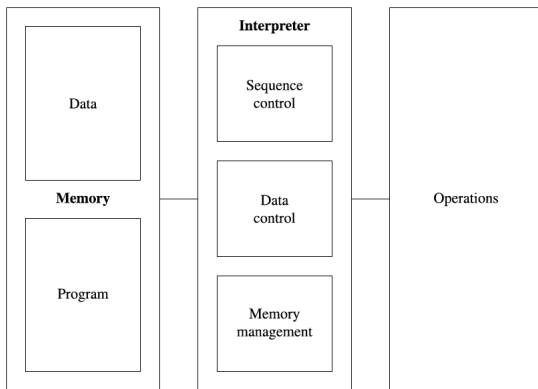


Sommario: 6 marzo, 2018

- Macchina Astratta e l'interprete di Macchina
- High e Low Level Programming Languages
- Implementazione di un Linguaggio
- Macchina Intermedia
- Gerarchia di Macchine
- Trasformazioni di Programmi
- Esercizi

# Struttura di una Abstract Machine /1



- Una AM è l'esecutore di un Linguaggio Eseguitabile (inclusi i L. di Programmazione)
- AM di un Linguaggio di Programmazione ha la struttura in figura ed è un'astrazione per un computer
- Tre Componenti principali: **Memoria**, **Interprete** della macchina, **Operazioni**, in comunicazione
- **Discussione:** Cosa non ci aspetteremmo di trovare se AM fosse l'esecutore di una calcolatrice ?
- **Discussione:** Analisi dei componenti

## Definition (Macchina Astratta)

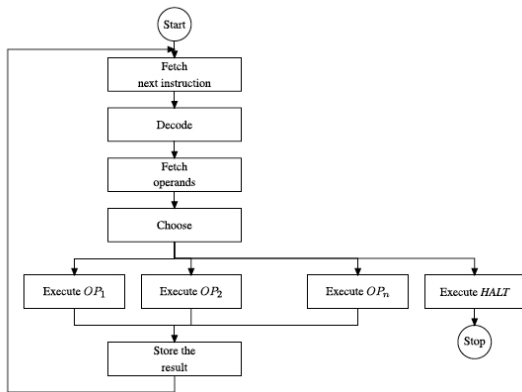
Una macchina astratta per un linguaggio  $\mathcal{L}$ , denotata con  $\mathcal{M}_{\mathcal{L}}$  è un qualsiasi insieme di strutture dati e di algoritmi che permettono di memorizzare ed eseguire i programmi di  $\mathcal{L}$

Quando  $\mathcal{M}_{\mathcal{L}}$  ha componenti e loro comunicazione in Hardware, allora  $\mathcal{M}_{\mathcal{L}}$  è una **Macchina Concreta**.

## Definition (Linguaggio Macchina)

Il linguaggio macchina è il linguaggio  $\mathcal{L}$  di una macchina concreta  $\mathcal{M}_{\mathcal{L}}$

# Ciclo di Esecuzione generico di un Interprete di Macchina



- **Discussione:** Questa struttura potrebbe essere il nucleo di un chip di un processore
- **Discussione:** Commentiamola su istruzioni di 2AC (2 Address Code<sup>1</sup> machine language): ADD R5 R0
  - **Fetch istruzione** coinvolge *controllo di sequenza*, *controllo di memoria* e *memoria* della AM,
  - **Decode** coinvolge *controllo dati* e secondo le modalità di indirizzamento degli operandi...
  - **Fetch operandi** coinvolge ... e consiste in ....

<sup>1</sup>2AC e 3AC sono note classi di Linguaggi Macchina di cui parleremo in slide...

- Quando vogliamo usare un LP  $\mathcal{L}$  dobbiamo realizzare il suo esecutore, ovvero  $\mathcal{M}_{\mathcal{L}}$
- Implementare  $\mathcal{L}$  significa realizzare  $\mathcal{M}_{\mathcal{L}}$
- **Discussione:** Che relazione intercorre tra realizzare  $\mathcal{M}_{\mathcal{L}}$  e la funzione  $\mathcal{U}_{\mathcal{L}}$ , introdotta in slide ...?
- 3 realizzazioni principali:
  - **Hardware:** Costruiamo una macchina ad hoc: Una struttura a K-chip che realizza tutti i componenti (RAM, CASHE, CPU e interfaccia BUS)
  - **Software:** Utilizziamo una macchina esistente  $\mathcal{M}_{\mathcal{L}_0}$  e scriviamo *opportuni programmi* in  $\mathcal{L}_0$  per "implementare"  $\mathcal{U}_{\mathcal{L}}$ .
  - **Firmware:** Adattiamo ad hoc (utilizzando *microcodice* caricato su ROM) una macchina esistente molto simile ad  $\mathcal{M}_{\mathcal{L}}$

# Realizzazione di una Macchina Astratta /2

## Definition (Low-Level Programming Language)

È un LP con basso livello espressivo dove la struttura dei programmi e dei costrutti utilizzati è *condizionata dalla realizzazione* della sua AM.

- La costruzione della AM di un Linguaggio Low-Level nasce contestualmente alla definizione del linguaggio
- La sua costruzione non presenta problemi
- La somiglianza della AM con la struttura in 3 componenti e con lo schema di Interprete di macchina, visti nelle slides, è quasi totale.

## Definition (High-Level Programming Language)

È un LP con alto livello espressivo dove la struttura dei programmi e dei costrutti utilizzati è *condizionata dalle metodologie di programmazione* supportate piuttosto che dalla realizzazione della sua AM.

## Definition (High Level Programming Language)

È un LP con alto livello espressivo dove la struttura dei programmi e dei costrutti utilizzati è *condizionata dalle metodologie di programmazione* supportate piuttosto che dalla realizzazione della sua AM.

Per LP ad alto livello solo la soluzione Software è praticabile.

Questa soluzione avviene in due forme principali. Sia  $\mathcal{M}_{\mathcal{L}}$  la macchina da realizzare, sia  $\mathcal{M}_{\mathcal{L}_0}$  la macchina da utilizzare.

- **Interpretativa Pura:**  $\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_0}$   
Costruzione di un *Interprete* (di linguaggio)  $\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_0}$  di  $\mathcal{L}$  in  $\mathcal{L}_0$ .
- **Compilativa Pura:**  $\mathcal{C}_{\mathcal{L}, \mathcal{L}_0}$   
Costruzione di un *Compilatore* che trasforma programmi  $\mathcal{L}$  in programmi equivalenti scritti in  $\mathcal{L}_0$ .

# Realizzazione di una Macchina Astratta /4

- Sia  $\mathcal{M}_{\mathcal{L}}$  la macchina da realizzare, sia  $\mathcal{M}_{\mathcal{L}_O}$  la macchina da utilizzare.
- Sia  $\mathcal{P}_{\mathcal{L}}$  l'insieme dei programmi di  $\mathcal{L}$ , sia  $\mathcal{P}_{\mathcal{L}_O}$  quello di  $\mathcal{L}_O$
- Siano  $\mathcal{U}_{\mathcal{L}}$  e  $\mathcal{U}_{\mathcal{L}_O}$  le funzioni universali per  $\mathcal{L}$  e  $\mathcal{L}_O$

## Definition (Interprete)

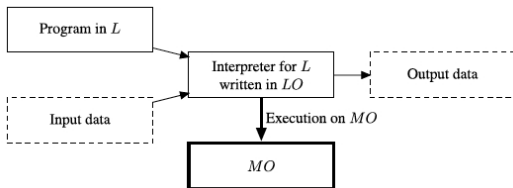
Un interprete per  $\mathcal{L}$  in  $\mathcal{L}_O$ , denotato  $\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_O}$ , è un programma di  $\mathcal{L}_O$ , i.e.  $\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_O} \in \mathcal{P}_{\mathcal{L}_O}$ :

$\forall p \in \mathcal{P}_{\mathcal{L}}, \forall x \in \mathcal{D},$

$$\mathcal{U}_{\mathcal{L}_O}(\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_O})(\langle p, x \rangle) = \mathcal{U}_{\mathcal{L}}(\bar{p})(x)$$

dove:  $\langle \cdot, \cdot \rangle: (\mathcal{P}_{\mathcal{L}} \times \mathcal{D}) \rightarrow \mathcal{D}$  è iniettiva

$\mathcal{D}$  il dominio dei valori delle funzioni calcolabili  $\mathcal{F} \equiv [\mathcal{D} \rightarrow \mathcal{D}]$





# Realizzazione di una Macchina Astratta /5

- Sia  $\mathcal{M}_{\mathcal{L}}$  la macchina da realizzare, sia  $\mathcal{M}_{\mathcal{L}_O}$  la macchina da utilizzare.
- Sia  $\mathcal{P}_{\mathcal{L}}$  l'insieme dei programmi di  $\mathcal{L}$ , sia  $\mathcal{P}_{\mathcal{L}_O}$  quello di  $\mathcal{L}_O$
- Siano  $\mathcal{U}_{\mathcal{L}}$  e  $\mathcal{U}_{\mathcal{L}_O}$  le funzioni universali per  $\mathcal{L}$  e  $\mathcal{L}_O$

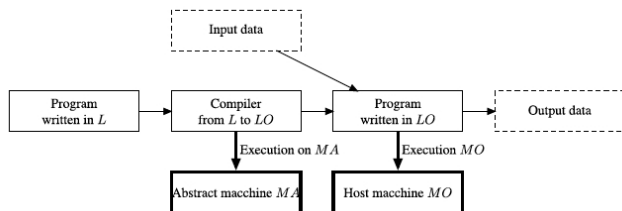
## Definition (Compilatore)

Un compilatore da  $\mathcal{L}$  in  $\mathcal{L}_O$ , denotato  $\mathcal{C}_{\mathcal{L},\mathcal{L}_O}$ , è un programma di un qualche linguaggio  $\mathcal{L}_A$ , i.e.  $\mathcal{C}_{\mathcal{L},\mathcal{L}_O} \in \mathcal{P}_{\mathcal{L}_A}$ :

$$\forall p \in \mathcal{P}_{\mathcal{L}}, \text{ sia } \mathcal{U}_{\mathcal{L}_A}(\overline{\mathcal{C}_{\mathcal{L},\mathcal{L}_O}})(\langle p \rangle) = q \in \mathcal{L}_O$$
$$\text{in } \mathcal{U}_{\mathcal{L}_O}(\bar{q})(x) = \mathcal{U}_{\mathcal{L}}(\bar{p})(x) \quad (\forall x \in \mathcal{D})$$

dove:  $\langle \cdot \rangle: \mathcal{P}_{\mathcal{L}} \rightarrow \mathcal{D}$  è iniettiva

$\mathcal{D}$  il dominio dei valori delle funzioni calcolabili  $\mathcal{F} \equiv [\mathcal{D} \rightarrow \mathcal{D}]$



- I compilatori di un PC non usano Host Machine:  $\mathcal{C}_{\mathcal{L},\mathcal{L}_O} \in \mathcal{P}_{\mathcal{L}_O}$ 
  - $\mathcal{C}_{\mathcal{L},\mathcal{L}_O}$  è scritto in codice  $\mathcal{L}_O$
  - $\mathcal{C}_{\mathcal{L},\mathcal{L}_O}$  è eseguibile su  $\mathcal{M}_{\mathcal{L}_O}$
- Quando usare una Host Machine:  $\mathcal{C}_{\mathcal{L},\mathcal{L}_O} \in \mathcal{P}_{\mathcal{L}_A}$ 
  - La macchina  $\mathcal{M}_{\mathcal{L}_O}$  non ha risorse adeguate per  $\mathcal{C}_{\mathcal{L},\mathcal{L}_O}$ 
    - Bare Machines, Macchine per Servizi, Cloud, Palmari, ...
  - Si vuole proteggere il codice sorgente dell'applicazione  $p \in \mathcal{P}_{\mathcal{L}}$ :
    - Rilasciamo solo il codice oggetto della traduzione di  $p$ .
    - $p$  è un *app* proprietaria e/o non deve essere modificabile
  - Portabilità di codice: Scrivo una volta, Uso ovunque
    - Macchine diverse (HP, Dell, Toshiba, ...) usano uno stesso sistema: ad Esempio UniX,
    - Scriviamo "from scratch" una differente versione UniX nel codice nativo di ogni differente Macchina?
    - Vediamo il caso della Bare Machine (tr. macchina nuda)

# Compilatore su una Host Machine: Bare Machine

- **Bare Machine.**

È dotata del solo linguaggio macchina (Lm) e del suo esecutore.

- Nessuna nozione di processo. Niente File System. Nessun sistema di gestione delle attività. Solo Hardware e un
- Firmware per l'avvio di poche, singole attività di base che includono
- il System Boot (SB) per installare un Sistema Operativo (OS)

- **Come avviene l'installazione.**

- SB carica, a partire da una data locazione, il Boot Program del OS (BOS), un programma in codice nativo, e ne avvia l'esecuzione
- BOS installa il Kernel (K) di OS e ne avvia lo Start
- Kernel contiene strutture dati e programmi per il restante OS (ROS) inclusi
- Compilatore e L. di Programmazione (Los) in cui è scritto ROS
- Stast avvia la compilazione di ROS in codice nativo e lo installato
- La macchina può essere riavviata
- Quando riavviata, il controllo è sotto OS.

- **Come Otteniamo OS in codice nativo.**

- BOS è piccolo e potrebbe essere scritto from scratch
- Kernel è medio/grande, è scritto in Lso, compilato in Lm, via Host, per fasi
- ROS è grande, è scritto in Lso, ed è compilato in Lm direttamente.

- **Nessuna di queste compilazioni può essere rimpiazzata da interpretazione**

# Compilatore e Interprete: Principi Costruttivi

Spesso la realizzazione di un Linguaggio risulta in un kit di strumenti che includono sia il Compilatore che l'Interprete

- La costruzione di C. e di I. hanno alcune parti che possono essere esattamente le stesse:
  - Front-End
  - RTS
- **Front-End.** Parte iniziale (50 %) del codice di C. e di I.:
  - Analizza il programma e ne genera *Abstract Tree*, AT
  - Consiste in A. lessicale, A. sintattica e A. statica e generazione di AT
  - AT è utilizzato da C. e I. come rappresentazione interna del programma.
- **RTS.**

# Compiler/Interprete: Front-End e Back-End

Front-End  
(Compilatore  
Interprete)

SYMBOL TABLE	
1	position    . . .
2	initial    . . .
3	rate      . . .
4	

Back-End  
Compilatore

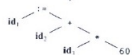
Aho, A.V. et al., Compilers: Principles, Techniques, & Tools, 2 ed., Addison-Wesley, 2007, pag. 7, Fig. 1.7

position := initial + rate \* 60

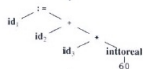
lexical analyzer

$id_1 := id_2 + id_3 * 60$

syntax analyzer



semantic analyzer



intermediate code generator

```
temp1 := intoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

code optimizer

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

code generator

```
MOVf id3, R2
MULF #60.0, R2
MOVf id2, R1
ADDF R2, R1
MOVf R1, id1
```

- La costruzione di C. e di I. hanno alcune parti che possono essere esattamente le stesse
  - Front-End
  - RTS
- **RTS.** Collezione di librerie che definisce strutture e codice in linguaggio oggetto,  $\mathcal{L}_0$ , che implementano meccanismi del linguaggio sorgente,  $\mathcal{L}$ ,
  - Sono meccanismi utilizzati da programmi diversi in modo identico
  - Organizzazione della memoria, Strutture per il controllo dell'esecuzione dei programmi di  $\mathcal{L}$ .

# Confronto tra i due approcci

Dato un L. di Programmazione  $\mathcal{L}$ , confrontiamo una  $\mathcal{M}_{\mathcal{L}}$  basata su interprete con una basata su compilatore e ne discutiamo prestazioni dei programmi *sorgente* ed *oggetto* eseguiti utilizzandole, e costi della loro realizzazione.

- **Costo Tempo.** Compilatore<sup>+</sup>

Impiega meno l'esecuzione di un oggetto o l'interpretazione del suo sorgente?

- **Compilatore:** La decodifica è fatta una volta per tutte le esecuzioni
- **Interprete:** a) La decodifica è ri-fatta ad ogni esecuzione e ancor peggio,  
b) Uno statement può essere decodificato più volte durante una stessa esecuzione

```
P1: for (I = 1, I<=n, I=I+1) C;
```

```
P2:
```

```
  R1 = 1
```

```
  R2 = n
```

```
L1: if R1 > R2 then goto L2
```

```
  translation of C
```

```
  ...
```

```
  R1 = R1 + 1
```

```
  goto L1
```

```
L2: ...
```

- **Costo Memoria.** Interprete<sup>+</sup>

Allochiamo meno memoria durante l'esecuzione di un oggetto o durante l'interpretazione del suo sorgente?

- oggetto cresce linearmente alla dimensione del sorgente (vedi, fig. sopra)

# Confronto tra i due approcci /2

- **Flessibilità e Integrazione.** Interprete<sup>+</sup>

**Q:** Per individuare malfunzionamenti nel sorgente, è più conviene vedere l'esecuzione dell'oggetto o del sorgente?

- L'interprete si interfaccia meglio con strumenti per testing e sviluppo.

- **Sviluppo.** Interprete<sup>+</sup>

**Q:** Durante lo sviluppo del programma è meglio disporre di un interprete o di un compilatore?

- Lo sviluppo di un programma procede attraverso successive fasi di scoperta e rimozione di errori sintattici e di comportamento inatteso.
- Errori sintattici: uso di Front-End di compilatore o interprete è indifferente.
- Errori di Comportamento: indagati meglio sul sorgente che sull'oggetto

- **Uso.** Compilatore<sup>+</sup>

**Q:** Per le esecuzioni di un programma "corretto" e al termine del suo sviluppo conviene usare l'oggetto o il sorgente?

- L'oggetto salta il Front-End ed esegue il codice della macchina sottostante.

- **Semplicità di costruzione.** Interprete<sup>+</sup>

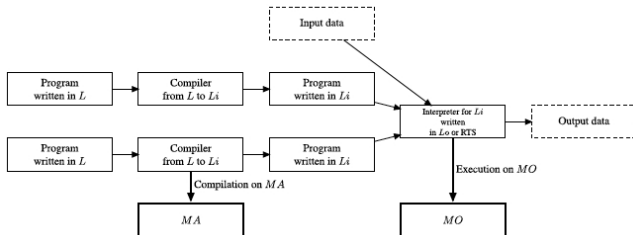
- Compilatore: Deve utilizzare  $\mathcal{U}_{\mathcal{L}}, \mathcal{U}_{\mathcal{L}O}$
- Interprete: Deve esprimere  $\mathcal{U}_{\mathcal{L}}$  con un programma di  $\mathcal{L}O$ .

Adatto per Rapid Prototyping (come vedremo in Laboratorio)



# Realizzazione di una Macchina Astratta: Uso di Macchina Intermedia /6

- Uso di una AM intermedia,  $\mathcal{M}_{\mathcal{L}_i}$ 
  - Compiliamo da  $\mathcal{L}$  in  $\mathcal{L}_i$
  - Interpretiamo  $\mathcal{L}_i$  in  $\mathcal{L}_O$



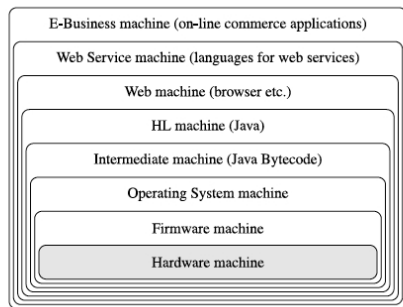
- Interpretativa pura:  $\mathcal{L} = \mathcal{L}_i$
- Compilativa pura:  $\mathcal{L}_i = \mathcal{L}_O$
- Mista:  $\mathcal{L} \neq \mathcal{L}_i \neq \mathcal{L}_O$ 
  - Interpretativa:  $\mathcal{L} > \mathcal{L}_i \gg \mathcal{L}_O$ <sup>2</sup> (Java e Java Virtual Machine, 1995)
  - Compilativa:  $\mathcal{L} \gg \mathcal{L}_i > \mathcal{L}_O$  (Pascal e P-code Machine, 1975)

<sup>2</sup> >> significa molto distante (per livello e/o paradigma), > significa poco distante

# Una gerarchia di Macchine Astratte

- Un computer contiene una gerarchia di AM
- Non tutte sono AM di LP
- Oppure, ogni AM estende la AM inferiore con funzionalità per nuovi "servizi"

Fig. 1.8 A hierarchy of abstract machines



# Esercizi L3

- 1 Si forniscano 3 esempi, in contesti diversi, di macchine astratte.
- 2 In cosa differisce un L. di Programmazione Low-Level da uno High-Level?
- 3 Si elenchino le trasformazioni presenti nel diagramma di una AM, basata su compilazione, di linguaggio  $\mathcal{L}$  in  $\mathcal{L}_o$  utilizzando  $\mathcal{L}_a$ .
- 4 Si consideri il diagramma di una AM, basata su compilazione. Si dica:
  - (a) Quante AM sono presenti nel diagramma;
  - (b) Per ciascuna AM si specifichi Input ed Output
- 5 Si consideri il diagramma di una AM, basata su interpretazione. Si dica:
  - (a) Quante AM sono presenti nel diagramma;
  - (b) Per ciascuna AM si specifichi Input ed Output
- 6 Si consideri il diagramma di una AM, basata su compilazione. Si dica:
  - (a) Se e quando la macchina MA può essere rimossa;
  - (b) Quando rimovibile, si mostri il diagramma ottenuto rimuovendo MA
- 7 Si consideri il diagramma di una AM, basata su realizzazione Mista.
  - (a) Si mostri il diagramma quando il linguaggio intermedio  $\mathcal{L}_i = \mathcal{L}_o$ ;
  - (b) Sia  $\mathcal{L} \neq \mathcal{L}_i$  e  $\mathcal{L}_i \neq \mathcal{L}_o$ . Si dica se e quando MA può essere rimossa;
  - (c) ...

Altri esercizi in:

\* Esercizi3Lezione3

\* Cap. 1 [GM]