

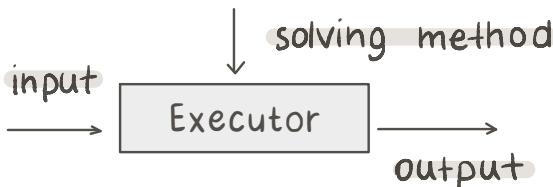
## - Esistono problemi irrisolvibili?

Si esistono problemi irrisolvibili. Esistono anche problemi che possono essere risolti solo da determinati linguaggi.

Algoritmo = sequenza finita di passi che risolve una classe di problemi in un certo tempo finito.

Un programma può anche non terminare mai (Sistema Operativo).

## Esecuzione



## Esistono vari esecutori:

- Approccio astratto (macchine a stati finiti)
- Approccio funzionale (funzioni matematiche)
- Approccio della riscrittura.

NB: Se un problema non può essere risolto da una macchina super potente allora non è risolvibile.

## MACCHINE DI BASE

I → input

O → output

Mfn → I → O machine function

## Esempio di un bool:

I → {{0,1} × {0,1}}

O → {0,1}

Mfn → AND logico

Limiti: - Dobbiamo definire tutte le possibili configurazioni dell'input ;  
- Non si può salvare lo stato della macchina

Non c'è memoria , non posso fare il riconoscimento di una stringa oppure fare la somma di un a sequenza di numeri. O

## MACCHINE A STATI FINITI

I → input

O → output

S → stati

Mfn → I × S → O machine function

Sfn → I × S → S status function

Limiti: - Hanno una memoria finita ed il risultato dipende dall'input e dallo stato

dobbiamo a priori la dimensione dell'input

## MACCHINE DI TURING

I → input  
O → output  
S → stati  
Mfn → I × S → O machine function  
Sfn → I × S → S status function  
Dfn → I × S → D,D direzione testina (left,right,none)

Esempio palindrome :

A={0,1,^,:,:) }  
S={HALT,s0,s1,s2,s3,s4,s5}

La macchina di Turing è la macchina più potente che conosciamo, se infatti un problema non è risolvibile da questa macchina viene considerato irrisolvibile.

### Problemi irrisolvibili ?

Se la macchina a cui diamo in pasto il problema è programmata correttamente e se il problema è risolvibile, la macchina ritorna un risultato corretto.

Se invece non riesce a ritornare un risultato corretto la macchina non si ferma, entriamo in un loop infinito.

### Ma cosa prendono in input le macchine di Turing? Problemi?

- Le macchine di turning computano funzioni NON problemi, dobbiamo quindi trovare il modo di esprimere i problemi in termini di funzioni.

└

X → gruppo di input

Y → gruppo di output

Funzione caratteristica  $f(p)$  che lega ad ogni input  $x \in X$  un output  $y \in Y$

$$f(p): X \rightarrow Y$$

- se  $f(p)$  è computabile dalla macchina allora il problema è RISOLVIBILE
- se  $f(p)$  non è computabile dalla macchina allora il problema è IRRISOLVIBILE

### La domanda sorge quindi spontanea, quando una funzione è COMPUTABILE?

- Una funzione  $f(p)$  si dice computabile se esiste una macchina di Turing che è in grado di:

– data la rappresentazione dell'input dopo un numero finito di passi produrre un output corretto.

└

$f$  non è computabile se la macchina di Turing non può produrre un output corretto in un numero finito di passi (loop infinito).

### Esempio di funzione definibile ma non computabile

#### Il problema dell' HALT

Questo problema può essere definito ma NON computato.

DATI:

- $M \rightarrow$  insieme di tutte le macchine di Turing
  - |
    - $m \in M \rightarrow$  una generica macchina di Turing
- $X \rightarrow$  un generico insieme di input
  - |
    - $x \in X \rightarrow$  un generico input
- $*$   $\rightarrow$  un risultato indefinito, quando non riesce a darci una risposta

$$f_H(m, x) = \begin{cases} 1, & \text{se } m \text{ con input } x \text{ si ferma (risponde)} \\ 0, & \text{se } m \text{ con input } x \text{ non si ferma (non risponde)} \end{cases}$$

Processo di Gödel: ad ogni macchina di Turing può essere associato un numero naturale

Consideriamo un'altra funzione  $g(\text{HALT})$  a cui passiamo il numero naturale associato come descrizione ( quindi passiamo se stessa come input )

$$g_{\text{HALT}}(n_g) = \begin{cases} 1, & \text{se } f_H(n_g, n_g) = 0 \\ *, & \text{se } f_H(n_g, n_g) = 1 \end{cases}$$

Questo è assurdo perché stiamo dicendo che:

- se  $f_H$  si ferma allora  $g(\text{HALT})$  ritorna che non si ferma;
- se  $f_H$  non si ferma allora  $g(\text{HALT})$  ritorna che si ferma;

## Decidibilità di un gruppo

- Dato un elemento dobbiamo capire se tale appartiene al gruppo oppure no.

Gruppo numerabile  $\rightarrow$  gruppo i cui elementi possono essere numerati, le cui funzioni potrebbero essere non computabili.

Gruppo numerabile ricorsivamente  $\rightarrow$  un gruppo numerabile le cui funzioni sono sempre computabili, possiamo cioè generare un elemento ma non significa che quell'elemento appartiene all'insieme

|

- esempio numero primo :

Chiediamo alla MT di generare tutti i numeri primi, adesso se in input diamo un insieme relativamente piccolo 100 ad esempio allora la macchina calcolerà tutti i numeri primi fino a 100, ma se aumentiamo la dimensione dell'input e la facciamo tendere ad infinito allora non sappiamo le macchina non ci da come output il numero che vogliamo perché non appartiene all'insieme o perché ancora non è arrivato a calcolarlo.

**Gruppo decidibile** → un gruppo i cui elementi possono essere numerati ed esiste un algoritmo che finisce dopo un numero finito di passi e sa decidere se un elemento appartiene al gruppo.



teoremi dei gruppi decidibili:

- 1° → un gruppo decidibile è anche semi-decidibile
- 2° → un gruppo G è decidibile solo se N-G è semidecidibile

Siamo interessati ai gruppi decidibili perché un linguaggio è un gruppo di parole ed i linguaggi di programmazione sono gruppi di parole (alfabeti) che seguono una rigida grammatica

**Grammatica** → notazione formale attraverso la quale specifichiamo la sintassi di un linguaggio.



VT : gruppo finito di simboli terminali (caratteri o stringhe di alfabeto A)

VN : gruppo finito di simboli non terminali (categorie sintattiche)

L'unione di VT e VN forma un vocabolario della grammatica

Una forma sentenziale è una stringa che contiene simboli terminali e non.

Una parola è una forma sentenziale composta da soli simboli terminali.

Esempio :

SUBJECT VERB PREDICATE



Non-terminali

bob washes the car



Terminali

## GRAMMATICA FORMALE

Una **Grammatica** è una **notazione formale** con cui esprimere **in modo rigoroso** la **sintassi** di un linguaggio.

Una grammatica è una quadrupla  $\langle VT, VN, P, S \rangle$  dove:

- **VT** è un **insieme finito di simboli terminali**
- **VN** è un **insieme finito di simboli non terminali**
- **P** è un **insieme finito di produzioni, ossia di regole di riscrittura**  $\alpha \rightarrow \beta$  dove  $\alpha$  e  $\beta$  sono stringhe:  $\alpha \in V^*$ ,  $\beta \in V^*$ 
  - ogni regola esprime una trasformazione lecita che permette di scrivere, nel contesto di una frase data, una stringa  $\beta$  al posto di un'altra stringa  $\alpha$ .
- **S** è un particolare simbolo non-terminali detto **simbolo iniziale o scopo della grammatica**

Esempio : EXP=EXP OP EXP  
 EXP=CONST  
 CONST = (1 2 3 4 5 6 7 8 9 10)  
 OP = (+ - \* /)

## DERIVAZIONE

Siano  $\alpha, \beta$  due stringhe  $\in (VN \cup VT)^*$ ,  $\alpha \neq \epsilon$

Si dice che  $\beta$  deriva direttamente da  $\alpha$  ( $\alpha \rightarrow \beta$ ) se

- le stringhe  $\alpha, \beta$  si possono decomporre in  
 $\alpha = \eta A \delta$        $\beta = \eta \gamma \delta$   
 ed esiste la produzione  $A \rightarrow \gamma$ .

$$\eta A \delta$$

Si dice che  $\beta$  deriva da  $\alpha$  (anche non direttamente) se

- esiste una sequenza di  $N$  derivazioni dirette che da  $\alpha$  possono infine produrre  $\beta$

$$\alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \dots \rightarrow \alpha_N = \beta$$

Voglio esprimere 34+18

S=EXP  
 EXP=EXP OP EXP  
 EXP=CONST OP CONST  
 3CONST OP CONST  
 34CONST OP CONST  
 34 OP CONST  
 34 + CONST  
 34 + 1CONST  
 34 + 18 CONST  
 34+18

## GRAMMATICA DI TIPO 0

### CLASSIFICAZIONE DI CHOMSKY TIPO 0

Le grammatiche sono classificate in 4 tipi  
in base alla struttura delle produzioni

- **Tipo 0:**  
*nessuna restrizione sulle produzioni*

In particolare, le regole possono specificare riscritture che accorcianno la forma di frase corrente.

Esempio (grammatica di tipo 0)

$$\begin{array}{llll} S \rightarrow aSBC & CB \rightarrow BC & SB \rightarrow bF & FB \rightarrow bF \\ FC \rightarrow cG & GC \rightarrow cG & G \rightarrow \epsilon & \end{array}$$

Possibile derivazione  $S \rightarrow aSBC \rightarrow abFC \rightarrow abcG \rightarrow abc$   
 $lung=4 \quad lung=3$

## GRAMMATICA DI TIPO 1

### CLASSIFICAZIONE DI CHOMSKY TIPO 1

Le grammatiche sono classificate in 4 tipi in base alla struttura delle produzioni

- **Tipo 1 (dipendenti dal contesto):**  
 produzioni vincolate alla forma:

$$\beta A \delta \rightarrow \beta \alpha \delta$$

con  $\beta, \delta \in V^*, \alpha \in V^+, A \in VN$

$$\alpha \neq \epsilon$$

Quindi,  $A$  può essere sostituita da  $\alpha$  solo nel contesto  $\beta A \delta$ .  
 Le riscritture non accorcianno ma la forma di frase corrente.

Esempio (grammatica di tipo 1)

$$\begin{array}{llll} S \rightarrow aBC | aSBC & & & \\ CB \rightarrow DB & DB \rightarrow DC & DC \rightarrow BC & \\ aB \rightarrow ab & bB \rightarrow bb & bC \rightarrow bc & cC \rightarrow cc \end{array}$$

Infatti, secondo la definizione  $\beta A \delta \rightarrow \beta \alpha \delta$  si può trasformare un metasimbolo per volta ( $A$ ), lasciando intatto ciò che gli sta intorno

Osserva: la lunghezza del lato destro delle produzioni non è mai inferiore a quella del lato sinistro.

### ESEMPIO

$$\begin{array}{llll} S \rightarrow aBC | aSBC & \beta = \epsilon & \delta = \epsilon & \\ CB \rightarrow DB & \beta = \epsilon & \delta = B & \\ DB \rightarrow DC & \beta = D & \delta = \epsilon & \\ DC \rightarrow BC & \beta = \epsilon & \delta = C & \\ aB \rightarrow ab & \beta = a & \delta = \epsilon & \\ bB \rightarrow bb & \beta = b & \delta = \epsilon & \\ bC \rightarrow bc & \beta = b & \delta = \epsilon & \\ cC \rightarrow cc & \beta = c & \delta = \epsilon & \end{array}$$

## GRAMMATICA DI TIPO 2

### CLASSIFICAZIONE DI CHOMSKY TIPO 2

Le grammatiche sono classificate in 4 tipi in base alla struttura delle produzioni

- **Tipo 2: context free (*indipendenti dal contesto*):**

produzioni vincolate alla forma:

$$A \rightarrow \alpha$$

con  $\alpha \in (VT \cup VN)^*$ ,  $A \in VN$

**Qui A può sempre essere sostituita da  $\alpha$ , indipendentemente dal contesto.**

Se  $\alpha$  ha la forma  $u$  oppure  $u B v$ , con  $u, v \in VT^*$  e  $B \in VN$ , la grammatica si dice *lineare*.

## GRAMMATICA DI TIPO 3

### CLASSIFICAZIONE DI CHOMSKY TIPO 3

Le grammatiche sono classificate in 4 tipi in base alla struttura delle produzioni

- **Tipo 3 (grammatiche regolari):** produzioni vincolate alle *forme lineari*:

lineare a destra

$$A \rightarrow \sigma$$

$$A \rightarrow \sigma B$$

con  $A, B \in VN$  e  $\sigma \in VT^*$

lineare a sinistra

$$A \rightarrow \sigma$$

$$A \rightarrow B \sigma$$

Si intende che le produzioni di una data grammatica devono essere **TUTTE o lineari a destra, o lineari a sinistra – non mischiate**.

Si noti che  $\sigma$  può essere  $\epsilon$ .

### GRAMMATICHE REGOLARI CASO PARTICOLARE

Per grammatiche regolari è **sempre possibile e spesso conveniente trasformare la grammatica in forma strettamente lineare**

- non più  $\sigma \in VT^*$  ( $\sigma$  è una stringa di caratteri)

lineare a destra      lineare a sinistra

$$A \rightarrow \sigma$$

$$A \rightarrow \sigma B$$

$$A \rightarrow \sigma$$

$$A \rightarrow B \sigma$$

- bensì  $a \in VT$  (a è un singolo carattere)

lineare a destra      lineare a sinistra

$$X \rightarrow a$$

$$X \rightarrow a Y$$

$$X \rightarrow a$$

$$X \rightarrow Y a$$

## GRAMMATICHE LINEARI: ESEMPI

$$VT = \{ a, +, - \}, VN = \{ S \}$$

- **Grammatica G1 (lineare a sinistra:  $A \rightarrow B y$ , con  $y \in VT^*$ )**

$$S \rightarrow a \quad S \rightarrow S + a \quad S \rightarrow S - a$$

- **Grammatica G2 (lineare a destra:  $A \rightarrow x B$ , con  $x \in VT^*$ )**

$$S \rightarrow a \quad S \rightarrow a + S \quad S \rightarrow a - S$$

- **Grammatica G3 (G2 resa strettamente lineare a destra)**

$$S \rightarrow a \quad S \rightarrow a A \quad A \rightarrow + S \quad A \rightarrow - S$$

- **Grammatica G4 (lineare a destra e anche a sinistra)**

$$S \rightarrow ciao$$

- **Grammatica G5 (G4 resa strettamente lineare a destra)**

$$S \rightarrow c T \quad T \rightarrow i U \quad U \rightarrow a V \quad V \rightarrow o$$

## Paradigma Imperativo

- I linguaggi di programmazione imperativi sono scritti come una guida passo passo per il computer. Descrivono esplicitamente quali passaggi devono essere eseguiti ed in quale ordine per raggiungere la soluzione desiderata.
- I linguaggi di programmazione imperativa sono caratterizzati dal loro carattere istruttivo e pertanto di solito richiedono molte più righe di codice per ciò che può essere descritto come poche istruzioni nello stile dichiarativo.

Riassumendo la programmazione imperativa si concentra più sul "COME" che sul "COSA".

## Paradigma Funzionale

### Funzioni nei linguaggi imperativi

- Una funzione è solo un costrutto per encapsulare codice
- NON può essere assegnata ad una variabile ( i puntatori a funzioni in C puntano al codice della funzione non all'entità funzione vera e propria)
- NON può essere passata come argomento di una funzione
- Una funzione non può essere creata e ritornata (solo i puntatori a funzioni esistenti possono essere ritornati)
- Sono entità definite staticamente

### Funzioni nei linguaggi funzionali

- Una funzione è un tipo di dato che può essere maneggiato come gli altri tipi di dato (int, char ecc..)
- Può essere assegnata ad una variabile di tipo funzione
- Può essere ritornata da un'altra funzione(una funzione può essere creata all'interno di altre funzioni)
- Può essere creata e definita on the fly

## Classi di una funzione

Innestare funzioni una dentro l'altra ( passando una funzione come argomento di un'altra) porta a creare una gerarchia di funzioni e a definire quindi funzioni di ordine superiore.

- La funzione a cui è passato un a funzione come argomento sarà una funzione di secondo grado, una funzione di terzo grado sarà una funzione che ha come argomento una funzione e di secondo grado ecc..

## Esempi di funzione in JavaScript

- Le funzioni sono entità di tipo <funzione>

```
var f1 = function (z) { return z*z; }
```

In questo caso la funzione definita non ha nome.

- Una funzione può ricevere un'altra funzione come argomento

```
var c = function (f, x) { return f(x); }
```

**f** parametro di tipo funzione

- Una funzione può ritornare il risultato di una sua funzione innestata

```
function createPlus10(x) {return  
function(r){return r+10}}
```

- Una funzione può essere creata testo

```
square = new Function("x", "return x*x")
```

## Chiusure

Una chiusura è una funzione di prima classe definita all'interno di uno o più blocchi (sono quindi utilizzabili solo in linguaggi che ne fanno riferimento solo come entità di prima classe)

Hanno origine nei linguaggi funzionali (Lisp, Scheme..) ma recentemente sono disponibili nei linguaggi OO (Object Oriented)

- Il codice di una chiusura può far riferimento a variabili locali definite nei blocchi che la circondano
- Le variabili usate da una chiusura non sono deallocate fino a quando la chiusura è accessibile (i dati sono allocati nell'heap invece che sullo stack)

## Esempi di chiusure in JavaScript

```
function ff(f, x) { // creates a closure  
    return function(r){ return f(x)+r; }  
}
```

- Ad ogni chiamata di ff viene creata una nuova funzione (senza nome) che ha bisogno di:
  - f, x, r per eseguire.
- Si verifica quindi la chiusura, che tiene in vita le variabili anche dopo che ff è stata eseguita.

```
var f1 = ff( Math.sin, 0.8)  
var f2 = ff( function(q){return q*q}, 0.8)
```

- f1 ed f2 contengono due funzioni differenti in quanto la prima calcolerà il seno di  $0.8 + r$  e la seconda il quadrato di  $0.8 + r$ .

▶ Now we can call f1 and f2

```
var r1 = f1(3) // 3.717356091 [sin(0.8) + 3]
```

▶ Computes the sin of the 0.8 plus 3

```
var r2 = f2(0.36) // 1.0000000 [0.64 + 0.36]
```

▶ Computes the square of 0.8 plus 0.36

- Questo esempio mette in evidenza il funzionamento della chiusura in quanto x ed r sono sopravvissute anche alla fine della prima chiamata di ff.

Alcuni possibili usi della chiusura sono:

- nascondere le variabili della funzione, in modo che non siano accessibili dall'esterno.
- le varie funzioni di grado diverso possono utilizzare il loro stato per comunicare (una funzione figlia può accedere variabili della classe padre).

## Lambda expression

Una chiusura è creata con una lambda expression, una funzione anonima (senza nome) che può essere usata direttamente nel corpo di un metodo.

Esempi di chiusure in vari linguaggi

JavaScript-> function() {body}

- Dato che JS è un linguaggio ad oggetti anche le funzioni sono oggetti, generate dal costruttore Function.

Scala-> (params) => body

- Come JS, anche scala è object-oriented

C++-> auto f = [ext\_var\_ref](args)

- Questa espressione crea una funzione lambda e quindi una chiusura.

C#-> { params => body }

Java8-> (params) -> {body}

- Come C++ e C# con l'unica differenza che qui le variabili sono read-only.

## Alcuni esempi di utilizzo delle lambda expression in C++

- auto f = [=] (int x)      accesses the external variables by copying their value
- auto f = [&] (int x)      accesses the external variables by reference
- auto f = [&v, u, &w] (int x)      accesses v and w by reference, u by copy

### ► Some examples in C#

```
{ int x => x+1 }           (int x) -> x+1  
{ int x, int y => x+y }    (int x, int y) -> x+y  
  
{ string s =>                (String s) -> s + "a"  
int.parse(s) }               (int x) ->  
{ =>                         {System.out.println(x); }  
Console.WriteLine("hi") }
```

### ► Some examples in Java 8

```
(int x) -> x+1  
(int x, int y) -> x+y  
  
(String s) -> s + "a"  
(int x) ->  
{System.out.println(x); }  
}
```

- Le variabili salvate in Java tramite chiusura sono read-only e devono essere necessariamente costanti (final), in caso contrario il compilatore si arrabbia.

```
import java.util.function.*;  
public class ProvaChiusura {  
    public static void main(String  
args[]){  
        int z = 23;  
        IntFunction<Integer> h = (int x) ->  
x + (z--);  
        System.out.println(h.apply(2));  
    }  
}
```

