fermata chiede se esista un algoritmo generale che, dato un qualsiasi programma $P$ e un input $x$, possa determinare in anticipo se $P$ terminerà o andrà in loop infinito su $x$.

Si supponga di avere un programma di debugging $H$ che prende in ingresso:

- Un programma $P$ scritto in un linguaggio $L$
- Un input $x$ per il programma $P$

$H(P, x)$ dovrebbe restituire:

- `true` se $P(x)$ termina
- `false` se $P(x)$ va in loop infinito

# Indecibilità

Il problema della fermata è **indecidibile**, cioè non è possibile costruire un algoritmo che risolva questo problema per tutti i possibili programmi e input, rendendolo una funzione non calcolabile
Per ogni funzione calcolabile esiste una macchina di Turing che la formalizzi.

# Macchine di Turing

La **macchina di Turing** è un modello matematico, concepito per formalizzare il concetto di algoritmo e per esplorare i limiti del calcolo meccanico, infatti è stato il primo formalismo (o linguaggio) a dimostrare l'indecibilità del problema della fermata

La macchina di Turing è formata da diverse componenti:

- **Nastro infinito**: Un nastro suddiviso in celle, estendibile indefinitamente in entrambe le direzioni, che funge da memoria della macchina.
  Ogni cella può contenere un simbolo appartenente a un alfabeto finito.
- **Testina di lettura/scrittura**: Un dispositivo che si muove lungo il nastro, leggendo e scrivendo simboli nelle celle e spostandosi a sinistra o a destra dopo ogni operazione.

- **Stato interno**: La macchina possiede un insieme finito di stati interni, in ogni momento si trova in uno di questi stati che influenzano il comportamento della macchina durante l'elaborazione.
- **Funzione di transizione**: Un insieme di regole che determinano, in base allo stato attuale e al simbolo letto sul nastro, quale simbolo scrivere, in quale direzione muovere la testina e in quale nuovo stato entrare.

L'unità di controllo della macchina esegue un programma P sui dati memorizzati sul nastro, come

```
<simbolo_letto,
stato_corrente,
simboli_da_scrivere,
sinistra/destra
nuovo_stato>
```

# MdT: modello matematico

Una MdT è definita da una quintupla:

$$M = (X, Q, fm, fd, \delta)$$

Dove:

- **X**: insieme finito di simboli (alfabeto della MdT), incluso il simbolo blank (cella vuota).
- **Q**: insieme finito di stati, tra cui lo stato speciale *HALT* che indica la terminazione.
- **fm**: funzione di macchina che determina il simbolo da scrivere sul nastro. ( $fm : Q \times X \to X$)
- **fd**: funzione di direzione che determina il movimento della testina (S=sinistra, D=destra, F=ferma) ($fd : Q \times X \to Q$)
- $\delta$: funzione di transizione di stato che definisce il prossimo stato della computazione. ( $\delta : Q \times X \to X$)

# Scrivere algoritmi per MdT: il controllo

Quando progettiamo un algoritmo per una MdT, dobbiamo trasformare una **configurazione iniziale del nastro** in una **configurazione finale** che rappresenti la soluzione del problema. Per farlo, dobbiamo definire tre funzioni fondamentali:

1. $fm$ (funzione di macchina): stabilisce quale simbolo scrivere nel nastro.
2. $fd$ (funzione di direzione): stabilisce se la testina si sposta a sinistra (S), destra (D o resta ferma (F).
3. $\delta$ (funzione di transizione di stato): stabilisce lo stato successivo della macchina.

L'algoritmo per la MdT è una sequenza di **quintuple** del tipo:

$$< x_i \in X, q_j \in Q, x_{ij} \in X, fd, q_{ij} \in Q >$$

Dove:

- $x_i$ = simbolo letto sulla cella corrente del nastro.
- $q_j$ = stato attuale della macchina.
- $x_{ij}$ = simbolo da scrivere nella cella corrente.
- $fd$= direzione del movimento della testina (**S, D, F**).
- $q_{ij}$ = nuovo stato in cui la MdT continuerà la computazione.

In corrispondenza di un simbolo letto $x_i$ e dello stato $q_j$ in cui si trova l'unità di controllo, si determinano:

- Il simbolo $x_{ij} = fm(x_i, q_j)$) da scrivere nella cella corrente
- Lo spostamento della testina $fd(x_i, qj)$
- Il nuovo stato $q_{ij} = \delta(x_i, q_j)$ in cui la MdT continuerà la computazione

# Tesi di Church Turing

Siccome ogni funzione calcolabile può essere formalizzata con una MdT, non esiste alcun formalismo capace di risolvere una classe i funzioni calcolabili più ampia di quella che si può risolvere con MdT.
Quindi, la classe delle funzioni calcolabili coincide con la classe delle funzioni calcolabili da MdT, questa affermazione è chiamata Tesi di Church-Turing.

# Un parallelo tra MdT e moderni processori

### Macchina di Turing

- Nastro infinito: la MdT utilizza un nastro potenzialmente infinito come memoria, dove ogni cella può contenere un simbolo di un alfabeto finito.
- Testina di lettura/scrittura: può leggere e scrivere su una cella del nastro e spostarsi a sinistra o a destra di una cella alla volta.
- Stati finiti: la MdT ha un insieme finito di stati e transizioni che determinano il comportamento della macchina.
- Programma fisso: il programma è cablato nella macchina sotto forma di una tabella di transizioni. Ogni MdT è progettata per risolvere un problema specifico.

### Moderni processori

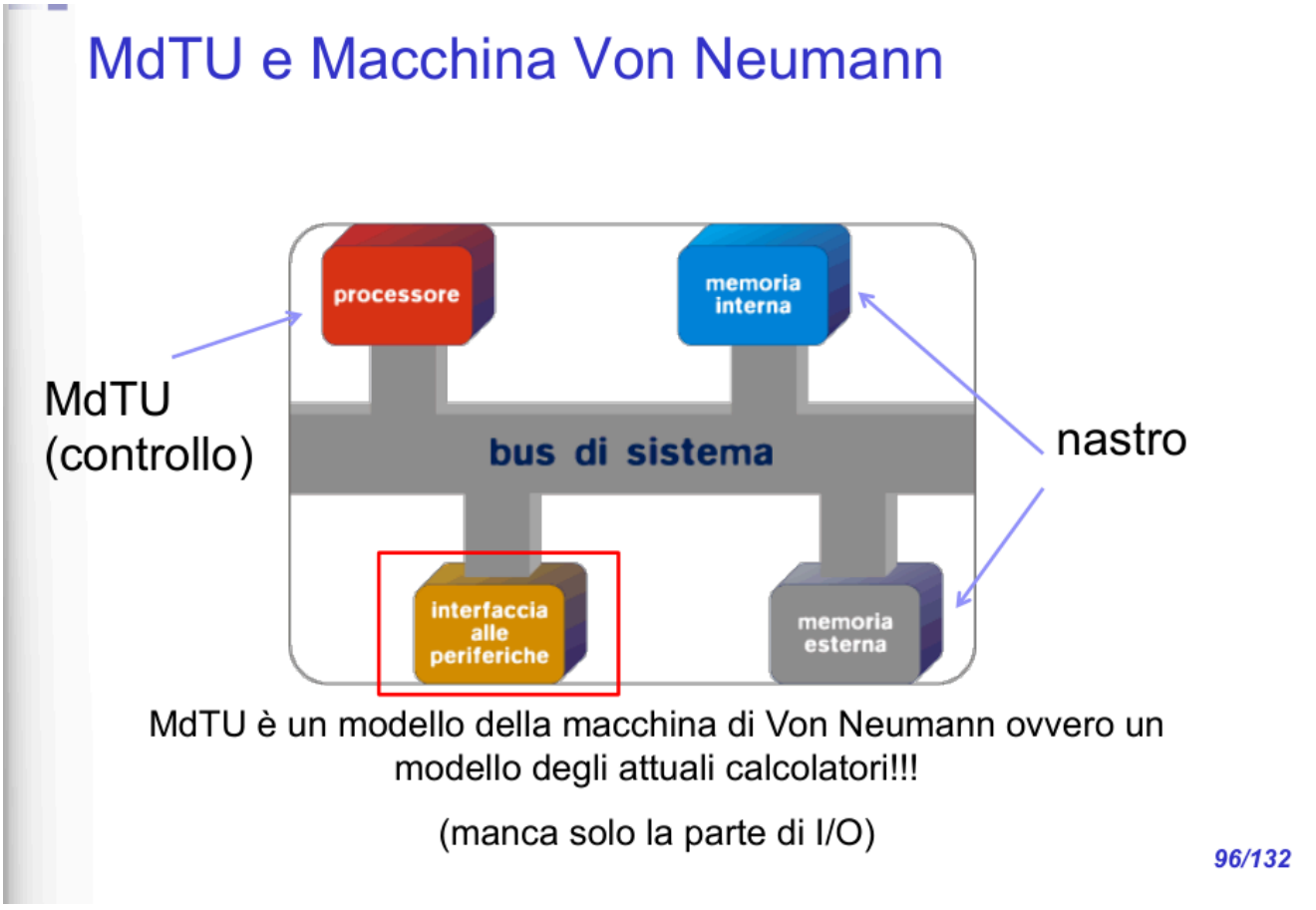
- Memoria RAM/ROM: i processori moderni utilizzano memorie RAM e ROM per leggere e scrivere dati, con capacità limitate ma sufficienti per la maggior parte delle applicazioni pratiche.
- Registri: utilizzano registri per operazioni veloci e gestione degli indirizzi.

- Istruzioni complesse: supportano un set di istruzioni più complesso e vario rispetto alle semplici operazioni delle MdT.
- Programma variabile: i processori sono progettati per eseguire programmi diversi e possono caricare diversi software dalla memoria, rendendoli molto più versatili.

## MdT Universale (MdTU)

La Macchina di Turing universale, a differenza di una macchina di Turing standard, che ha un programma fisso, legge sia i dati che il programma direttamente dal nastro. Questo permette alla MdTU di eseguire qualsiasi algoritmo che può essere rappresentato come una macchina di Turing.
Questa versatilità la rende programmabile, nel senso che può essere configurata per simulare il comportamento di qualsiasi altra macchina di Turing.
Agisce come un interprete, leggendo le istruzioni (le "5-ple" che definiscono l'algoritmo solutivo) dal nastro, decodificandole ed eseguendole.
Questo processo è simile al ciclo di fetch-decode-execute dei moderni processori.

La macchina di Von Neumann non è altro che una MdTU.




# Introduzione alla teoria dei linguaggi formali

La teoria dei linguaggi formali serve per possedere una buona comprensione dell'operazione di compilazione, questa si fonda pesantemente sulla teoria dei linguaggi formali.
Per poter definire un linguaggio servono diversi livelli di descrizione:

- **Grammatica**: definire quali frase sono corrette
- **Semantica**: cosa significa una frase corretta
- **Pragmatica**: come usare una frase corretta e sensata
- **Implementazione (per i linguaggi di programmazione)**: come eseguire una frase corretta in modo da rispettarne il significato

## Concetto intuitivo di grammatica

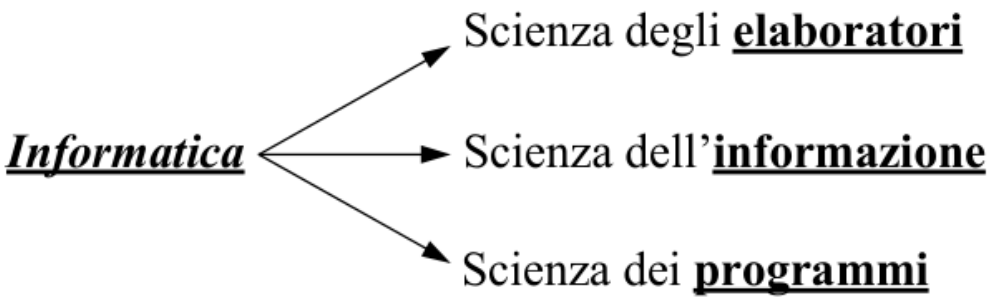
La grammatica è una forma con una serie di regole che permettono di formulare frasi corrette composto da:

- **Alfabeto del linguaggio**: simboli con cui costruire le parole del linguaggio
- **Lessico**: parole previste dal linguaggio (con delle regole per poter poi costruire tutto)
- **Sintassi**: determina quale sequenze di parole costituiscono frasi legali corrette

Le grammatiche quindi sono uno strumento utile per descrivere una sintassi di un linguaggio

# Informatica teorica

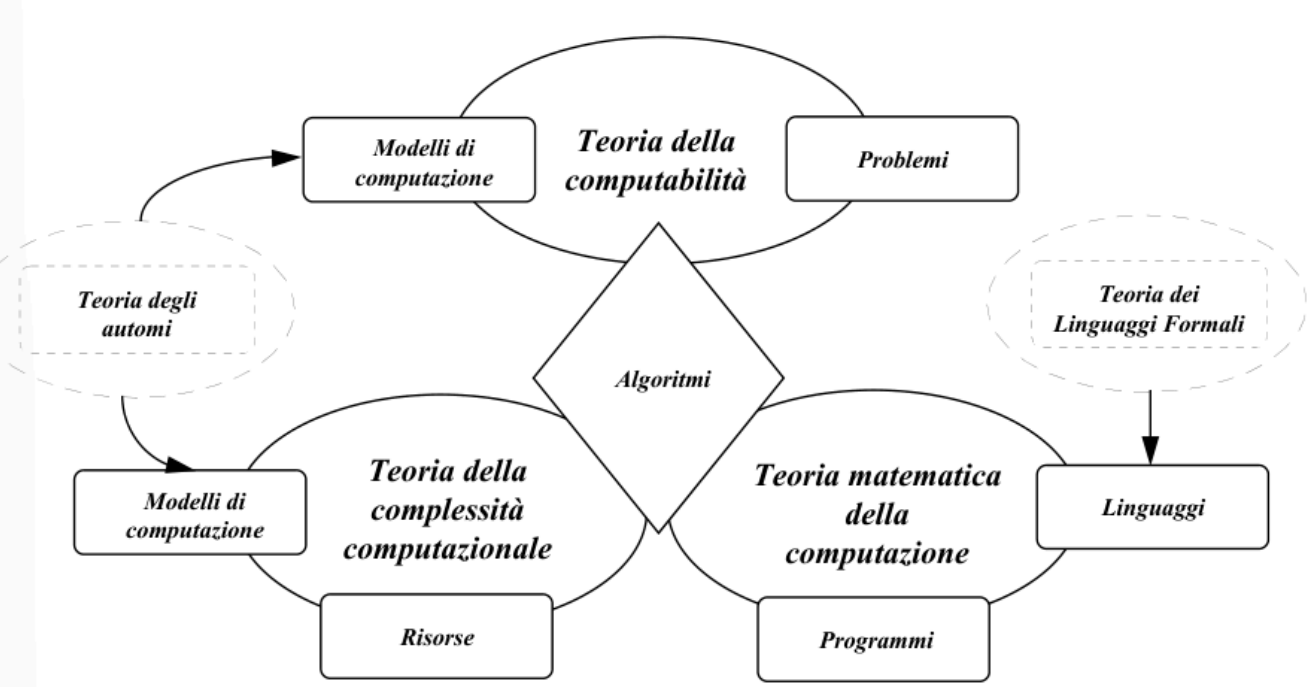
L'informatica teorica è la **scienza degli algoritmi** in tutti i loro aspetti, poiché è il concetto di algoritmo che è in qualche modo comune a tutti i settori dell'informatica




**elaboratori** ≡ macchine che eseguono gli algoritmi
**informazione** ≡ materia su cui lavorano gli algoritmi
**programmi** ≡ algoritmi descritti in un particolare linguaggio

## Aree di ricerca dell'informatica teorica

Le aree più importanti dell'informatica si basano su diverse aree di ricerca e diverse teorie




Le tre aree studiano poi a loro volta diversi argomenti:

1. **Teoria di computabilità**: si occupa di definire diverse classi di dispositivi di calcolo e, per ogni classe, dello studio di proprietà e delle classi di problemi che non sono in grado di risolvere
   Nell'informatica teorica esiste un forte legame tra la Teoria degli automi e la Teoria dei linguaggi, nata in ambito linguistico per caratterizzare i linguaggi naturali
2. Teoria matematica della computazione: definisce il rapporto tra gli algoritmi e i linguaggi di programmazione, assieme alla proprietà dei programmi (correttezza, terminazione, etc.)
3. Teoria della complessità computazionale: è il rapporto tra gli algoritmi e le risorse necessarie per essere eseguiti

# Studio dei linguaggi

I linguaggi di programmazione, una volta imparati le diverse regole, diventano familiari, qualunque esso sia.

## Sintassi e semantica

Nei nostri linguaggi naturali si può comunicare con frasi mal strutturate e incomplete, ma un computer richiede precisione, per questo i programmi devono aderire rigorosamente a regole stringenti e anche delle differenze minori vengono rigettate come errate.
Per un linguaggio macchina è essenziale che **la sintassi sia corretta per comunicare una qualsivoglia semantica**.

Lo studio della semantica non è nient'altro che lo studio della grammatica, quindi della struttura delle frasi, in un linguaggio naturale si usano diverse regole scritte in BNF(Backus Naur Form), metalinguaggio utilizzato per poter definire queste regole in modo semplice e derivabili

***The man hits the dog***

può essere analizzata sintatticamente, cioè risolta nelle parti grammaticali componenti, come segue:


L'insieme di regole è uguale a questo:

< frase semplice > :: =< parte nominale > < parte verbale > < parte nominale >

< parte nominale > :: =< articolo > < nome >

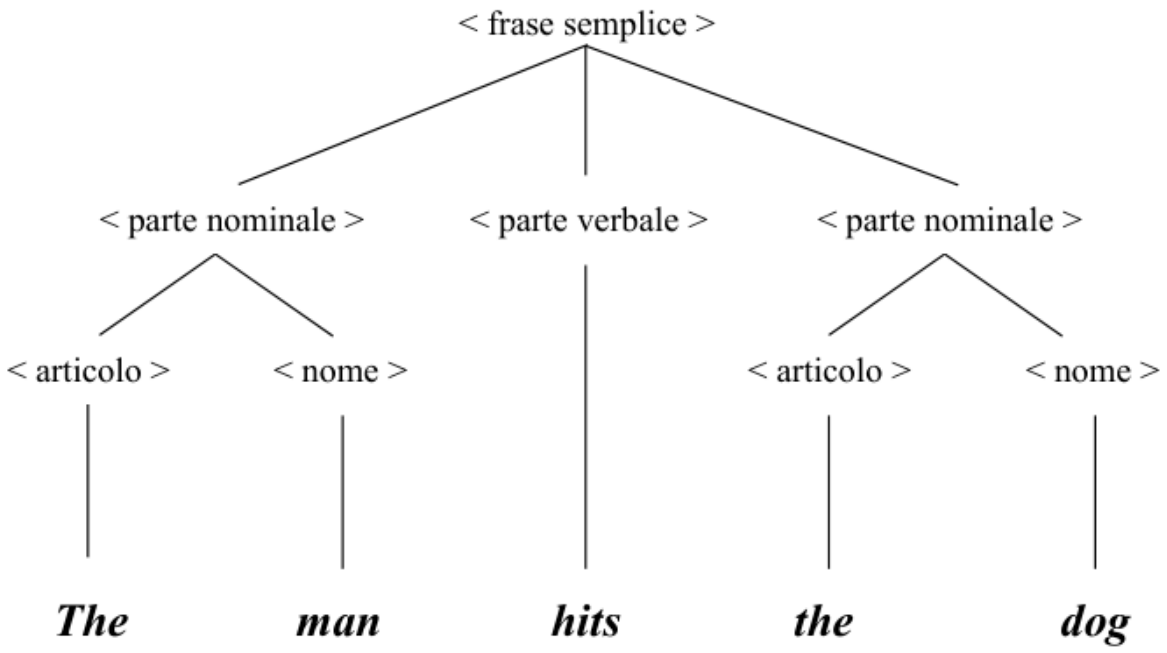
< nome > :: = ***car*** | ***man*** | ***dog***

< articolo > :: = ***The*** | ***a***

< parte verbale > :: = ***hits*** | ***eats***

Si potrebbe descrivere questa frase con un albero di derivazione

**Albero di derivazione**




La definizione che da la <frase semplice> permette 72 diverse frasi derivabili, che sebbene alcune siano sintatticamente corrette, non hanno interpretazione errata

Questo esempio illustra un importantissimo concetto nella computazione, cioè nel processo di compilazione avviene come fase cruciale **l'analisi sintattica** come passo essenziale per la sua **interpretazione semantica**

# Teoria dei linguaggi formali

Negli anni 50 **Chomsky** cerca di descrivere la sintassi del linguaggio naturale secondo semplici regoli di **riscrittura e trasformazione**.
Chomsky considera alcune **restrizioni** sulle regole sintattiche e classifica dei linguaggi in base alle restrizioni imposte alle regole che generano tali linguaggi, una classe importante di regole derivata che genera linguaggi formali si chiama "**grammatiche libere da contesto** (Context-free Grammars)", tale classe dimostra che è uno strumento adeguato a descrivere la sintassi di base di molti linguaggi di programmazione

Un esempio di linguaggio C.F è il linguaggio delle parentesi ben formate che comprende tutte le stringhe di parentesi aperte e chiuse bilanciate correttamente.

## Linguaggio delle parentesi ben formate

Questo linguaggio è infinito e permette di contrassegnare il **raggio di azione** nelle espressioni matematiche e, di conseguenza, nei linguaggi di programmazione

**Esempio:**

( ) è ben formata;
( ( ) ( ) ) è ben formata;
( ( ) ( ) non è ben formata.

Questo linguaggio è definito da un insieme di regole:

1. La stringa $()$ è ben formata
2. Se la stringa di simboli $A$ è ben formata, allora lo è anche $(A)$
3. Se le stringhe $A$ e $B$ sono ben formate, allora lo è anche la stringa $AB$ (concatenate)

In corrispondenza di questa definizione induttiva, possiamo considerare un sistema di riscrittura che genera esattamente l'insieme di stringhe lecite di parentesi ben formate come

$$1. S \to ()$$

$$2. S \to (S)$$

$$3. S \to SS$$

Queste regole di riscrittura sono dette anche di **produzioni** o **regole di produzione**, in questo caso stabiliscono che "data una stringa, si può formare una nuova stringa sostituendo una $S$ o con una $()$ O con $(S)$ o con $SS$

- Esempio: Generazione di (( ))(( )( ))
  - Primo modo:

$$S \underset{(3)}{\Rightarrow} SS \underset{(2)}{\Rightarrow} (S)S \underset{(1)}{\Rightarrow} (( ))S \underset{(2)}{\Rightarrow} (( ))(S) \underset{(3)}{\Rightarrow} (( ))(SS) \underset{(1)}{\Rightarrow} (( ))(( )S) \underset{(1)}{\Rightarrow} (( ))(( )( ))$$

  - La corrispondente sequenza di applicazione delle produzioni è:

$$(3) \to (2) \to (1) \to (2) \to (3) \to (1) \to (1)$$

La sequenza di applicazione di queste regole di produzione in successione viene definito come **derivazione** (definito come $\Rightarrow$)

Nell'esempio viene usata la notazione $\alpha \Rightarrow \beta$ (da alpha si produce direttamente beta) per effetto dell'applicazione della regola di riscrittura $n$