# Macchine astratte

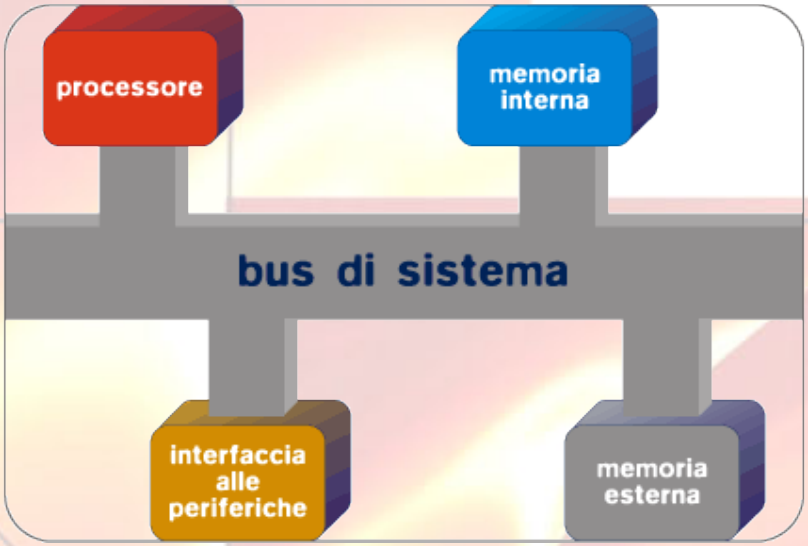
## Interpreti e compilatori

# Linguaggi, macchine astratte, implementazioni

- **Argomenti**:
  - Macchine astratte e linguaggi
  - Gerarchie di macchine astratte e di linguaggi
  - Implementare un linguaggio: Interpreti e Compilatori

# la macchina di von Neumann

Una struttura estremamente semplice




...ma anche estremamente flessibile e suscettibile di molte varianti

# ...e il suo “interprete”

- La macchina esegue un ciclo ripetitivo


programmi e dati sono indistinguibili
e risiedono nella memoria interna

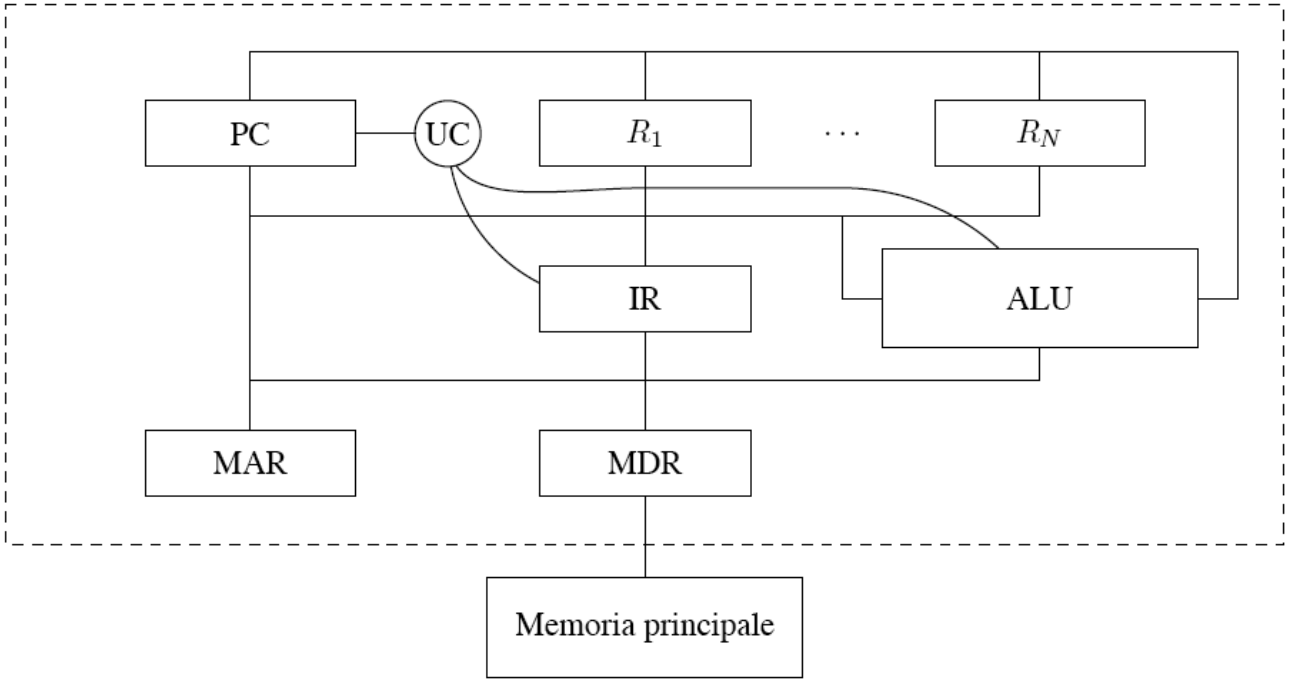
# ...e il suo “interprete”

- La macchina esegue un ciclo ripetitivo

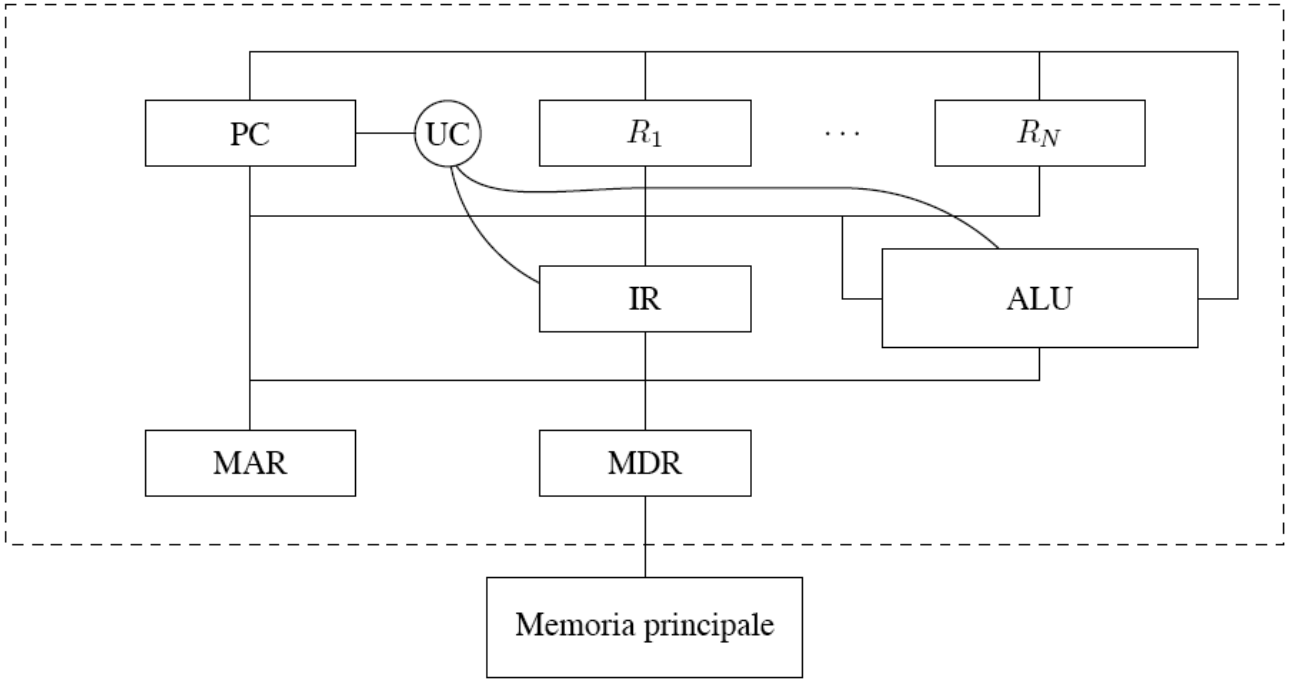

programmi e dati sono indistinguibili
e risiedono nella memoria interna

La macchina esiste per eseguire il proprio linguaggio

# il processore tipico



# il processore tipico




può essere complicato a piacere:

architetture parallele

con *piping*

con *prefetch*

# Morale

- Una macchina fisica esiste per eseguire il suo linguaggio

- Linguaggio e macchina “vengono assieme”, esistono in simbiosi. Ma:
  - una macchina corrisponde ad un (il suo) linguaggio
  - un linguaggio può essere eseguito da più macchine

- Cuore di una macchina fisica:
  - il ciclo fondamentale *fetch-decode-execute*: l’interprete

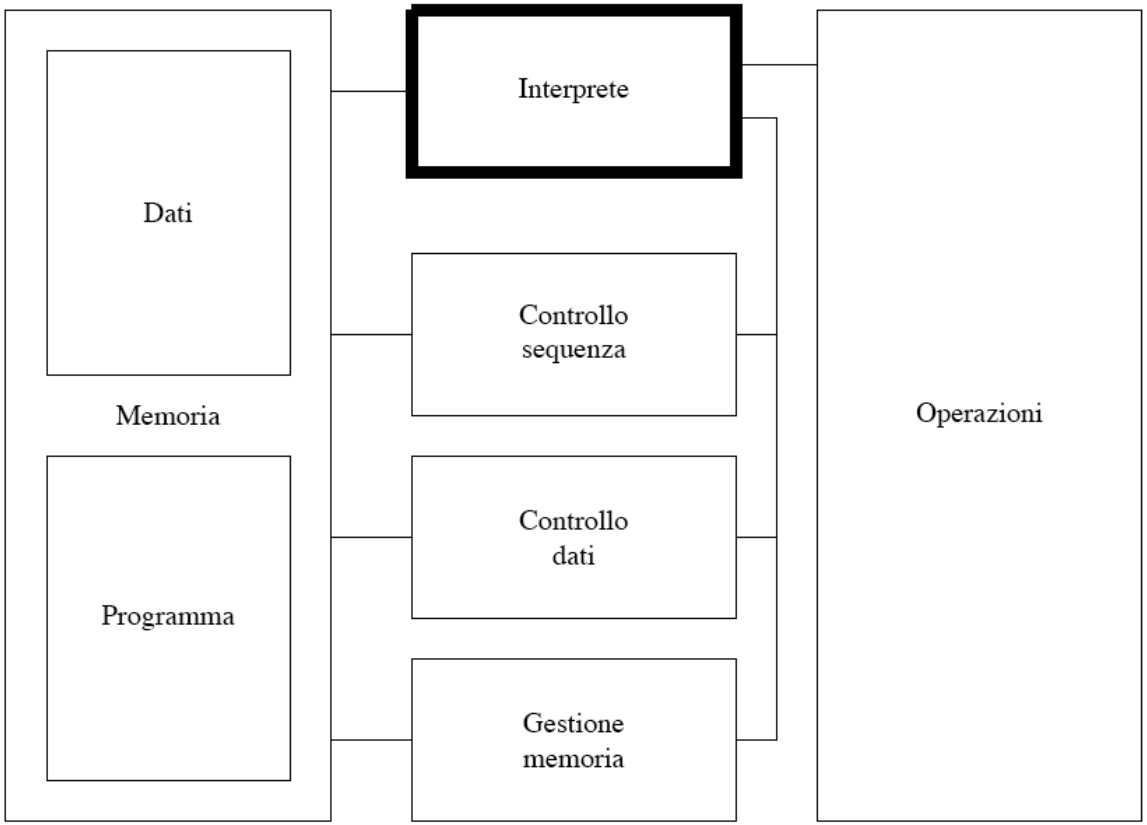
# Morale

- Una macchina fisica esiste per eseguire il suo linguaggio
- Linguaggio e macchina “vengono assieme”, esistono in simbiosi. Ma:
  - una macchina corrisponde ad un (il suo) linguaggio
  - un linguaggio può essere eseguito da più macchine
- Cuore di una macchina fisica:
  - il ciclo fondamentale *fetch-decode-execute*: l’interprete

Una macchina fisica è la realizzazione “a fili” di un particolare algoritmo che, sfruttando alcune strutture dati, è capace di “eseguire” programmi scritti in un certo linguaggio, detto il linguaggio macchina.

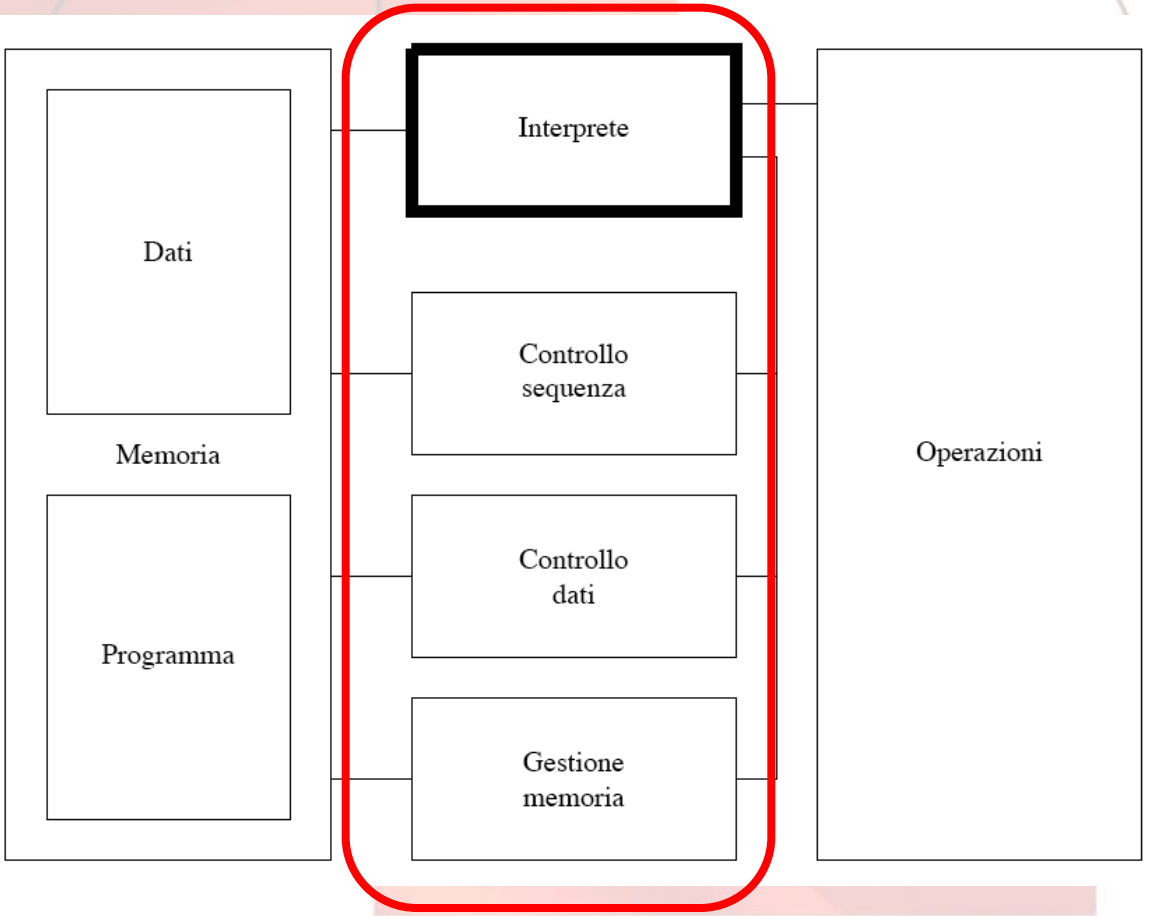
# Macchina astratta

- Una Macchina Astratta (MA) è un insieme di strutture dati ed algoritmi in grado di memorizzare ed eseguire programmi. Componente essenziale è l'interprete



# Macchina astratta

- Una Macchina Astratta (MA) è un insieme di strutture dati ed algoritmi in grado di memorizzare ed eseguire programmi. Componente essenziale è l'interprete



# Interprete

È il componente che *interpreta* le istruzioni ed è costituito da

- operazioni per l'elaborazione dei dati primitivi

  MF: controllo e sfruttamento della ALU

- operazioni e strutture dati per il controllo della sequenza di esecuzione delle operazioni

  MF: incremento del PC; salti

- operazioni e strutture dati per il controllo del trasferimento dei dati

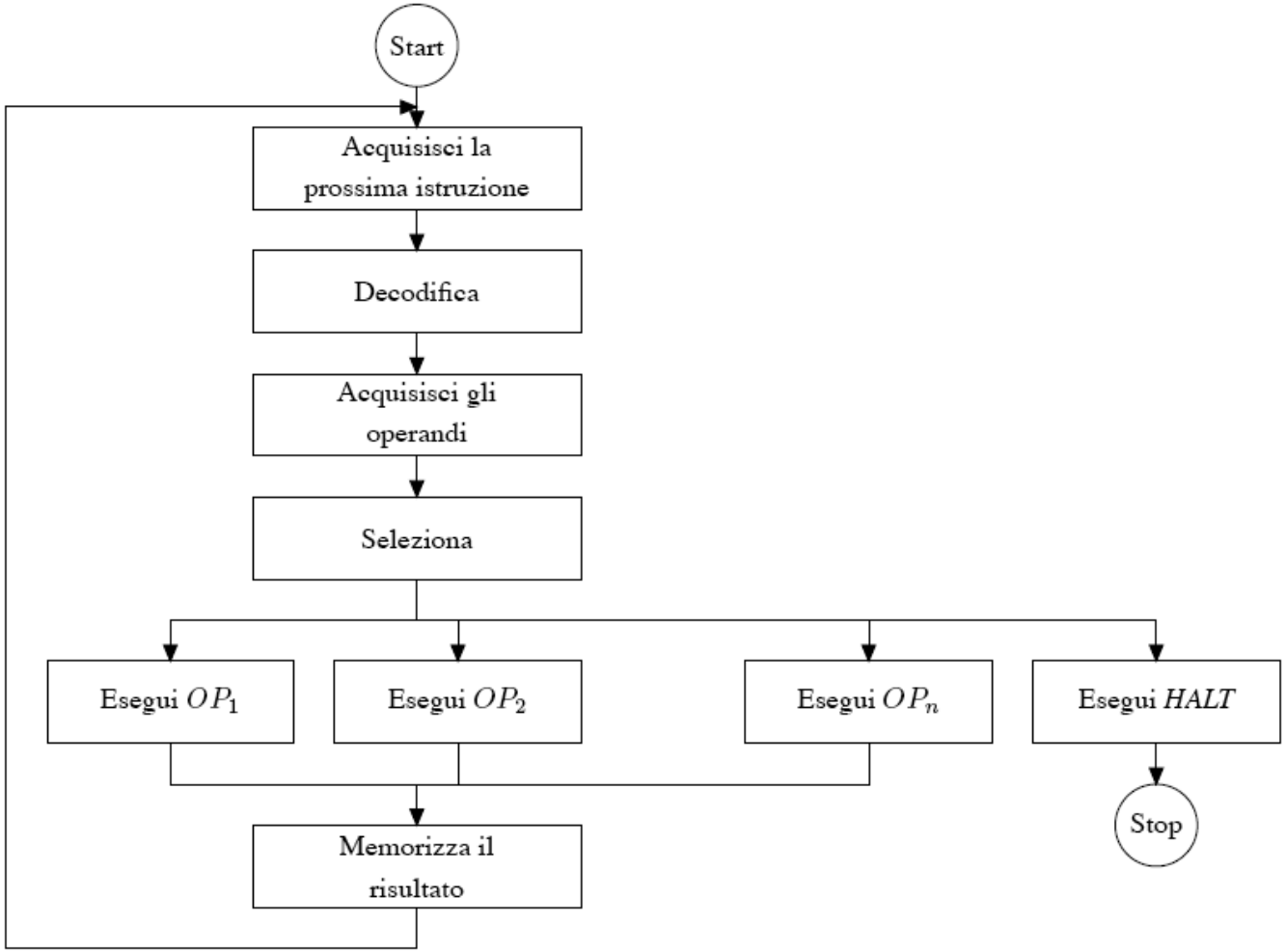
  MF: gestione dei metodi di indirizzamento

- operazioni e strutture dati per la gestione della memoria

  MF: indirizzamento e trasferimento di blocchi ecc.

**La *struttura* dell'interprete è la stessa per una qualunque MA; cambiano i diversi componenti di esso**

# Interprete


Start
Acquisisci la prossima istruzione
Decodifica
Acquisisci gli operandi
Seleziona
Esegui $OP_1$
Esegui $OP_2$
Esegui $OP_n$
Esegui *HALT*
Stop
Memorizza il risultato

# Linguaggio Macchina

- **M** Macchina Astratta
- $\mathbf{L_M}$ Linguaggio Macchina di M
- $\mathbf{L_M}$ è il linguaggio che è "compreso" dall'interprete di **M**:
  - I programmi sono particolari dati primitivi su cui opera l'interprete
- Ai componenti di **M** corrispondono opportuni componenti di $\mathbf{L_M}$
- Diverse rappresentazioni dei programmi scritti in $\mathbf{L_M}$
  - Interna: strutture dati in memoria
  - Esterna: stringa di caratteri

# Linguaggio della Macchina HW

- Set di istruzioni direttamente implementate nella ALU:
  - **CISC** (Complex Instruction Set Computers) – complesse (spesso molte)
  - **RISC** (Reduced Instruction Set Computers) – semplici (a volte poche)
- Tipica istruzione a due operandi (occupa una parola di memoria)

  Codice-operativo  Operando1  Operando2
- Ad esempio (in qualche assembler – rappresentazione esterna)
  - ADD R5, R0

    somma i contenuti dei registri R5 e R0 e memorizza il risultato in R0
  - ADD (R5) (R0)

    somma i contenuti delle celle di memoria i cui indirizzi sono contenuti nei registri R5 e R0, e memorizza il risultato nella cella il cui indirizzo è in R0
- In realtà, rappresentazione interna:
  - Opportune sequenze di bit, dette parole (codice operativo, modo d'indirizzamento e indirizzi assoluti)

# Macchina hw come macchina astratta

La Macchina HW è un tipo (molto concreto…) di MA, il cui interprete esegue:

•Operazioni primitive -> operazioni aritmetico logiche, di manipolazione di stringhe di bit, lettura e scrittura su registri e celle di memoria, input/output

•Controllo sequenza -> Salti, salti condizionali, chiamate di subroutine mediante registro contatore programma e strutture dati per punti di ritorno sottoprogrammi

•Controllo dati -> acquisizione operandi e memorizzazione risultato

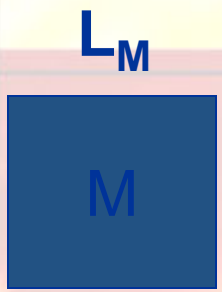
- Architettura a registri -> registri indice e indirizzamento indiretto
- Architettura a pila -> gestione pila

•Gestione memoria ->

- Architettura a registri -> niente (memoria statica)
- Architettura a pila -> allocazione e recupero dati sulla pila.

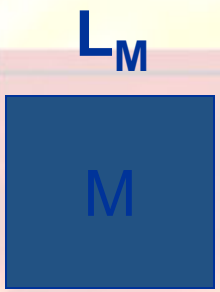
# la macchina hw

- Una processore convenzionale è una (forma molto concreta di...) macchina astratta
- Il suo linguaggio è il linguaggio macchina



# la macchina hw

- Una processore convenzionale è una (forma molto concreta di...) macchina astratta
- Il suo linguaggio è il linguaggio macchina




Un'architettura più sofisticata può interpretare lo stesso linguaggio macchina

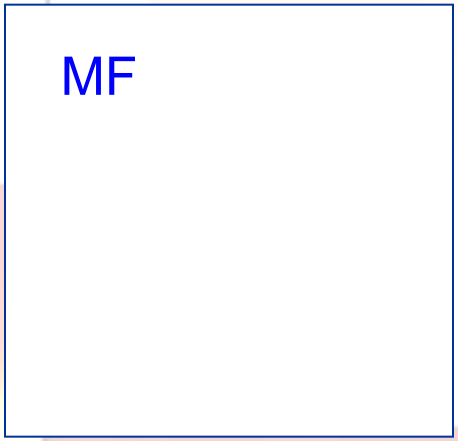
# la macchina hw

- Una processore convenzionale è una (forma molto concreta di...) macchina astratta
- Il suo linguaggio è il linguaggio macchina


cambia l'architettura,
cambia l'interprete,
ma non il linguaggio:
$L_M = L_{Mspider}$

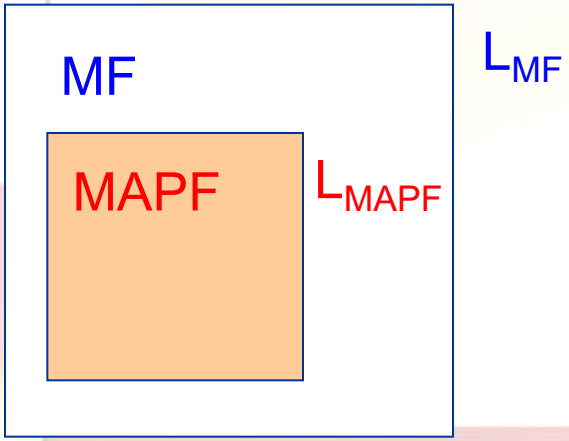
Un'architettura più sofisticata può interpretare lo stesso linguaggio macchina

# Scatole cinesi




- Alcune macchine hw (e.g. non RISC) sono *microprogrammate*:
  - ciclo fetch-execute di MF non è realizzato in hardware;
  - ogni istruzione di $L_{MF}$ è realizzata mediante istruzioni di più basso livello, dette μistruzioni (micro-istruzioni)
  - interpretata da un μinterprete (micro-interprete)

# Scatole cinesi




- Alcune macchine hw (eg non RISC) sono *microprogrammate*:
  - ciclo fetch-execute di MF non è realizzato in hardware;
  - ogni istruzione di $L_{MF}$ è realizzata mediante istruzioni di più basso livello, dette μistruzioni (micro-istruzioni)
  - interpretata da un μinterprete (micro-interprete)

Un programma in linguaggio macchina (in $L_{MF}$):
- viene interpretato da un interprete (scritto in $L_{MAPF}$)
- a sua volta eseguito dal (μ)interprete (hw) di MAPF

# Realizzare una macchina astratta

Possiamo realizzare una MA (con varie combinazioni delle seguenti tecniche):

1) realizzazione in HARDWARE
2) *emulazione* o simulazione via FIRMWARE

1) Teoricamente sempre possibile ma
   - usata solo per macchine di basso livello o macchine dedicate
   - massima velocità
   - flessibilità nulla.

2) strutture dati e algoritmi MA realizzati mediante microprogrammi, che risiedono in una memoria di sola lettura (ROM)
   - macchina ospite (fisica) microprogrammabile
   - alta velocità
   - flessibilità maggiore che HW puro.

# Realizzare una macchina astratta

Possiamo realizzare una MA (con varie combinazioni delle seguenti tecniche):

1) realizzazione in HARDWARE
2) *emulazione* o simulazione via FIRMWARE
3) *interpretazione* o simulazione via SOFTWARE

1) Teoricamente sempre possibile ma
   - usata solo per macchine di basso livello o macchine dedicate
   - massima velocità
   - flessibilità nulla.

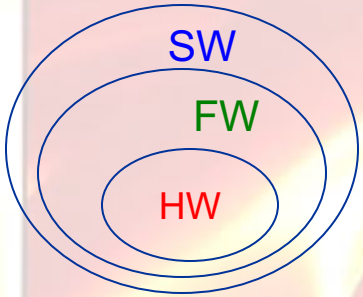
2) strutture dati e algoritmi MA realizzati mediante microprogrammi, che risiedono in una memoria di sola lettura
   - macchina ospite (fisica) microprogrammabile
   - alta velocità
   - flessibilità maggiore che HW puro.

# Realizzazione mediante sw: l'interprete

3) strutture dati e algoritmi della macchina astratta MA realizzati mediante programmi scritti nel linguaggio della macchina ospite MO
   - macchina ospite qualsiasi
   - minore velocità
   - massima flessibilità.

# Realizzazione di una macchina astratta

- Nella realtà, la MA viene realizzata su di una MO fisica mediante una combinazione  delle tre tecniche viste
- Tre livelli, *non necessariamente ogni livello maschera completamente i livelli sottostanti*



# c’era un volta un re

C’era una volta un re, seduto sul sofà, che disse alla sua serva:
“Raccontami una storia!”
E la serva incominciò:

C’era una volta un re, seduto sul sofà, che disse alla sua serva:
“Raccontami una storia!”
E la serva incominciò:

C’era una volta un re, seduto sul sofà, che disse alla sua serva:
“Raccontami una storia!”
E la serva incominciò:

C’era una volta un re, seduto sul sofà, che disse alla sua serva:
“Raccontami una storia!”
E la serva incominciò:

# livelli software

La possibilità di realizzare macchine astratte via software
rende possibili macchine il cui linguaggio è sofisticato

mediante il quale linguaggio realizzare macchine
il cui linguaggio sia ancora più sofisticato

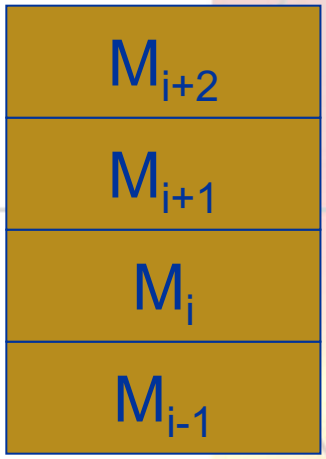
mediante il quale linguaggio realizzare macchine
il cui linguaggio sia ancora più sofisticato

mediante il quale linguaggio realizzare macchine
il cui linguaggio sia ancora più sofisticato

...

# gerarchia di macchine astratte

un'architettura informatica (*hw* o *sw*) si struttura in una serie di macchine astratte gerarchiche




$M_i$:
- usa i servizi forniti da $M_{i-1}$ (cioè il linguaggio $L_{M_{i-1}}$)
- per fornire servizi a $M_{i+1}$(cioè realizzare un interprete per il linguaggio $L_{M_{i+1}}$)
- nasconde (entro certi limiti) la macchina $M_{i-1}$

Stando al livello i può non essere noto (e in genere non serve sapere...) quale sia il livello 0 (hw)

# una gerarchia canonica

# esplicitiamo la gerarchia

Usando un word processor, WP:

interazione *diretta solo* con WP

scrivamo
salviamo il file
cerchiamo altri files
ecc.

usando i menù (i *servizi*, o *primitive*) di WP

# esplicitiamo la gerarchia, 2

Ma per soddisfare la nostra richiesta:

"salva questo file"

WP richiede al livello inferiore (l'interfaccia del SO)

l'apertura di una finestra per il nome del file

l'interfaccia del SO richiede al livello inferiore (il SO)

la disponibilità di un'area di memorizzazione

il SO richiede al *file system*

un indirizzo logico per il file

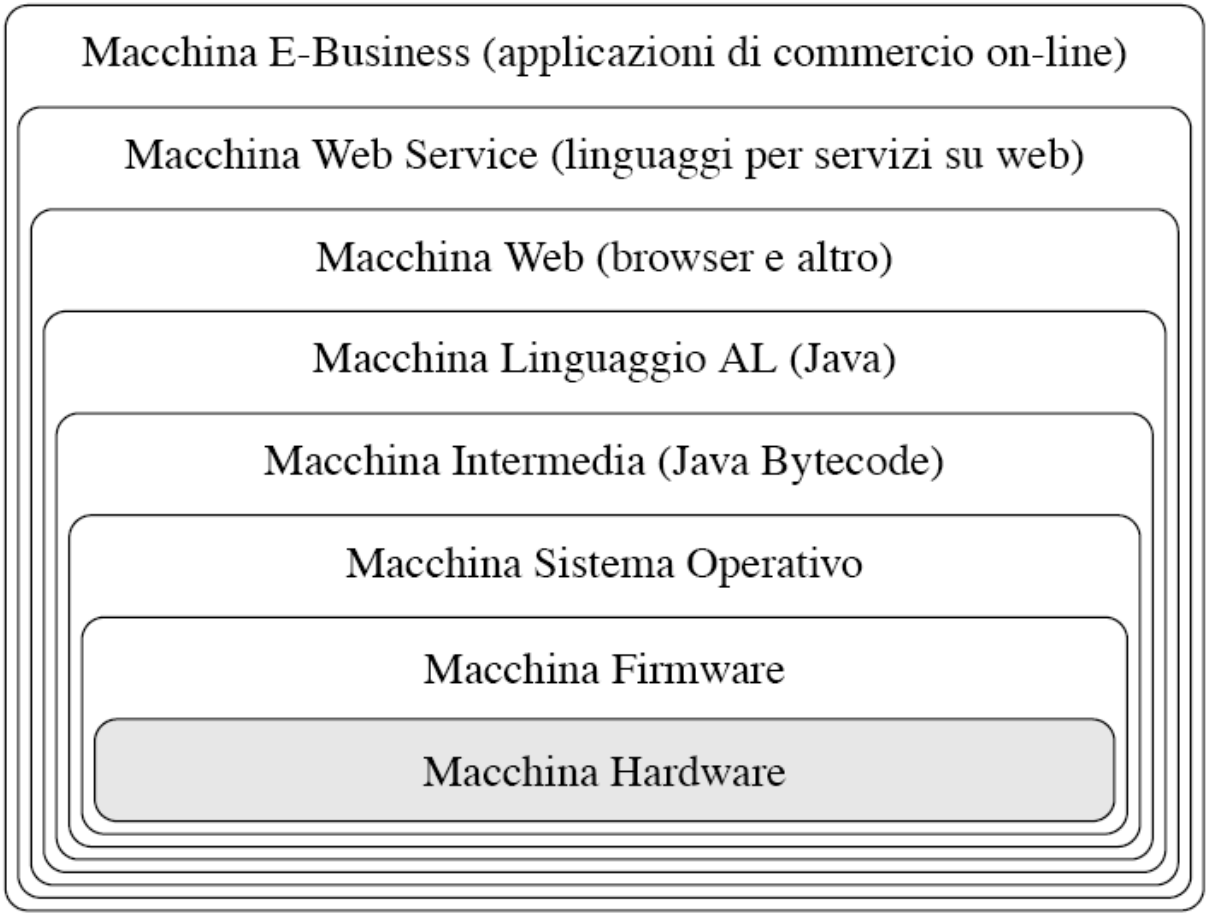
il *file system* richiede alla gestione fisica

l'indice di alcuni blocchi sul disco

.....

# esempio: applicazione

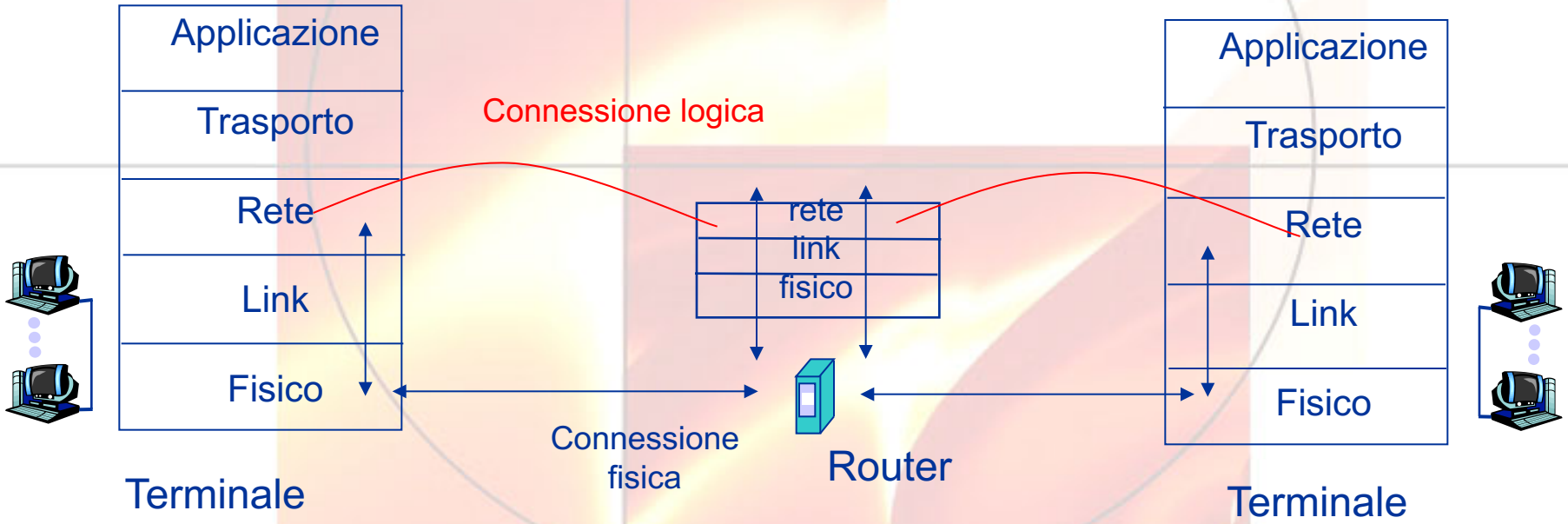
## Esempio: Web application



# esempio: protocolli

# un’analogia famosa

# esempio: i protocolli di internet



# Implementare un linguaggio

- Scartiamo la soluzione hw
- Assimiliamo sw e firmware

- Sono dati:
  - un linguaggio $\mathcal{L}$ da implementare
    - cioè di cui realizzare la macchina astratta $M_{\mathcal{L}}$
  - una macchina astratta $Mo_{\mathcal{L}o}$ (macchina *ospite*)
    - col suo linguaggio $\mathcal{L}o$

- Si vuole
  - implementare $\mathcal{L}$ su $Mo_{\mathcal{L}o}$

- Due modi radicalmente diversi…

# Preliminari e notazione

- Funzione: $f : A \rightarrow B$

  corrispondenza tra elementi di A e di B: ad ogni elemento $a \in A$ la funzione f associa un solo elemento $b \in B$ .

- Funzione *parziale*: $f : A \text{ -o}\rightarrow B$

  corrispondenza che *può* essere non definita su qualche $a \in A$

- $\mathcal{P}_r^{\mathcal{L}}$ indica un programma $\mathcal{P}_r$ scritto nel linguaggio $\mathcal{L}$
- A $\mathcal{P}_r^{\mathcal{L}}$ è associata una funzione parziale $\mathcal{P}^{\mathcal{L}}$ sull'insieme dei dati (non definizione per un certo input = non terminazione del programma per quell'input) che rappresenta la semantica del programma $\mathcal{P}_r^{\mathcal{L}}$ $\quad \mathcal{P}^{\mathcal{L}} : \mathcal{D} \rightarrow \mathcal{D}$

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

# Esempio: semantica di un prog. sequenziale

Frammento di programma sequenziale

$\mathcal{P}_r^{\mathcal{L}}$ = read(x);
If (x == 1) then print(x)
else while true do skip;

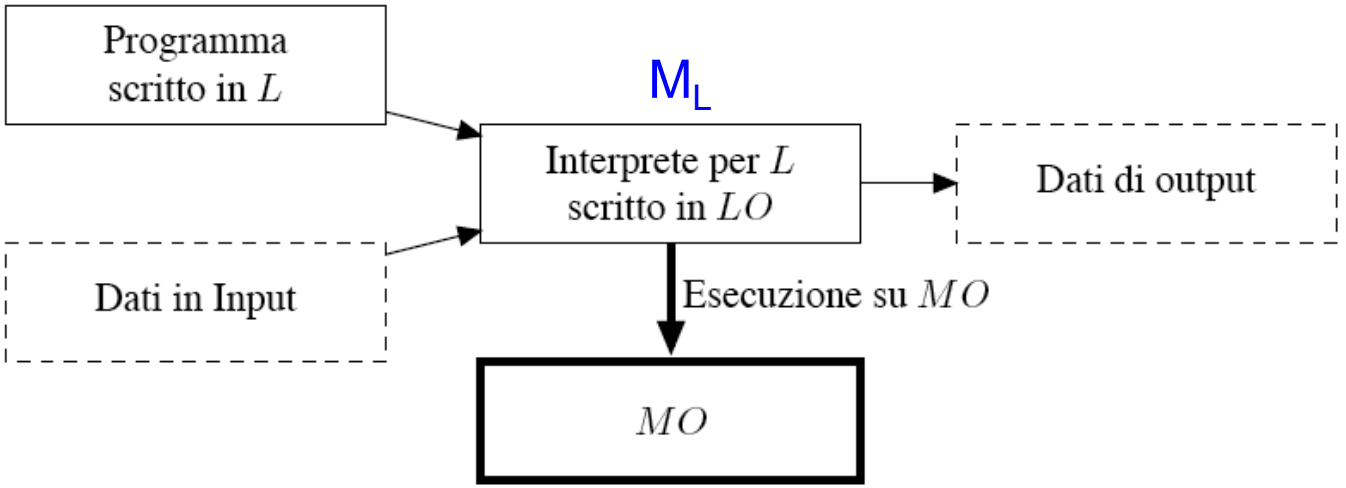
E sua **semantica** come funzione da input ad output

$\mathcal{P}^{\mathcal{L}}$ (x) = 1 se x = 1, indefinita altrimenti

(N.B.: per programmi concorrenti la questione è più delicata)

# Implementazione interpretativa pura

- $M_{\mathcal{L}}$ è realizzata scrivendo un *interprete* per $\mathcal{L}$ su $Mo_{\mathcal{L}o}$:




- Più formalmente

**Definizione 1.3 (Interprete)** *Un interprete per il linguaggio $\mathcal{L}$, scritto nel linguaggio $\mathcal{L}o$, è un programma che realizza una funzione parziale*

$$\mathcal{I}_{\mathcal{L}}^{\mathcal{L}o} : (\mathcal{P}rog^{\mathcal{L}} \times \mathcal{D}) \rightarrow \mathcal{D} \quad tale\ che \quad \mathcal{I}_{\mathcal{L}}^{\mathcal{L}o}(\mathcal{P}_{\mathsf{r}}^{\mathcal{L}}, Input) = \mathcal{P}^{\mathcal{L}}(Input). \quad (1.1)$$

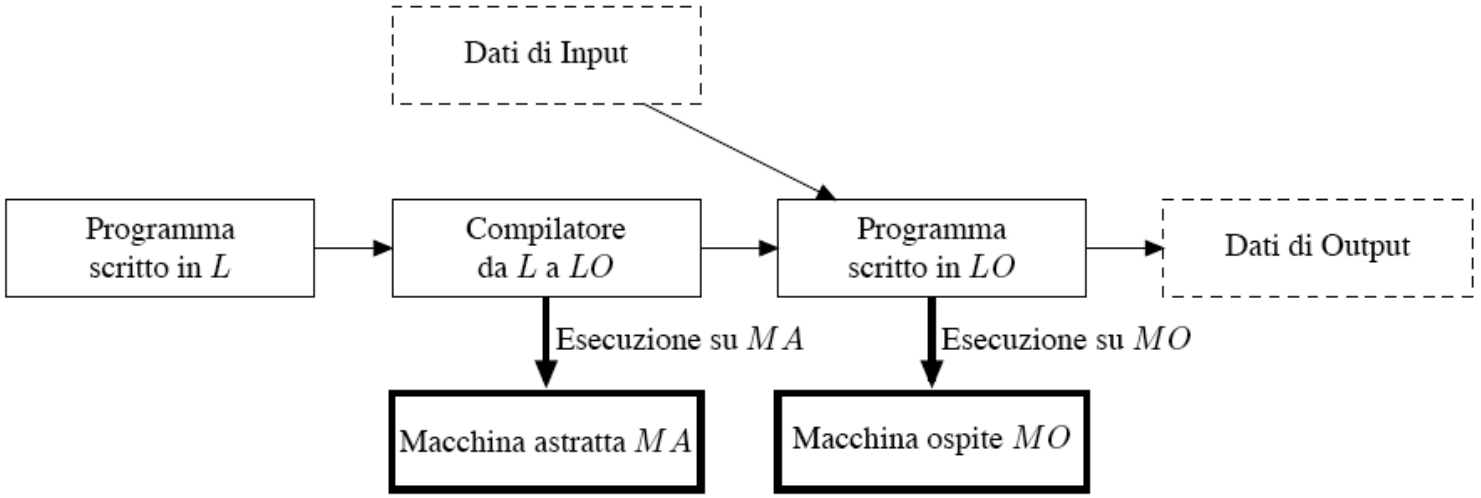
Ovvero l'interprete "calcola la corretta semantica" del programma!

# Implementazione compilativa pura

- I programmi in $\mathcal{L}$ sono *tradotti* in programmi *equivalenti* in $\mathcal{L}o$
- Traduzione effettuata da un (altro) programma

$$C^{\mathcal{L}a}_{\mathcal{L},\mathcal{L}o}$$

il compilatore da $\mathcal{L}$ a $\mathcal{L}o$ scritto in $\mathcal{L}a$



# Implementazione compilativa pura, 2

- Più formalmente:

**Definizione 1.4 (Compilatore)** *Un compilatore da $\mathcal{L}$ a $\mathcal{L}o$ è un programma che realizza una funzione*

$$\mathcal{C}_{\mathcal{L},\mathcal{L}o} : \mathcal{P}rog^{\mathcal{L}} \rightarrow \mathcal{P}rog^{\mathcal{L}o}$$

*tale che, dato un programma $\mathcal{P}_r^{\mathcal{L}}$, se*

$$\mathcal{C}_{\mathcal{L},\mathcal{L}o}(\mathcal{P}_r^{\mathcal{L}}) = \mathcal{P}_r c^{\mathcal{L}o} \tag{1.2}$$

*allora, per ogni $Input \in \mathcal{D}^5$,*

$$\mathcal{P}^{\mathcal{L}}(Input) = \mathcal{P}c^{\mathcal{L}o}(Input). \tag{1.3}$$

Ovvero il compilatore "preserva la semantica" del programma: il programma originale $P_r^L$ e quello tradotto $P_r c^{L0}$ calcolano la stessa funzione: $P^L = Pc^{L0}$ !

# Intermezzo

- Cosa indicano le seguenti scritture?

- $I^{\mathcal{L}0}_{\mathcal{L}1}$ = *un interprete scritto in* $\mathcal{L}0$ *che esegue programmi scritti in* $\mathcal{L}1$
- *La macchina ospite è* $\mathcal{M}_{\mathcal{L}0}$, *mentre la macchina astratta realizzata su di essa dall'interprete è* $\mathcal{M}_{\mathcal{L}1}$
- $I^{\mathcal{L}0}_{\mathcal{L}1}(\mathcal{P}^{\mathcal{L}1}, x)$ = *risultato del calcolo del programma* $\mathcal{P}$ *(scritto in* $\mathcal{L}1$*) con* $x$ *come dato in input*

- $C^{\mathcal{L}0}_{\mathcal{L}1,\mathcal{L}2}$ = *un compilatore scritto in* $\mathcal{L}0$ *che traduce programmi scritti in* $\mathcal{L}1$ *in equivalenti programmi scritti in* $\mathcal{L}2$

# Intermezzo (2)

- Cosa indicano le seguenti scritture?

- $I^{L0}_{L1}(C^{L1}_{L2,L3}, P^{L2}) = P1^{L3}$ *cioè l'interprete, eseguito sulla macchina ospite* $ML_0$, *realizza la macchina astratta* $M_{L1}$ *ed esegue un compilatore (scritto in L1) che traduce il programma P (scritto in L2) in un equivalente programma P1 (scritto in L3)*

# Compilazione o interpretazione ?

- Implementazione interpretativa pura:
  - scarsa efficienza della macchina $\mathbf{M}_{\mathcal{L}}$

Ai tempi di esecuzione, vanno aggiunti i tempi necessari alla decodifica (mentre in impl. Compilativa, la traduzione viene fatta prima di eseguire)

Esempio:

Istruzione for (I = 1, I <= n, I = I+1) C;

Decodifica (al momento)

```
P2:
     R1 = 1
     R2 = n
L1:  if R1 > R2 then goto L2
     traduzione di C
     ...
     R1 = R1 + 1
     goto L1
L2:  ....
```

Se la stessa istruzione viene eseguita più volte, più volte l'interprete dovrà decodificarla (ad esempio C sopra)

# Compilazione o interpretazione ? (2)

- Implementazione interpretativa pura:
  - buona flessibilità: permette di interagire con l'esecuzione del programma (facile generare strumenti di debugging "dinamici" step-by-step)
  - Più facile da realizzare: il compilatore è di solito molto più complesso
  - Occupa meno memoria, perché non viene effettivamente generato codice da memorizzare (questione poco rilevante oggigiorno, ma che aveva interesse quando le memorie erano molto piccole e costose).

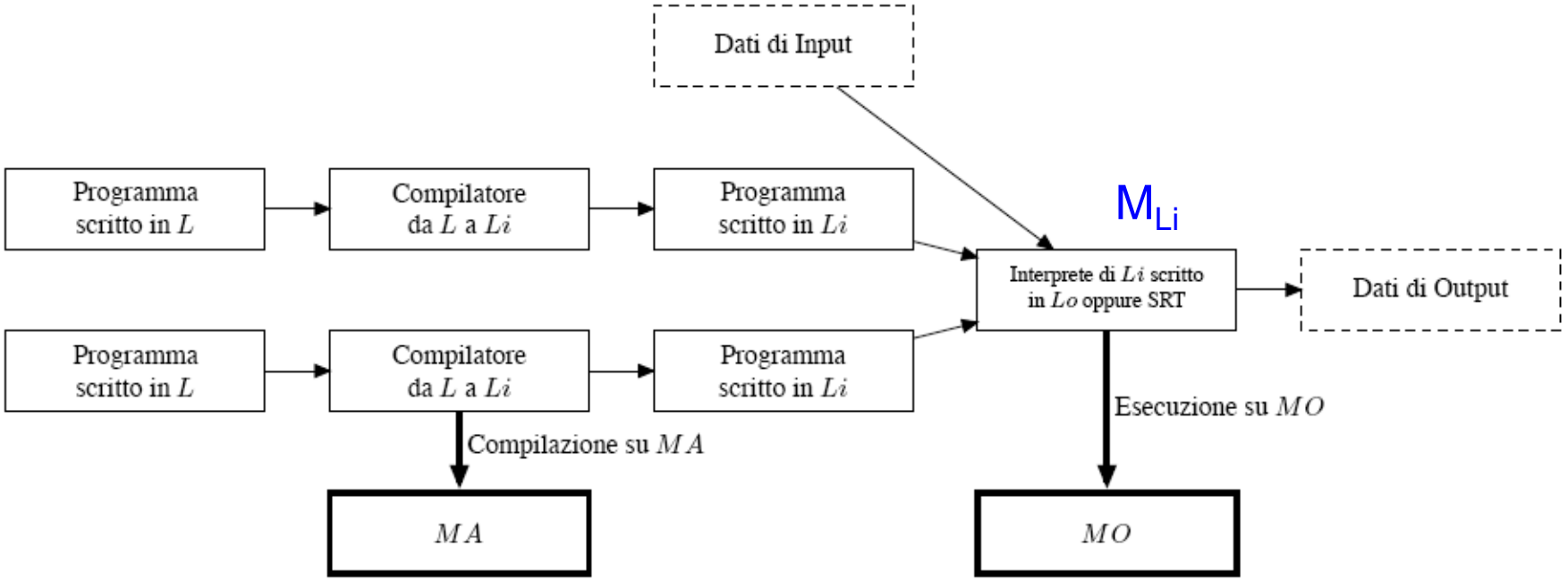
# Compilazione o interpretazione ? (3)

- Implementazione compilativa pura:
  - difficile, data la lontananza fra $\mathcal{L}$ e $\mathcal{L}o$
  - buona efficienza:
    - 1) costo decodifica a carico del compilatore
    - 2) ogni istruzione è tradotta una sola volta
  - scarsa flessibilità
  - perdita di info sulla struttura (astrazione) del programma sorgente (ad esempio: se si verifica un errore a run-time potrebbe essere molto difficile capire quale istruzione del programma sorgente lo ha determinato)
  - (occupazione di memoria del codice prodotto)

# Nel caso reale

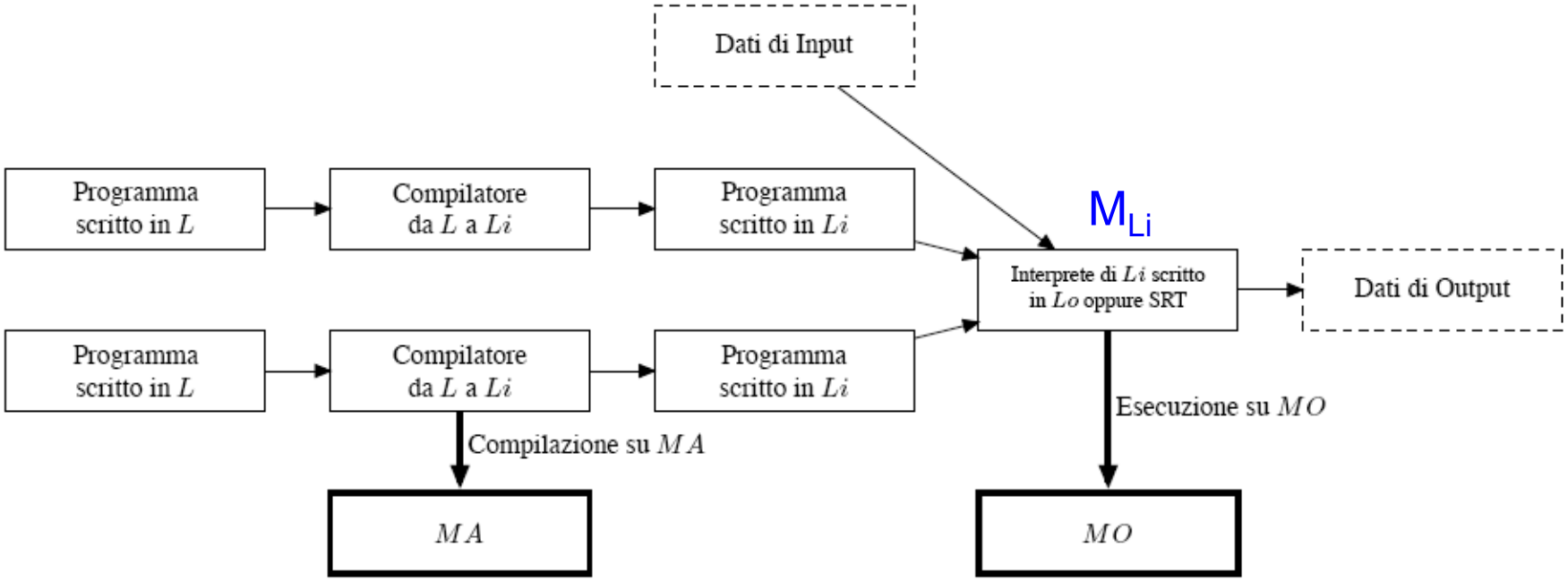
- **Entrambe le componenti coesistono**
  1) Alcune istruzione (es. ingresso/uscita) sono sempre simulate
  2) I programmi devono essere tradotti nella rappresentazione interna o in un codice intermedio

- La macchina *intermedia* ed il suo linguaggio $\mathcal{L}i$:



# Implementazione di tipo interpretativo

- L'interprete della macchina intermedia $M_{\mathcal{L}i}$ è sostanzialmente diverso dall'interprete della macchina ospite $M_{\mathcal{L}o}$



# Esempio: Java

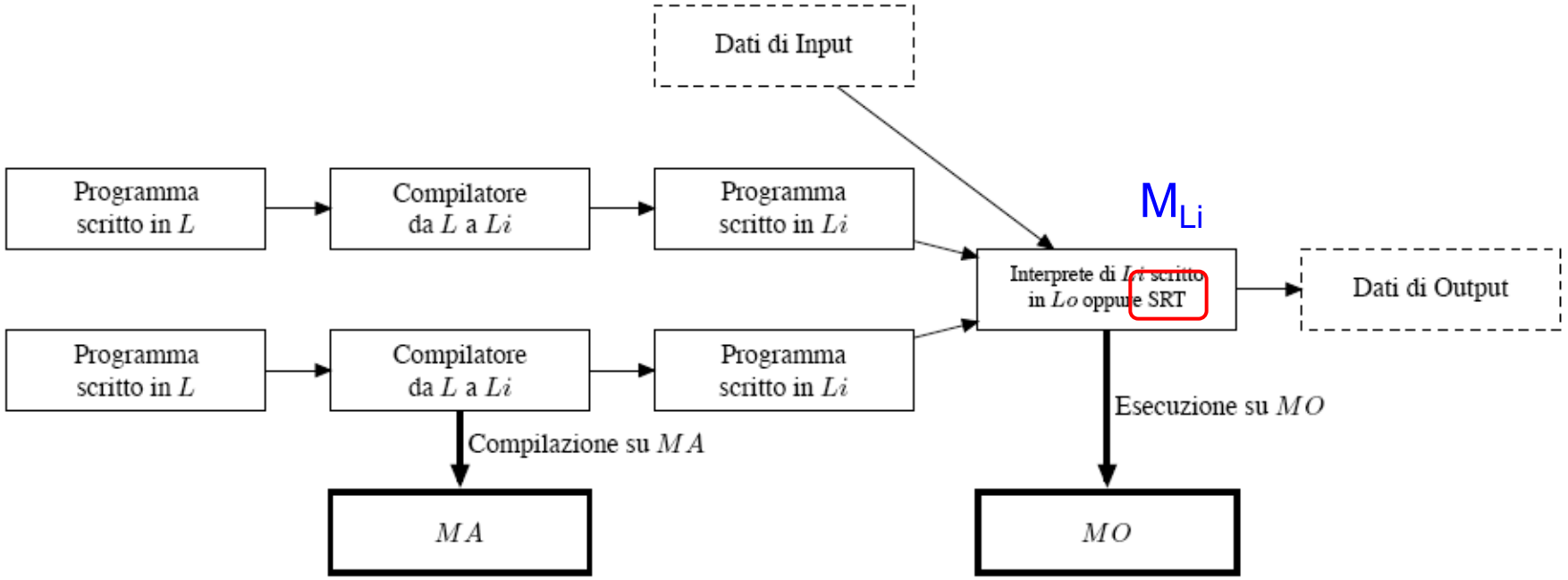
# Esempio: Java


$L_{MA}$ = Java
$L_{Mi}$ = Bytecode
Mi = JVM
M0 = opportuna macchina che esegue l'interprete della JVM

# Implementazione di tipo compilativo

- Interprete della macchina intermedia $M_{\mathcal{L}i}$ = interprete della macchina ospite $M_{\mathcal{L}o}$ + opportuni meccanismi (p.e. I/O, gestione della memoria ecc.)
  =

*supporto a run time di $\mathcal{L}$*



# Esempio: C


$L_{MA}$ = C
$L_{Mi}$ = codice generato da `cc`
Mi = ?
M0 sembra inutile: coincide con Mi

# Esempio: C


$L_{MA}$ = C
$L_{Mi}$ = codice generato da cc
Mi = ?
M0 ~~sembra inutile: coincide con Mi~~ NO

# Esempio: C


$L_{MA}$ = C
$L_{Mi}$ = linguaggio generato da cc + supporto per gestione memoria, I/O ecc.
Mi = M0 + interprete per le chiamate del supporto a run-time
M0 = macchina ospite

supporto a run-time

# Cosa compilare e cosa interpretare

- In linea di principio:
  - traduzione per quei costrutti di $\mathcal{L}$ che corrispondono da vicino a costrutti di $\mathcal{L}o$;
  - simulazione (cioè: interpretazione) per gli altri.

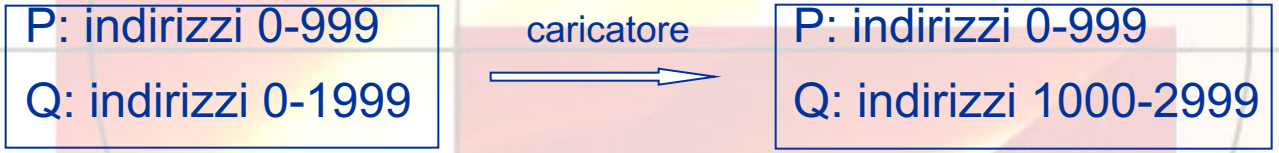
- soluzione di tipo compilativo
  - privilegia **l'efficienza**
- soluzione di tipo interpretativo
  - privilegia **flessibilità e portabilità**

# Linguaggi reali

- Linguaggi tipicamente implementati in modo compilativo
  - **C, C++, FORTRAN, Pascal, ADA**
- Linguaggi tipicamente implementati in modo interpretativo
  - **LISP, ML, Perl, Postscript, Pascal, Prolog, Smalltalk, Java**

# Esempi

- **Programmi Macchina Rilocabili**:
  - Implementazione *puramente* compilativa
  - Il traduttore è il caricatore (o link editor):
    - traduce codice macchina rilocabile in codice macchina eseguibile
  - Interprete = interprete HW (o FW)
- Es.




- **Linguaggio ASSEMBLER**
  - Implementazione *puramente* compilativa
  - Il traduttore è l'assemblatore:
    - traduce codice ASSEMBLER (che usa nomi simbolici) in codice macchina (rilocabile)

# Esempi

- **FORTRAN (77)**
  - Implementazione di tipo compilativo (non puro!)
  - traduzione fatta da compilatore che produce codice eseguibile della macchina ospite
  - Linker per unire i vari sottoprogrammi
  - Supporto a Run-Time = programmi (in linguaggio di $\mathbf{M_o}$) che simulano I/O e alcune operazioni aritmetiche non fornite da $\mathbf{M_o}$
  - Supporto a Run-Time minimo:
    - no strutture dati dinamiche
    - no ricorsione
    - no gestione dinamica della memoria (Eccetto FORTRAN 90)

- Dunque: FORTRAN è compilato in un linguaggio $\mathcal{L}_I$ la cui macchina astratta estende $M_o$ con il supporto a run-time

# Esempi

- **PASCAL e C**
  - Implementazione di tipo compilativo:
  - traduzione fatta da compilatore che produce codice eseguibile della macchina ospite;
  - Supporto a Run-Time  per
    - operazioni di  I/O
    - gestione (dinamica) memoria (puntatori, heap)
    - gestione controllo sequenza (ricorsione)
    - controllo dati (blocchi in Pascal), sottoprogrammi, ambiente locale dinamico
  - Necessaria estesa simulazione per realizzare queste componenti, usando opportune strutture dati (pile).

# Esempi

- **PASCAL**
  - implementazione di tipo interpretativo
  - traduzione fatta da compilatore che produce codice di una macchina intermedia (P-code)
  - macchina intermedia a Pila, con interprete e strutture simili a quelle di $M_{pascal}$
  - interprete di P-code simulato su macchina ospite
  - più portabile dell'implementazione compilativa

# L'interprete e il compilatore si possono sempre realizzare?

- Interprete esegue $L_{sorg}$ ed è scritto in un linguaggio $L_{osp}$
- Compilatore traduce $L_{sorg}$ in $L_{dest}$ ed è scritto in un linguaggio $L_{osp}$
- Compilatore preserva la semantica del programma tradotto, quindi preserva l'insieme delle funzioni che il linguaggio $L_{sorg}$ può calcolare!
- Allora è necessario che il linguaggio $L_{dest}$ sia non meno espressivo di $L_{sorg}$, ovvero che calcoli "almeno quanto" $L_{sorg}$.
- Stessa cosa deve valere per $L_{osp}$
- Fondamenti: Macchine di Turing e Turing-completezza
- Tutti i linguaggi di programmazione "veri" sequenziali sono Turing-completi (se forniti di memoria "illimitata"), ovvero sono ugualmente espressivi.
- Vedremo formalismi/linguaggi molto meno espressivi ma che sono utilissimi perché gli algoritmi relativi sono efficienti:
  - Automi finiti (analizzatori lessicali)
  - Automi a pila (analizzatori sintattici)

# Come viene generato un compilatore

- **Strumenti automatici**
  - Lex generatore di analizzatori lessicali
  - Yacc generatore di analizzatori sintattici

  data una descrizione formale della sintassi producono codice che riconosce le stringhe sintatticamente legali

- **Implementazione via kernel**

per implementare $\mathcal{L}$ si scrive un interprete o un compilatore in $\mathcal{H}$, sottoinsieme ristretto di $\mathcal{L}$

$C^{\mathcal{H}}{}_{\mathcal{L},\mathcal{L}o}$ compilatore scritto in $\mathcal{H}$ che traduce $\mathcal{L}$ in $\mathcal{L}o$

$I^{\mathcal{H}}{}_{\mathcal{L}}$ interprete per $\mathcal{L}$, scritto in $\mathcal{H}$

Si implementa poi a mano un compilatore (o un interprete) per $\mathcal{H}$

se $\mathcal{H}$ è piccolo e ben scelto, non richiede grande sforzo

# Implementazione via kernel

- Per implementare $\mathcal{L}$
  - Si implementa un opportuno sottoinsieme $\mathcal{H}$ di $\mathcal{L}$
  - Si usa $\mathcal{H}$ come linguaggio di implementazione
- Usato per i sistemi operativi:
  - si realizza prima il nucleo e poi l'intero sistema usando le primitive offerte dal nucleo
- Semplifica l'implementazione: il linguaggio $\mathcal{H}$ è più vicino di $\mathcal{L}$ a quello della macchina ospite
- Facilita la portabilità: basta re-implementare $\mathcal{H}$ nel nuovo linguaggio macchina

# Generazione di compilatori: bootstrapping

- I primi ambienti Pascal includevano
  - un compilatore in Pascal da Pascal a P-code: $C^{\text{Pascal}}_{\text{Pascal,P-Code}}$
  - lo stesso compilatore, tradotto in P-code: $C^{\text{P-Code}}_{\text{Pascal,P-Code}}$
  - un interprete per P-code, scritto in Pascal: $I^{\text{Pascal}}_{\text{P-code}}$

- Per aver un'implementazione locale su una specifica Mo:
  - produci (a mano) una traduzione di $I^{\text{Pascal}}_{\text{P-code}}$ nel linguaggio di Mo: $I^{\mathcal{L}o}_{\text{P-code}}$

- Eseguire su Mo un programma P in Pascal su dati x:

$$I^{\mathcal{L}o}_{\text{P-code}}(C^{\text{P-Code}}_{\text{Pascal,P-Code}}, \text{P}) = \text{P'} \quad \text{(scritto in P-code)}$$
$$I^{\mathcal{L}o}_{\text{P-code}}(\text{P'}, \text{x}) = \text{risultato voluto}$$

- Possiamo fare meglio?

# Generazione di compilatori: bootstrapping, 2

- *Migliorare l'efficienza*, con un compilatore scritto in $\mathcal{L}o$
- A mano, facendo hacking di $C^{\text{Pascal}}{}_{\text{Pascal,P-Code}}$, produci $C^{\text{Pascal}}{}_{\text{Pascal},\mathcal{L}o}$
- Adesso abbiamo:
  - $C^{\text{Pascal}}{}_{\text{Pascal,P-Code}}$
  - $C^{\text{P-Code}}{}_{\text{Pascal,P-Code}}$
  - $\mathcal{I}^{\text{Pascal}}{}_{\text{P-code}}$
  - $\mathcal{I}^{\mathcal{L}o}{}_{\text{P-code}}$
  - $C^{\text{Pascal}}{}_{\text{Pascal},\mathcal{L}o}$
- Bootstrapping

$$\mathcal{I}^{\mathcal{L}o}{}_{\text{P-code}}(C^{\text{P-Code}}{}_{\text{Pascal,P-Code}}, C^{\text{Pascal}}{}_{\text{Pascal},\mathcal{L}o}) = C^{\text{P-Code}}{}_{\text{Pascal},\mathcal{L}o}$$

$$\mathcal{I}^{\mathcal{L}o}{}_{\text{P-code}}(C^{\text{P-Code}}{}_{\text{Pascal},\mathcal{L}o}, C^{\text{Pascal}}{}_{\text{Pascal},\mathcal{L}o}) = C^{\mathcal{L}o}{}_{\text{Pascal},\mathcal{L}o}$$

# Esercizi

- Cosa viene calcolato nei seguenti casi?

- $\mathcal{I}^{\mathcal{L}0}_{\mathcal{L}1}(C^{\mathcal{L}1}_{\mathcal{L}2,\mathcal{L}3}, C^{\mathcal{L}2}_{\mathcal{L}3,\mathcal{L}0}) = ?$
- $\mathcal{I}^{\mathcal{L}0}_{\mathcal{L}1}(C^{\mathcal{L}1}_{\mathcal{L}0,\mathcal{L}2}, \mathcal{I}^{\mathcal{L}0}_{\mathcal{L}1}) = ?$
- $\mathcal{I}^{\mathcal{L}0}_{\mathcal{L}1}(C^{\mathcal{L}1}_{\mathcal{L}2,\mathcal{L}0}, \mathcal{I}^{\mathcal{L}2}_{\mathcal{L}1}) = ?$
- $\mathcal{I}^{\mathcal{L}0}_{\mathcal{L}1}(C^{\mathcal{L}2}_{\mathcal{L}0,\mathcal{L}1}, C^{\mathcal{L}0}_{\mathcal{L}2,\mathcal{L}1}) = ?$

# Soluzioni

- $I^{L0}_{L1}(C^{L1}_{L2,L3}, C^{L2}_{L3,L0}) = C^{L3}_{L3,L0}$
- $I^{L0}_{L1}(C^{L1}_{L0,L2}, I^{L0}_{L1}) = I^{L2}_{L1}$
- $I^{L0}_{L1}(C^{L1}_{L2,L0}, I^{L2}_{L1}) = I^{L0}_{L1}$

Attenzione! L'interpete ottenuto non è lo stesso interprete da cui siamo partiti, ma solo equivalente.

- $I^{L0}_{L1}(C^{L2}_{L0,L1}, C^{L0}_{L2,L1}) = errore$

non si può perché l'interprete non può eseguire un compilatore scritto in L2.