



Macchine astratte

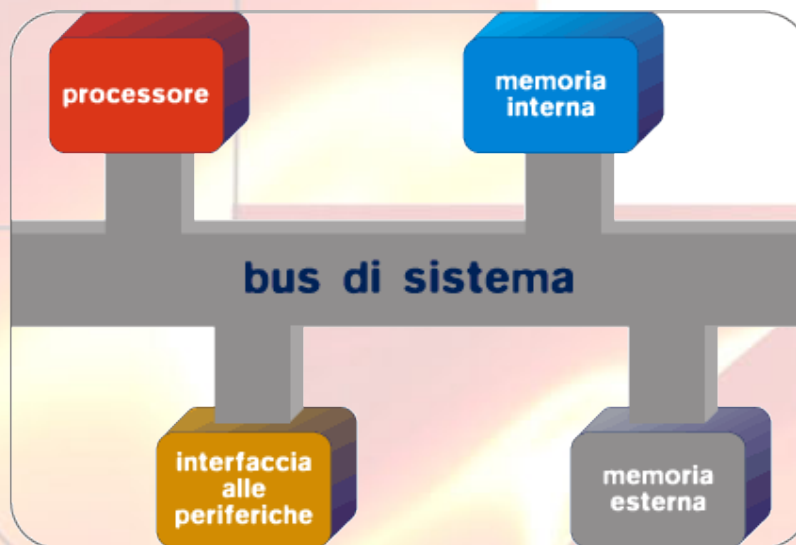
Interpreti e compilatori

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”

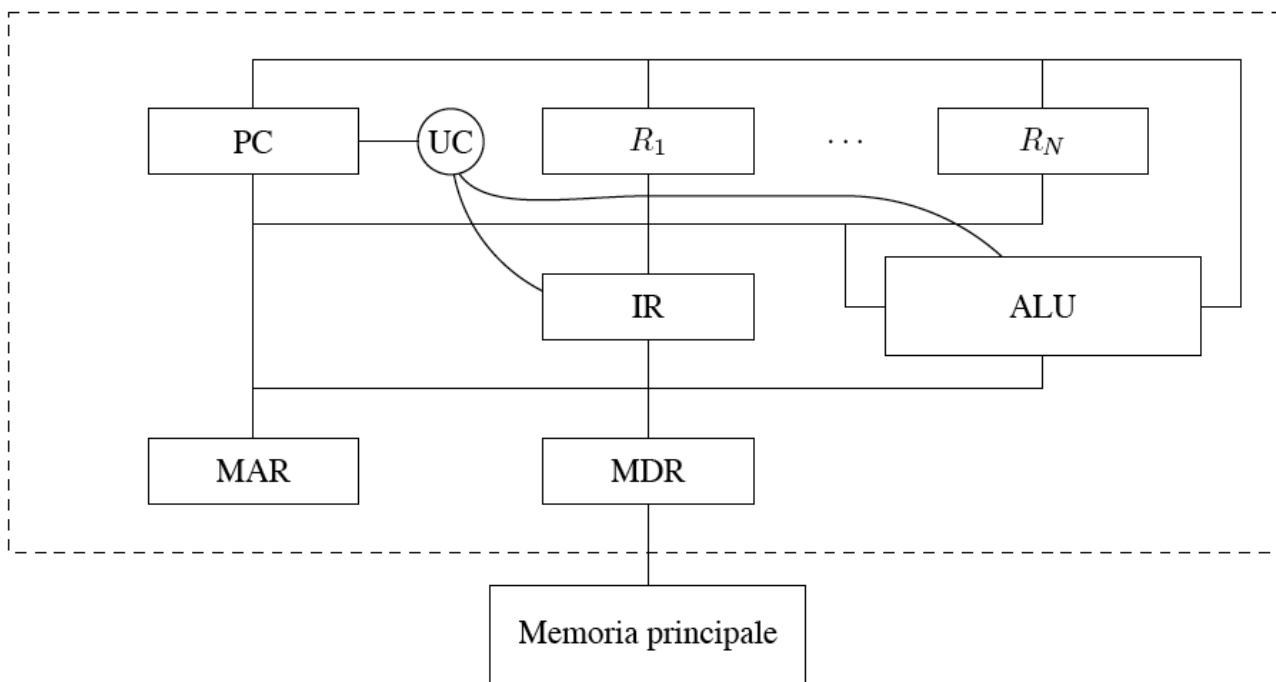
- La macchina esegue un ciclo ripetitivo



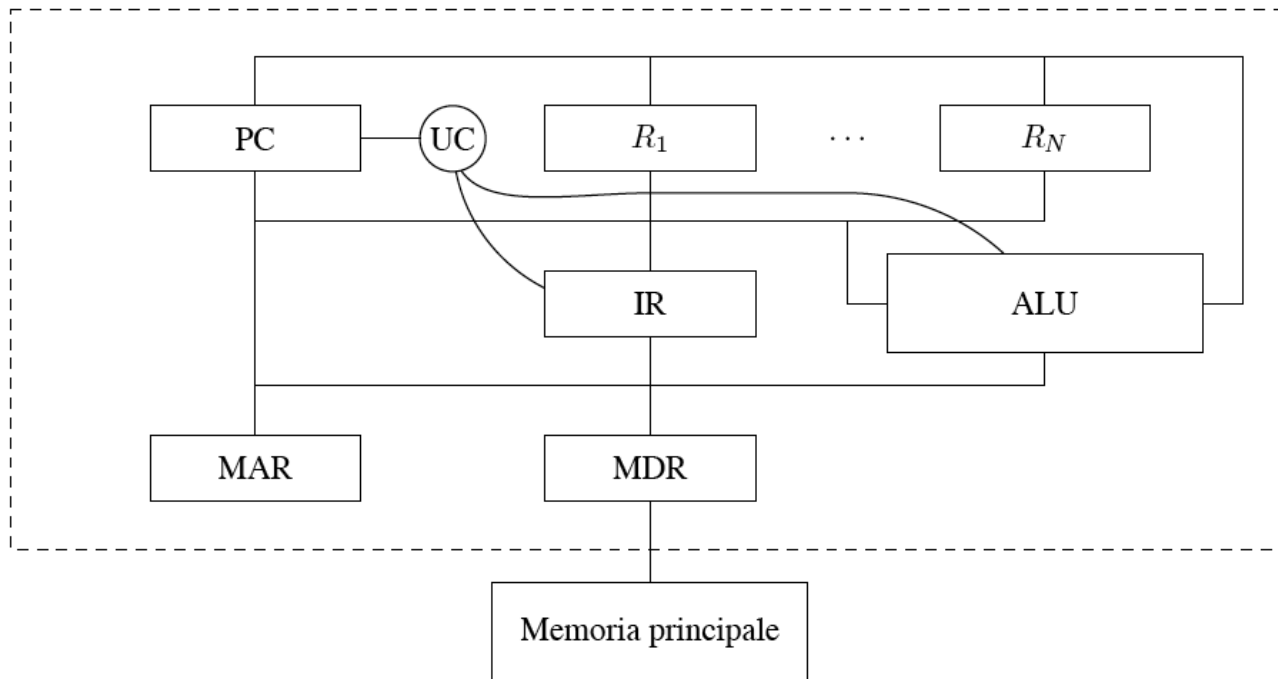
programmi e dati sono indistinguibili
e risiedono nella memoria interna

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

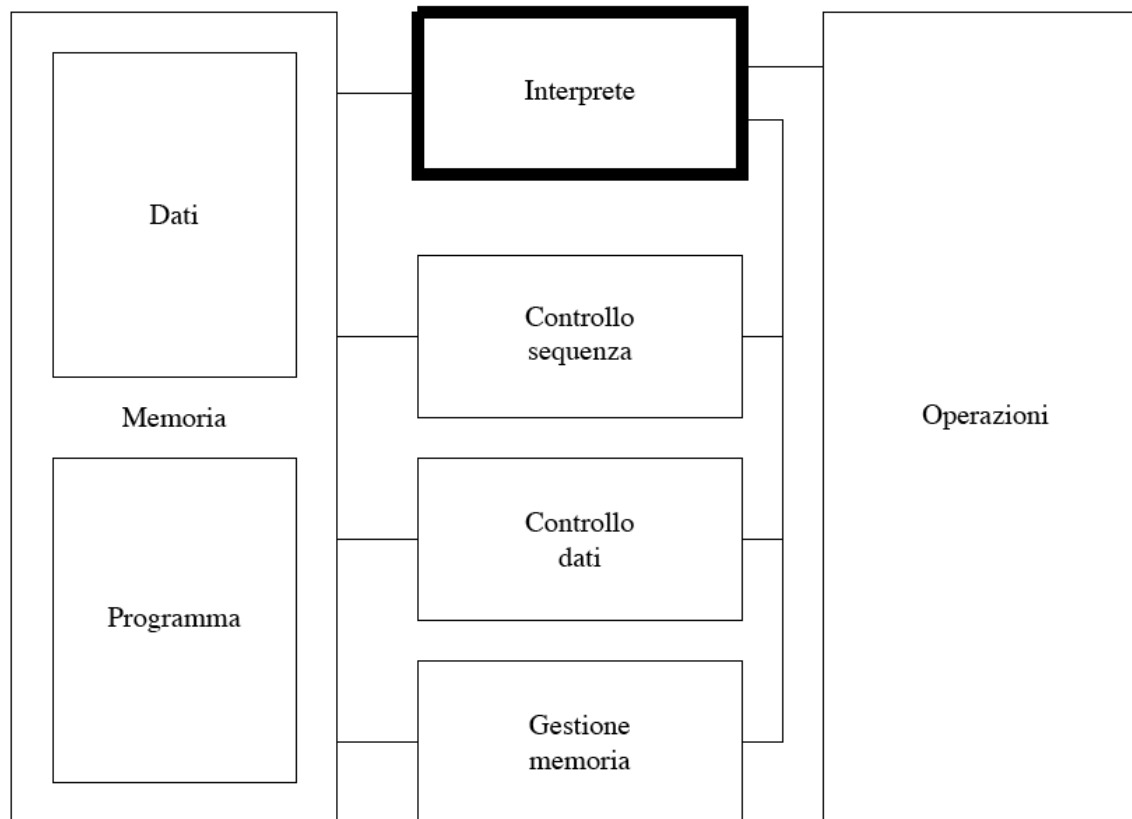
Morale

- **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**.

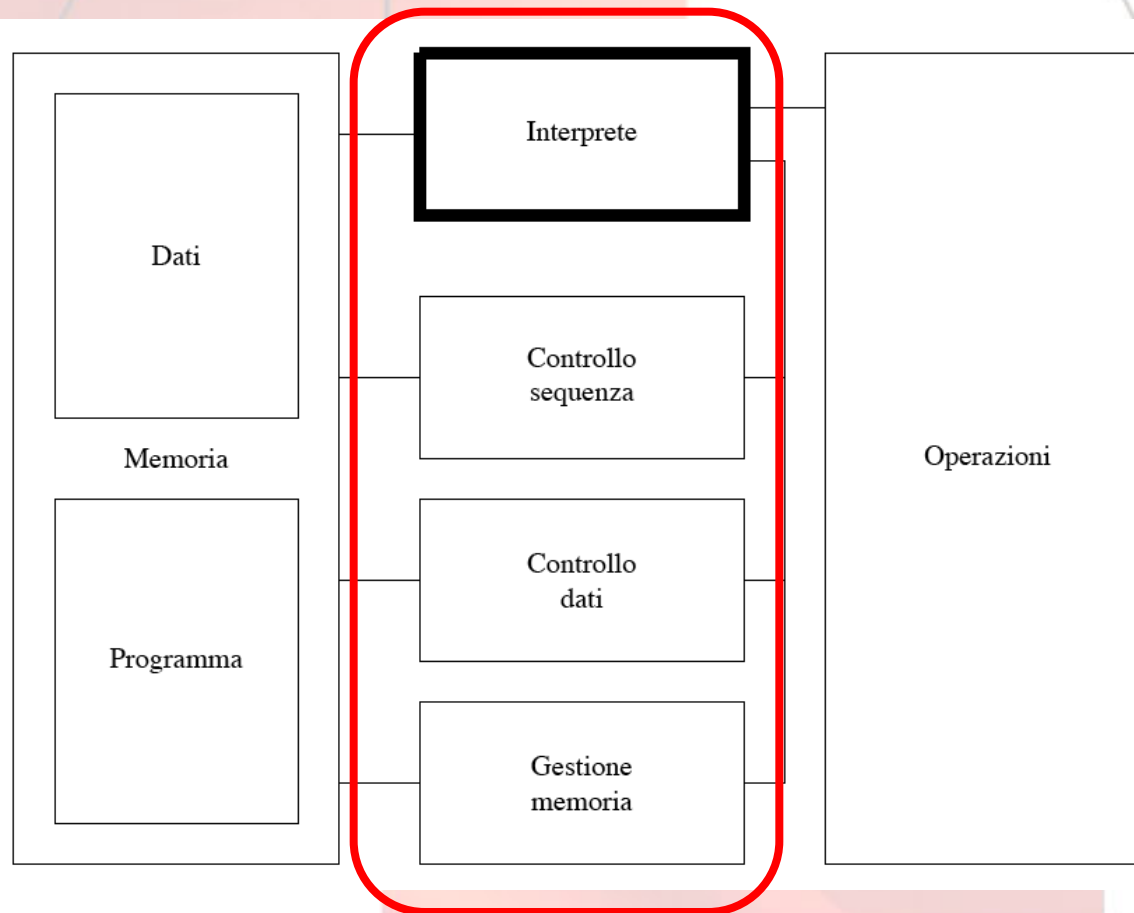
Macchina astratta

- 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



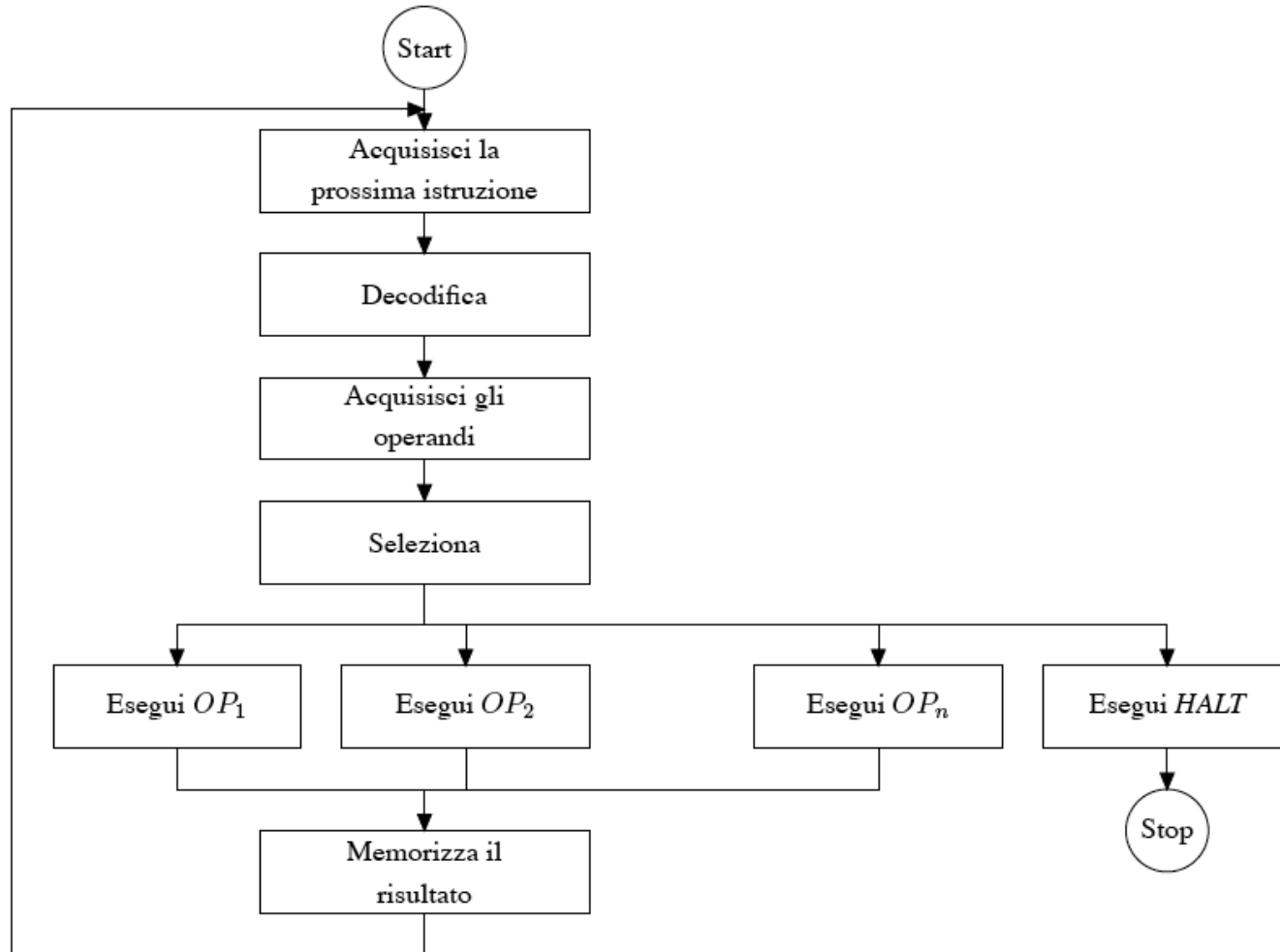
Interprete

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



Linguaggio Macchina

- **M** Macchina Astratta
- **L_M** Linguaggio Macchina di M
- **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 **L_M**
- Diverse rappresentazioni dei programmi scritti in **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

L_M

M

la macchina hw

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

L_M

M

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

L_M

M_{spider}

cambia l'architettura,
cambia l'interprete,
ma non il linguaggio:

$$L_M = L_{M_{spider}}$$

Un'architettura più sofisticata può interpretare
lo stesso linguaggio macchina

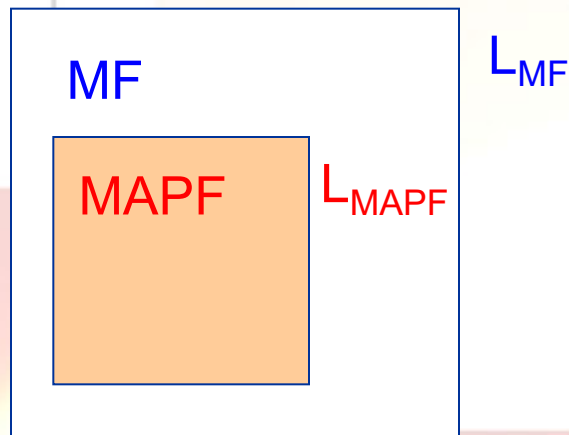
Scatole cinesi

MF

L_{MF}

- 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 μ istruzioni (micro-istruzioni)
 - interpretata da un μ interprete (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 μ istruzioni (micro-istruzioni)
 - interpretata da un μ interprete (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.

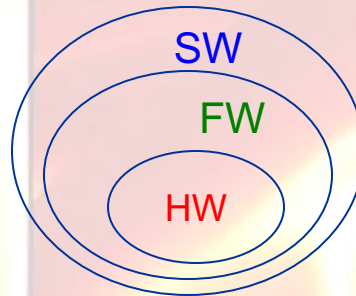
Realizzazione mediante sw: l'interprete

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

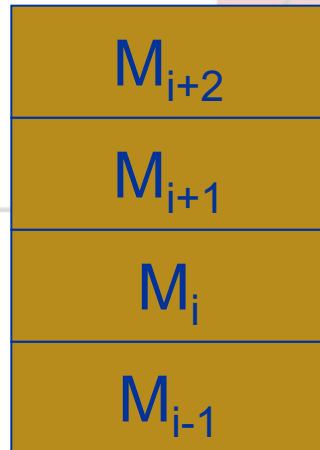
mediante il quale linguaggio realizzare macchine il cui linguaggio sia ancora più sofisticato

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

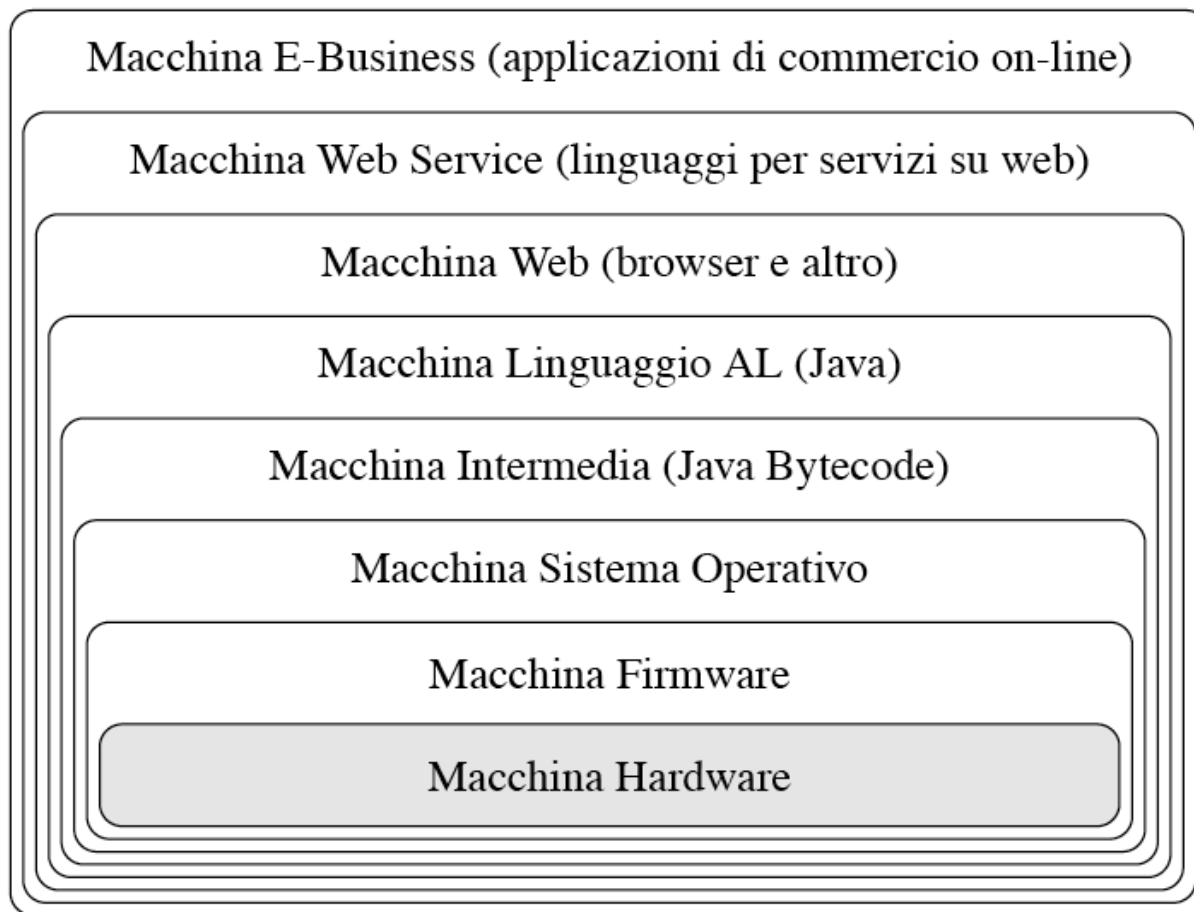
il SO richiede al *file system*
un indirizzo logico per il file

il *file system* richiede alla gestione fisica
l'indice di alcuni blocchi sul disco

.....

esempio: applicazione

Esempio: Web application



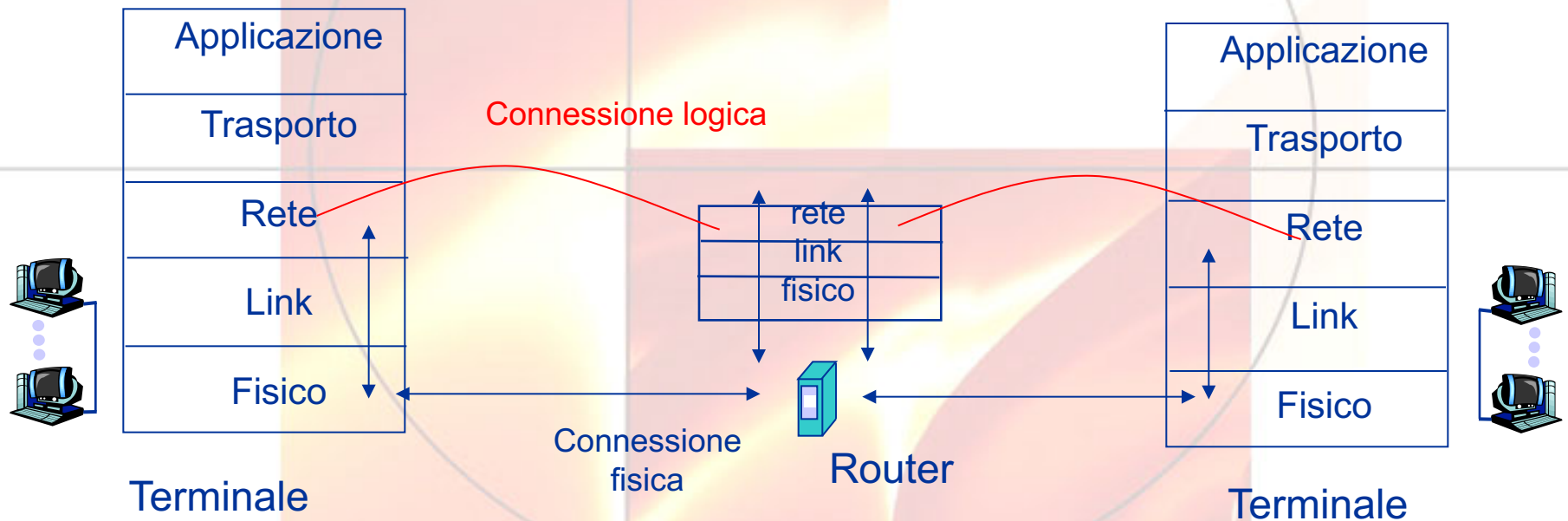
esempio: protocolli



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*:** $f : A \dashrightarrow B$
corrispondenza che *può* essere **non definita** su qualche $a \in A$
- $\mathcal{P}_r^{\mathcal{L}}$ indica un programma 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 $\mathcal{P}_r^{\mathcal{L}}$ $\mathcal{P}^{\mathcal{L}} : \mathcal{D} \rightarrow \mathcal{D}$

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

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;
```

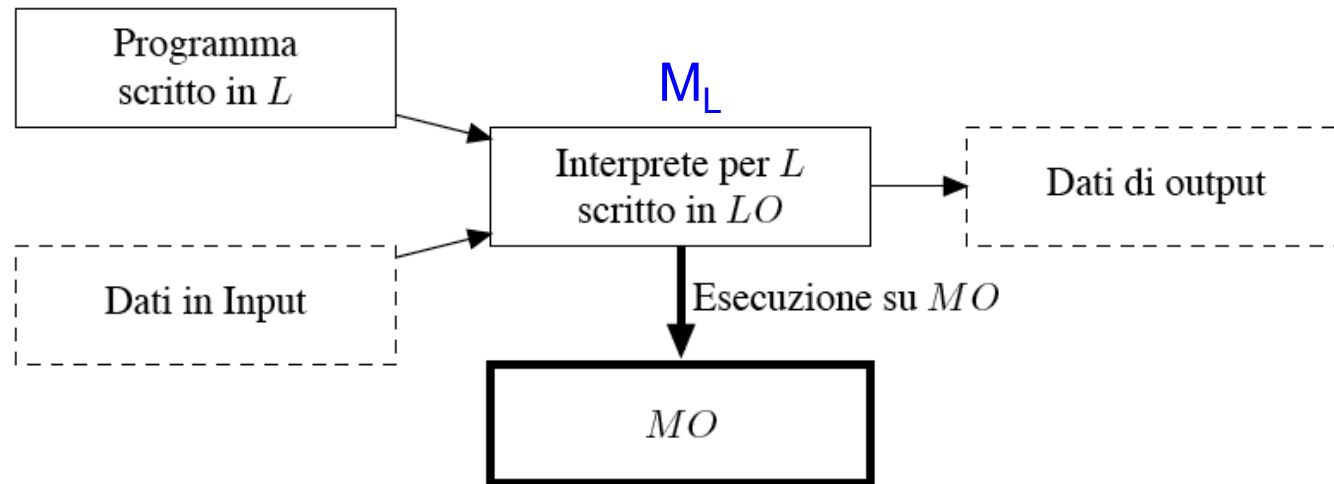
E sua **semantica** come funzione da input ad output

```
 $\mathcal{P}^{\mathcal{L}}$  (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*

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

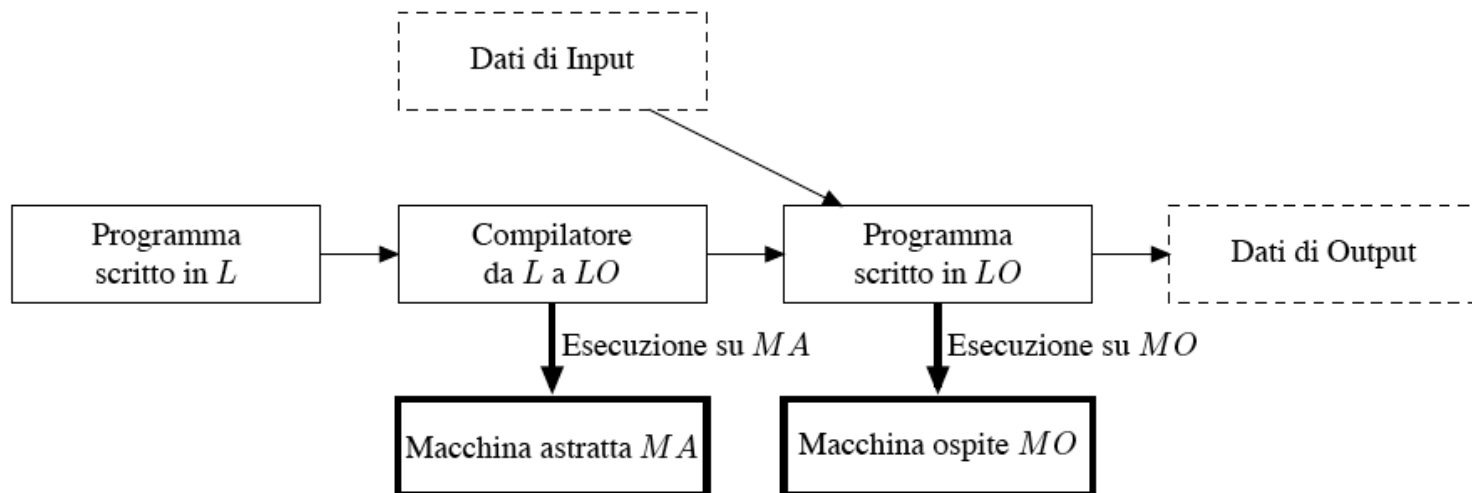
Ovvero l'interprete “**calcola la corretta semantica**” del programma!

Implementazione compilativa pura

- I programmi in \mathcal{L} sono *tradotti* in programmi *equivalenti* in \mathcal{L}_0
- Traduzione effettuata da un (altro) programma

$$C_{\mathcal{L}, \mathcal{L}_0}^{\mathcal{L}_a}$$

il **compilatore** da \mathcal{L} a \mathcal{L}_0 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*

$$C_{\mathcal{L}, \mathcal{L}^o} : \text{Prog}^{\mathcal{L}} \rightarrow \text{Prog}^{\mathcal{L}^o}$$

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

$$C_{\mathcal{L}, \mathcal{L}^o}(\mathcal{P}_r^{\mathcal{L}}) = \mathcal{P}_r^{\mathcal{L}^o} \quad (1.2)$$

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

$$\mathcal{P}^{\mathcal{L}}(\text{Input}) = \mathcal{P}^{\mathcal{L}^o}(\text{Input}). \quad (1.3)$$

Ovvero il compilatore “**preserva la semantica**” del programma: il programma originale $\mathcal{P}_r^{\mathcal{L}}$ e quello tradotto $\mathcal{P}_r^{\mathcal{L}^o}$ calcolano la stessa funzione: $\mathcal{P}^{\mathcal{L}} = \mathcal{P}^{\mathcal{L}^o}$!

Intermezzo

- Cosa indicano le seguenti scritture?
- $I_{\mathcal{L}_1}^{\mathcal{L}_0}$ = 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 $\mathcal{M}_{\mathcal{L}_1}$
- $I_{\mathcal{L}_1}^{\mathcal{L}_0}(\mathcal{P}^{\mathcal{L}_1}, x)$ = risultato del calcolo del programma \mathcal{P} (scritto in \mathcal{L}_1) con x come dato in input
- $I_{\mathcal{L}_1}^{\mathcal{L}_0}(I_{\mathcal{L}_2}^{\mathcal{L}_1}(\mathcal{P}^{\mathcal{L}_2}, x))$ = risultato del calcolo di \mathcal{P} (scritto in \mathcal{L}_2) con x come input
- La macchina ospite è $\mathcal{M}_{\mathcal{L}_0}$, sulla quale è realizzata una macchina intermedia $\mathcal{M}_{\mathcal{L}_1}$, grazie al primo interprete, mentre la macchina astratta $\mathcal{M}_{\mathcal{L}_2}$, grazie al secondo interprete, è in grado di eseguire il programma \mathcal{P} .

Intermezzo (2)

- Cosa indicano le seguenti scritture?
- $C_{\mathcal{L}_1, \mathcal{L}_2}^{\mathcal{L}_0}$ = un compilatore scritto in \mathcal{L}_0 che traduce programmi scritti in \mathcal{L}_1 in equivalenti programmi scritti in \mathcal{L}_2
- $J_{\mathcal{L}_1}^{\mathcal{L}_0}(C_{\mathcal{L}_2, \mathcal{L}_3}^{\mathcal{L}_1}, P^{\mathcal{L}_2}) = P_1^{\mathcal{L}_3}$ cioè l'interprete, eseguito sulla macchina ospite $M_{\mathcal{L}_0}$, realizza la macchina astratta $M_{\mathcal{L}_1}$ ed esegue un compilatore (scritto in \mathcal{L}_1) che traduce il programma P (scritto in \mathcal{L}_2) in un equivalente programma P_1 (scritto in \mathcal{L}_3)

Compilazione o interpretazione ?

- Implementazione **interpretativa** pura:

- scarsa efficienza della macchina M_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** ($l = 1, l \leq n, l = l+1$) **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)

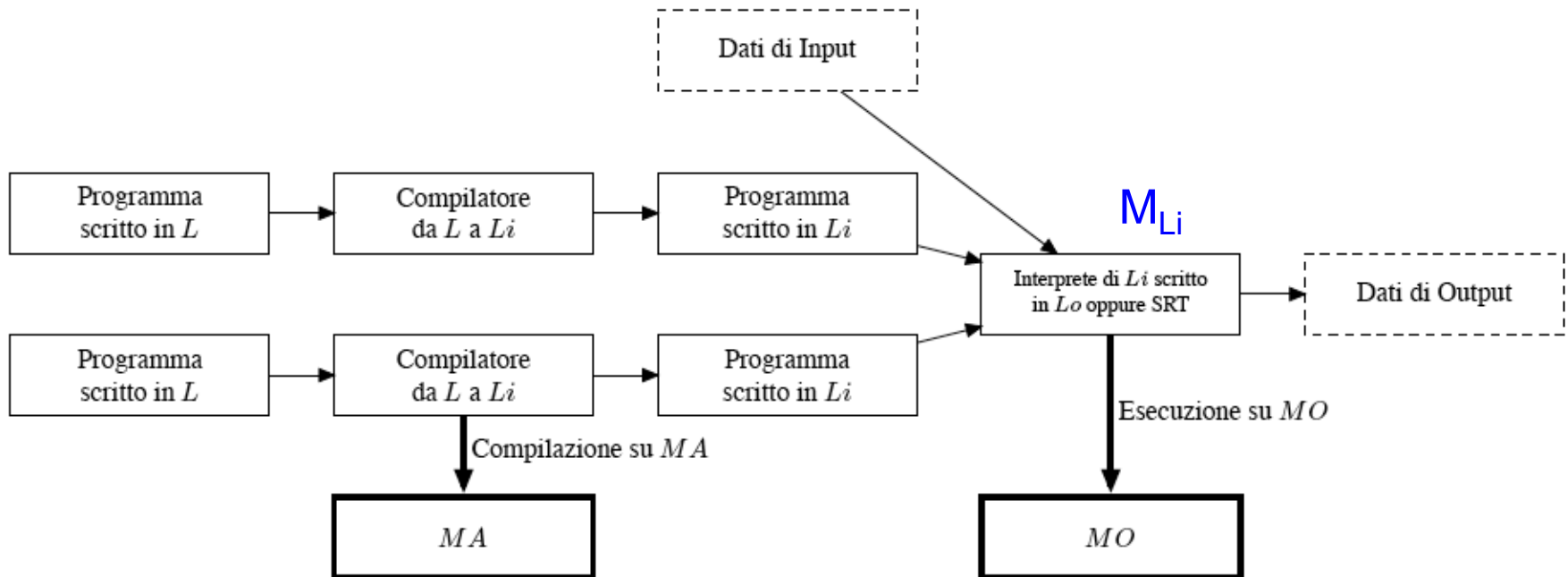
- Implementazione **interpretativa** pura:
 - **buona flessibilità**: permette di interagire con l'esecuzione del programma (facile generare strumenti di debugging "dinamici" step-by-step)
 - **Più facile da realizzare**: il compilatore è di solito molto più complesso
 - **Occupa meno memoria**, perché non viene effettivamente generato codice da memorizzare (questione poco rilevante oggi, 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}_0
 - 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)

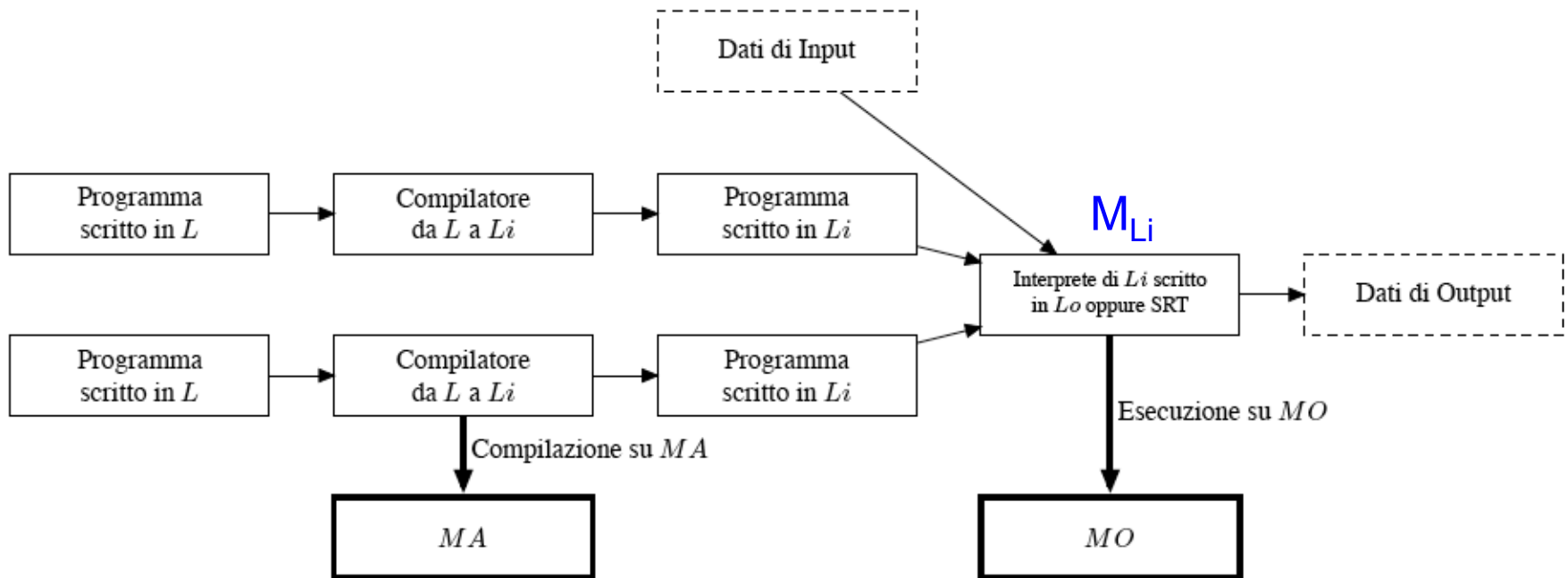
Nel caso reale

- **Entrambe le componenti coesistono**
 - 1) Alcune istruzioni (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_{Li} è sostanzialmente diverso dall'interprete della macchina ospite M_{Lo}



Esempio: Java



Esempio: Java



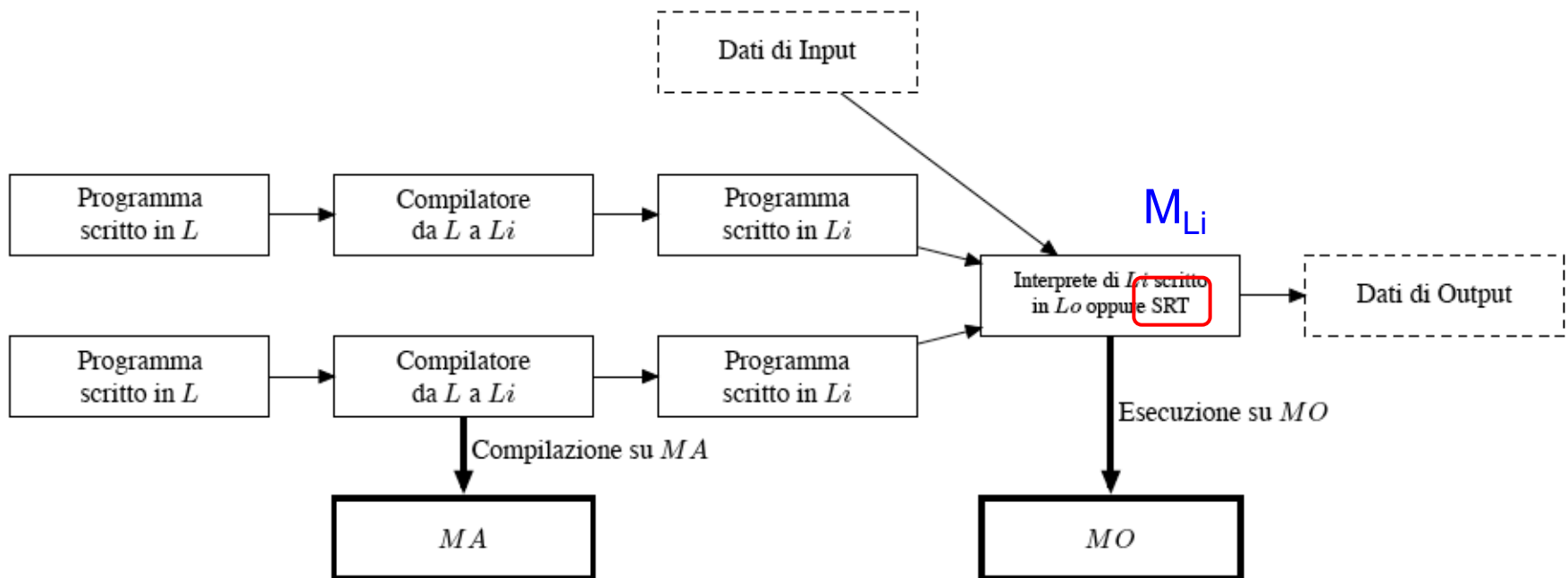
$L_{MA} = \text{Java}$
 $L_{Mi} = \text{Bytecode}$
 $Mi = \text{JVM}$
 $M0 = \text{opportuna macchina che esegue l'interprete della JVM}$

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



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

Esempio: C



$L_{MA} = C$

$L_{Mi} = \text{codice generato da cc}$

$Mi = ?$

~~$M0 \text{ sembra inutile: coincide con } Mi$~~ **NO**

Esempio: C



$$L_{MA} = C$$

$$L_{Mi} = \text{linguaggio generato da cc} +$$

supporto per gestione memoria, I/O ecc.

$$Mi = M0 + \text{interprete per le chiamate del supporto a run-time}$$

$$M0 = \text{macchina ospite}$$

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}_0 ;
 - simulazione (cioè: interpretazione) per gli altri.
- soluzione di tipo **compilativo**
 - privilegia **l'efficienza**
- soluzione di tipo **interpretativo**
 - privilegia **flessibilità e portabilità**

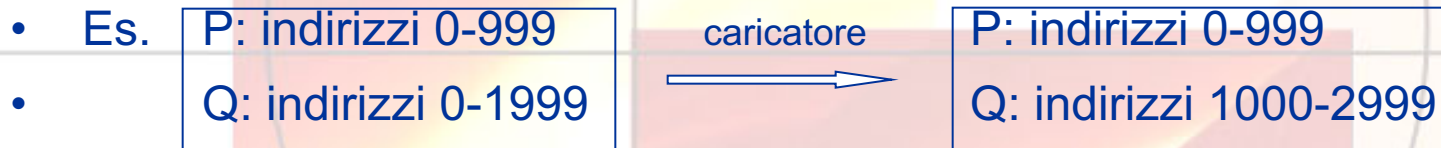
Linguaggi reali

- 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)



- **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 M_o) che simulano I/O e alcune operazioni aritmetiche non fornite da 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}

$C_{\mathcal{L},\mathcal{L}o}^{\mathcal{H}}$

compilatore scritto in \mathcal{H} che traduce \mathcal{L} in $\mathcal{L}o$

$I_{\mathcal{L}}^{\mathcal{H}}$

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: $C_{\text{Pascal,P-Code}}^{\text{Pascal}}$
 - lo stesso compilatore, tradotto in P-code: $C_{\text{Pascal,P-Code}}^{\text{P-Code}}$
 - un interprete per P-code, scritto in Pascal: $J_{\text{P-code}}^{\text{Pascal}}$
- Per aver un'implementazione locale su una specifica Mo:
 - produci (a mano) una traduzione di $J_{\text{P-code}}^{\text{Pascal}}$ nel linguaggio di Mo: $J_{\text{P-code}}^{\mathcal{L}o}$

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

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

- Possiamo fare meglio?

Generazione di compilatori: bootstrapping, 2

- Migliorare l'efficienza, con un compilatore scritto in \mathcal{L}_0

- A mano, facendo hacking di $C_{Pascal, P-Code}^{Pascal}$, produci

$$C_{Pascal, \mathcal{L}_0}^{Pascal}$$

- Adesso abbiamo:

- $C_{Pascal, P-Code}^{Pascal}$
- $C_{Pascal, P-Code}^{P-Code}$
- J_{P-code}^{Pascal}
- $J_{P-code}^{\mathcal{L}_0}$
- $C_{Pascal, \mathcal{L}_0}^{Pascal}$

- **Bootstrapping**

$$J_{P-code}^{\mathcal{L}_0}(C_{Pascal, P-Code}^{P-Code}, C_{Pascal, \mathcal{L}_0}^{Pascal}) = C_{Pascal, \mathcal{L}_0}^{P-Code}$$

$$J_{P-code}^{\mathcal{L}_0}(C_{Pascal, \mathcal{L}_0}^{P-Code}, C_{Pascal, \mathcal{L}_0}^{Pascal}) = C_{Pascal, \mathcal{L}_0}^{\mathcal{L}_0}$$

Esercizi

- Cosa viene calcolato nei seguenti casi?
- $J_{\mathcal{L}_1}^{\mathcal{L}_0}(C_{\mathcal{L}_2, \mathcal{L}_3}^{\mathcal{L}_1}, C_{\mathcal{L}_3, \mathcal{L}_0}^{\mathcal{L}_2}) = ?$
- $J_{\mathcal{L}_1}^{\mathcal{L}_0}(C_{\mathcal{L}_0, \mathcal{L}_2}^{\mathcal{L}_1}, J_{\mathcal{L}_1}^{\mathcal{L}_0}) = ?$
- $J_{\mathcal{L}_1}^{\mathcal{L}_0}(C_{\mathcal{L}_2, \mathcal{L}_0}^{\mathcal{L}_1}, J_{\mathcal{L}_1}^{\mathcal{L}_2}) = ?$
- $J_{\mathcal{L}_1}^{\mathcal{L}_0}(C_{\mathcal{L}_0, \mathcal{L}_1}^{\mathcal{L}_2}, C_{\mathcal{L}_2, \mathcal{L}_1}^{\mathcal{L}_0}) = ?$

Soluzioni

- $J_{\mathcal{L}_1}^{\mathcal{L}_0}(C_{\mathcal{L}_2, \mathcal{L}_3}^{\mathcal{L}_1}, C_{\mathcal{L}_3, \mathcal{L}_0}^{\mathcal{L}_2}) = C_{\mathcal{L}_3, \mathcal{L}_0}^{\mathcal{L}_3}$

- $J_{\mathcal{L}_1}^{\mathcal{L}_0}(C_{\mathcal{L}_0, \mathcal{L}_2}^{\mathcal{L}_1}, J_{\mathcal{L}_1}^{\mathcal{L}_0}) = J_{\mathcal{L}_1}^{\mathcal{L}_2}$

- $J_{\mathcal{L}_1}^{\mathcal{L}_0}(C_{\mathcal{L}_2, \mathcal{L}_0}^{\mathcal{L}_1}, J_{\mathcal{L}_1}^{\mathcal{L}_2}) = J_{\mathcal{L}_1}^{\mathcal{L}_0}$

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

- $J_{\mathcal{L}_1}^{\mathcal{L}_0}(C_{\mathcal{L}_0, \mathcal{L}_1}^{\mathcal{L}_2}, C_{\mathcal{L}_2, \mathcal{L}_1}^{\mathcal{L}_0}) = \textit{errore}$

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