

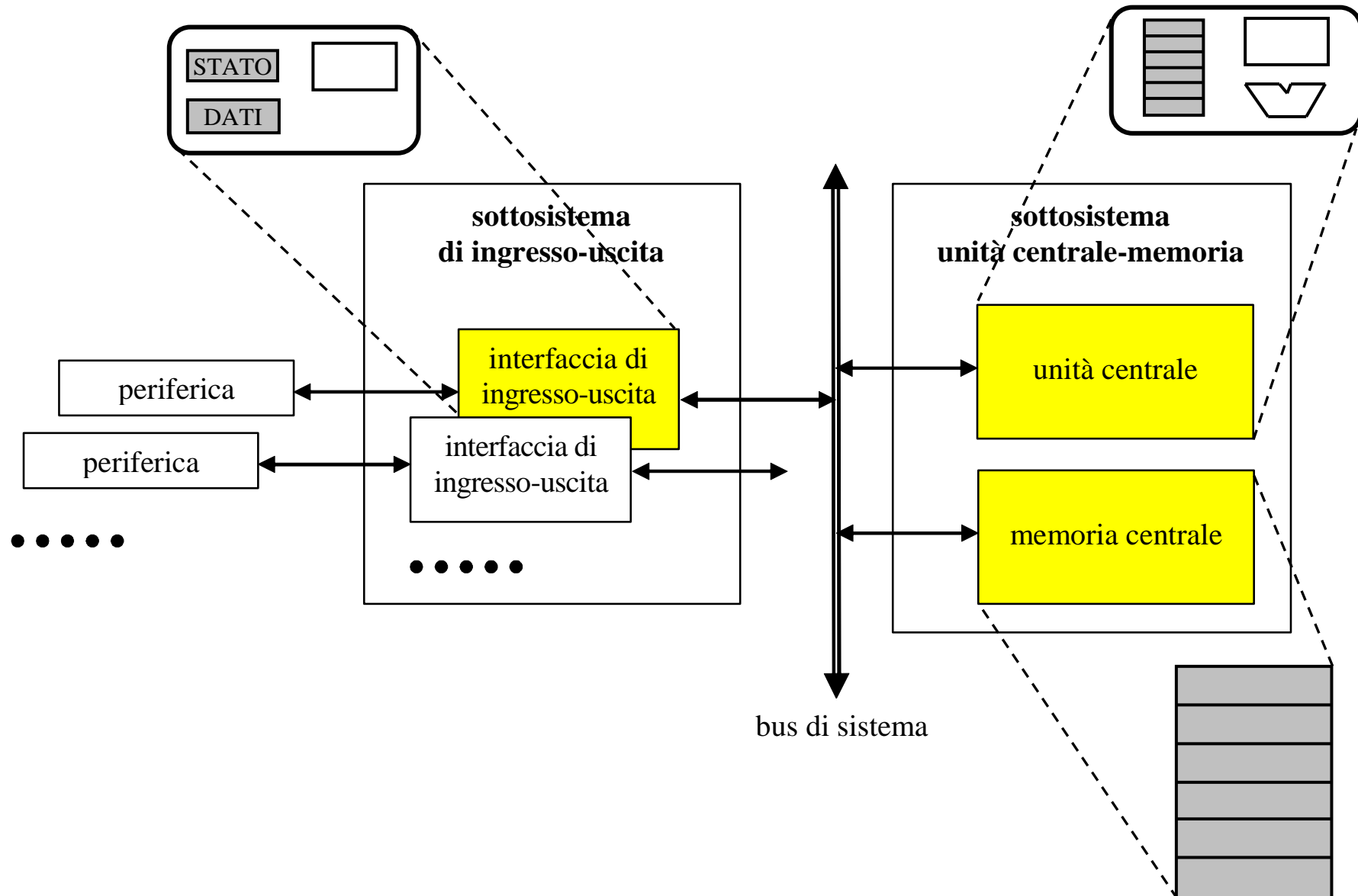
# Linguaggi di alto livello, compilatori e interpreti

*Percorso di Preparazione agli Studi di Ingegneria*

Università degli Studi di Brescia

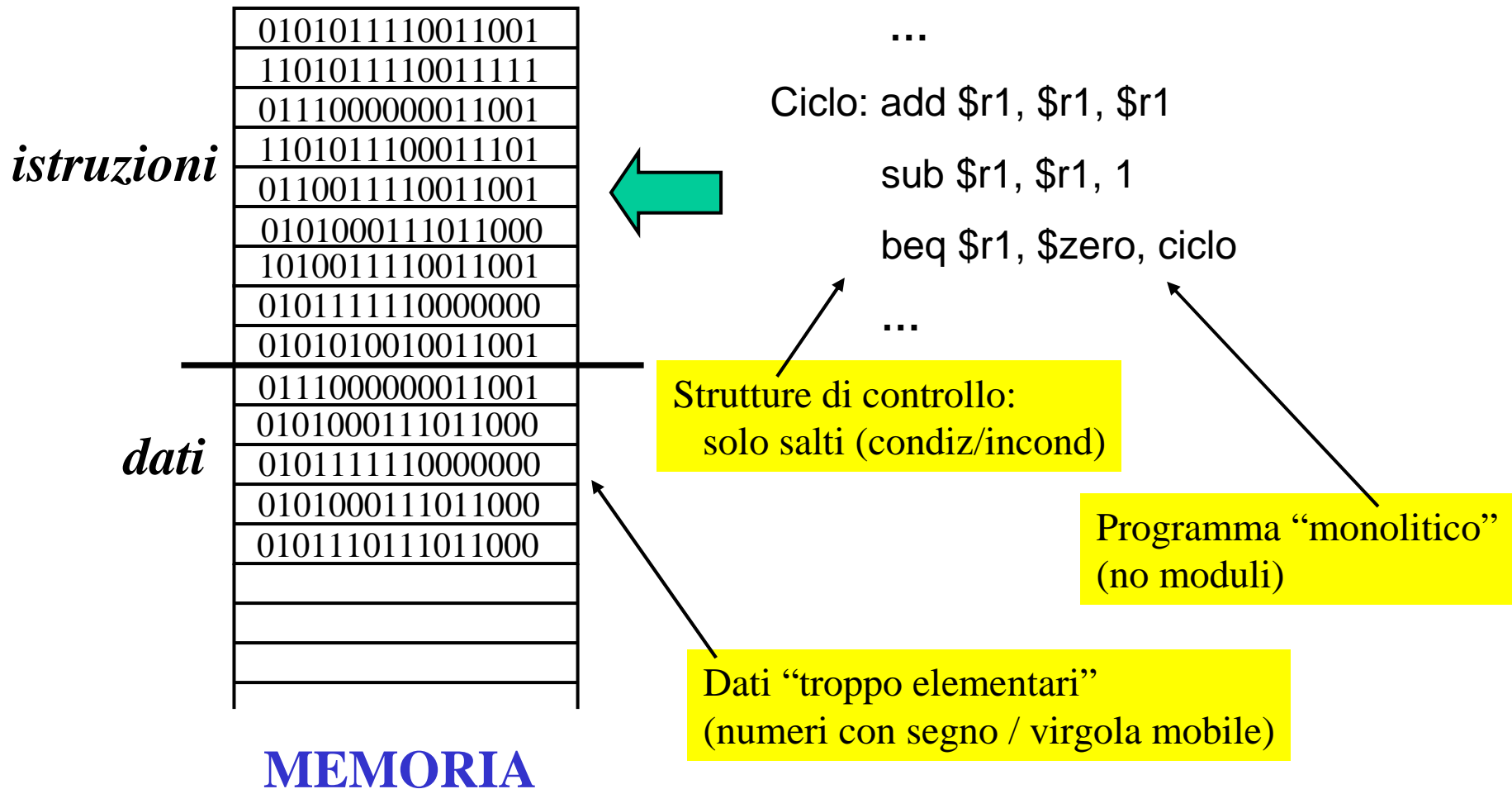
*Docente: Massimiliano Giacomini*

# Il punto della situazione...



# Limiti del linguaggio assembler

**Esempio**: gestione e ordinamento di una rubrica telefonica:  
sarebbe poco praticabile ricorrere al linguaggio assembler



# Linguaggi di programmaz. di alto livello

- Permettono di specificare gli algoritmi con un linguaggio:
  - sempre preciso (termini linguistici di significato preciso e univoco)
  - però più “astratto” rispetto al linguaggio macchina
    - ⇒ ciascuna istruzione di un linguaggio di alto livello corrisponde a più istruzioni macchina
- Vantaggi principali:
  - consentono lo sviluppo di sistemi complessi
  - gli errori commessi sono inferiori
  - il codice è più facile da “mantenere”
  - indipendenza dalla macchina
- Sviluppo storico: da specializzazione a orientamento alla metodologia

# Paradigmi di programmazione

- L'insieme delle caratteristiche che determina il modo di affrontare il progetto e la codifica di un programma, lo “stile di programmazione”

	-	<b>Ad oggetti</b>
<b>Imperativo</b>	<i>Pascal, Cobol, C...</i>	<i>C++, Java, VisualBasic,...</i>
<b>Funzionale</b>	<i>Lisp, APL, Haskell</i>	<i>CLOS</i>
<b>Logico</b>	<i>Prolog</i>	<i>Prolog++</i>

# IL PARADIGMA IMPERATIVO

- Tre tipologie di istruzioni:
    - Istruzioni di ingresso/uscita
    - Istruzioni aritmetico-logiche
      - Ruolo centrale: **assegnamento**
    - Istruzioni di controllo
- } NB: le stesse del linguaggio macchina!
- Ruolo e dimensioni dell'astrazione:
    - Astrazione sui dati
      - **Tipi di dati** predefiniti e definiti dal programmatore (vs. codici binari)
    - Astrazione sul controllo
      - **Condizionali** e **cicli** (vs. istruz di salto condiz/incondiz)
    - Astrazione procedurale (scarso supporto da ling. macchina)

## ASTRAZIONE PROCEDURALE

- Soluzione di problemi complessi (eventualmente lavoro di più persone in modo coordinato): è utile scomporre un programma in moduli più semplici, detti *procedure*
- Facilitano leggibilità, controllo correttezza, produttività, manutenzione del programma
- Definizione di una procedura (*nome + corpo istruzioni*):

*nome\_procedura*(*<parametri formali>*)  
*{ istruzioni... }*

- Parametri formali: sono gli argomenti della procedura (passati dalla procedura o dal programma chiamante):  
variabili utilizzate dalla procedura al suo interno

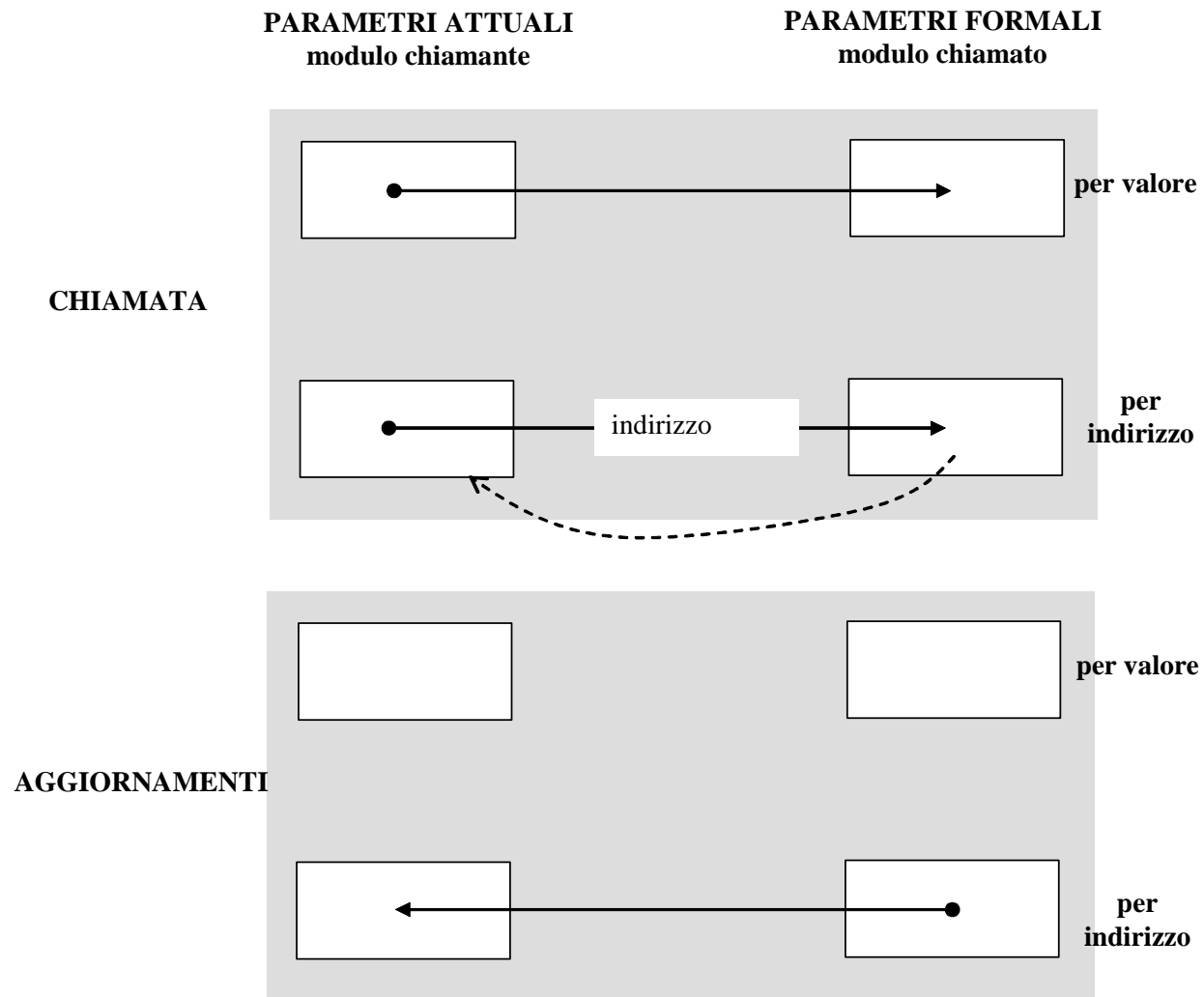
- *Chiamata* di una procedura nel programma:  
*nome\_procedura (<parametri attuali>)*
- Parametri attuali: variabili o espressioni il cui valore è attribuito ai parametri formali (**passaggio dei parametri**)
- I tipi dei parametri attuali devono corrispondere ai parametri formali
- Effetto della chiamata:
  - passaggio dei parametri
  - controllo passato alla procedura, le cui istruzioni vengono eseguite (poi il controllo ritorna all'istruzione successiva alla chiamata)

NB: *è come se il linguaggio si arricchisse di nuove istruzioni...*  
ogni nuova istruzione è la chiamata di una procedura e quindi corrisponde all'esecuzione di un sottoprogramma



- Alcuni linguaggi prevedono due modalità di passaggio dei parametri:
  - **per valore**: i parametri attuali sono copiati nelle variabili della procedura corrispondenti ai suoi parametri formali, la procedura quindi non li modifica
  - **per indirizzo**: la zona di memoria del parametro formale non contiene il valore, ma l'*indirizzo* del parametro attuale; tutte le operazioni sul parametro formale sono in realtà effettuate sul parametro attuale

NB: in C è disponibile solo il passaggio per valore



## Esempio (in C++)

```
void fattoriale(int& fatt, int w){
```

```
fatt=1;
```

```
  while (w>0){
```

```
    fatt = fatt * w;
```

```
    w = w - 1;
```

```
  }
```

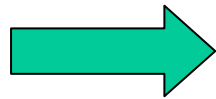
```
}
```

```
int fatt1;
```

```
fattoriale (fatt1, 5);
```

## PROBLEMA:

IL CALCOLATORE NON E' IN GRADO DI COMPRENDERE DIRETTAMENTE UN LINGUAGGIO AD ALTO LIVELLO

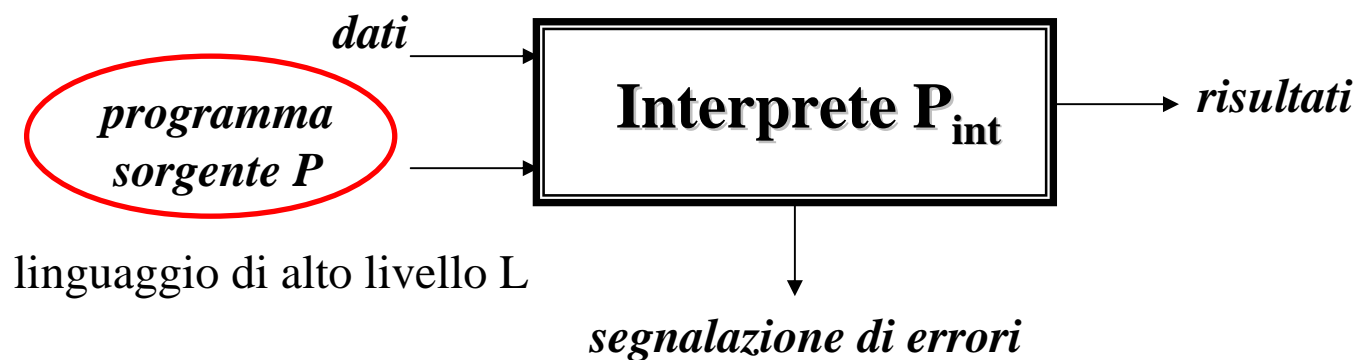
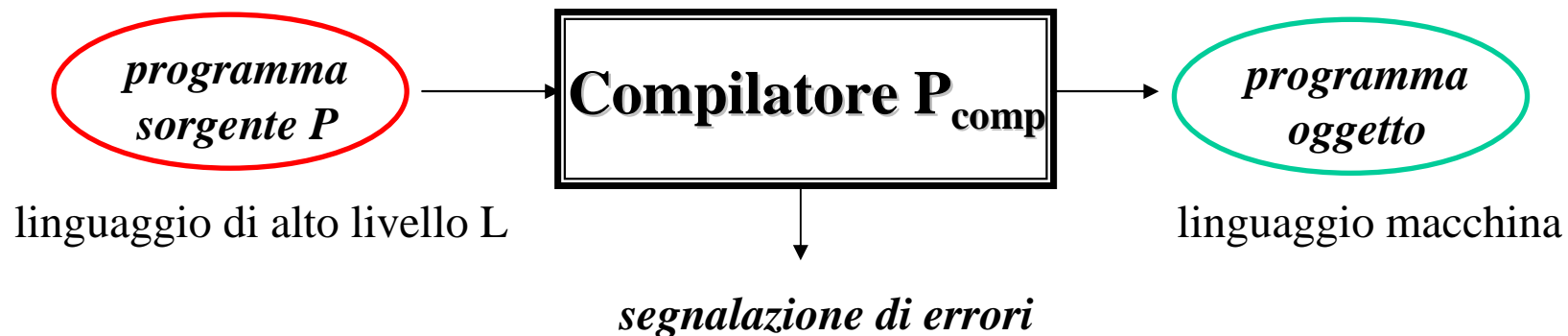


Un programma scritto in linguaggio di alto livello per essere eseguito deve essere **tradotto** in linguaggio macchina

Due modi:

- mediante un *compilatore*
- mediante un *interprete*

# Compilatore e interprete



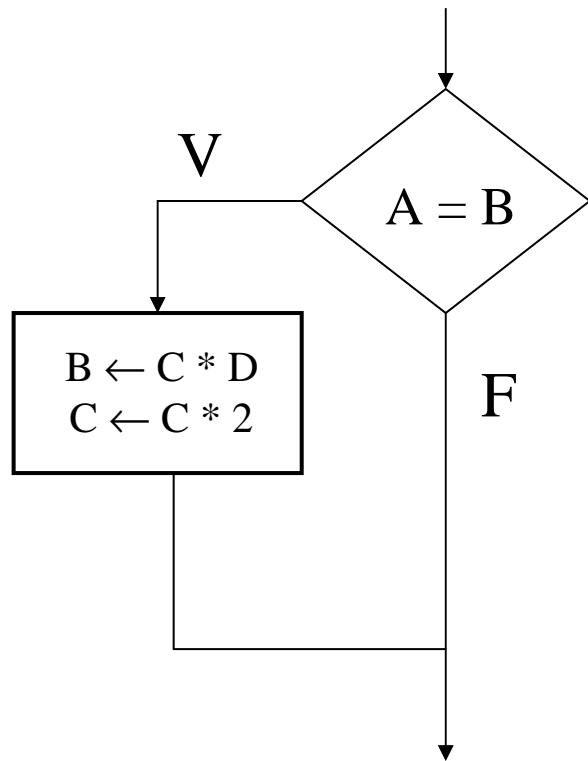
# Eseguire un programma scritto in un linguaggio compilato

- Il programma  $P$  scritto in linguaggio  $L$  viene dato in ingresso a un programma  $P_{Comp}$
- $P_{Comp}$  è il **programma compilatore** del linguaggio  $L$  (ad esempio il programma compilatore del C)
- L'esecuzione da parte di un calcolatore di  $P_{Comp}$  su  $P$  (dove  $P$  è il dato di ingresso) produce  $P_{exe}$  → **PRIMA FASE**
- L'esecuzione da parte di un calcolatore di  $P_{exe}$  su dei dati iniziali produce dei risultati finali → **SECONDA FASE**

# Eseguire un programma scritto in un linguaggio interpretato

- Il programma  $P$  scritto in linguaggio  $L$  viene dato in ingresso a un programma  $P_{Int}$
- $P_{Int}$  è il **programma interprete** del linguaggio  $L$  (ad esempio il programma interprete del Basic)
- **L'esecuzione da parte di un calcolatore di  $P_{Int}$  su  $P$  con i dati in ingresso di  $P$  produce i risultati finali**

# ESEMPIO: C e ASSEMBLER



```
main() /* C */  
{  
  int A, B, C, D;  
  ...  
  if (A == B)  
  {  
    B = C * D;  
    C = C * 2;  
  }  
  ...  
}
```

// A, B, C, D costanti...

```
lw $r1, $r0, A  
lw $r2, $r0, B  
lw $r3, $r0, C  
lw $r4, $r0, D  
...  
bne $r1, $r2, CONT  
mul $r2, $r3, $r4  
sw $r2, $r0, B  
add $r3, $r3, $r3  
sw $r3, $r0, C
```

CONT:

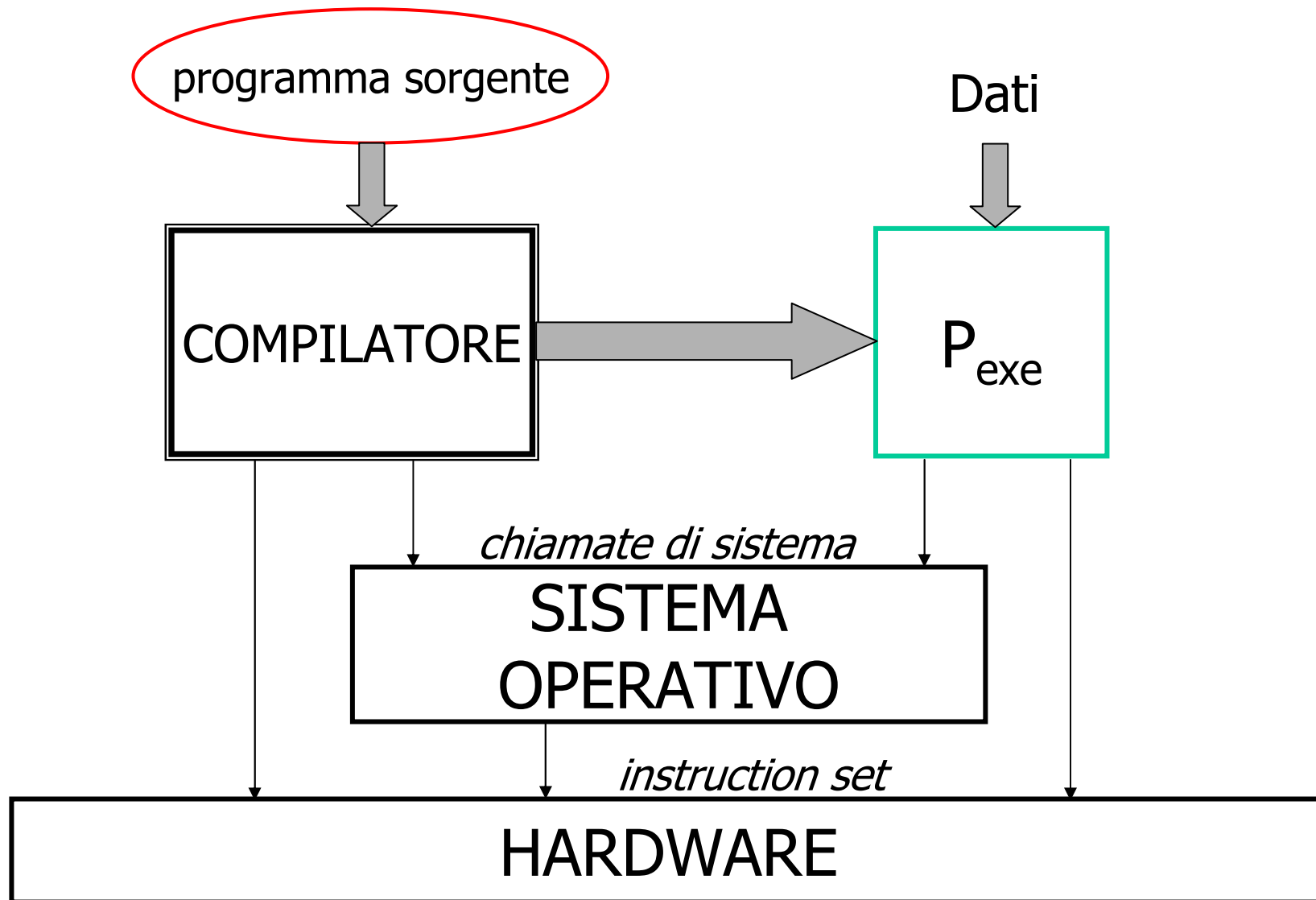
...



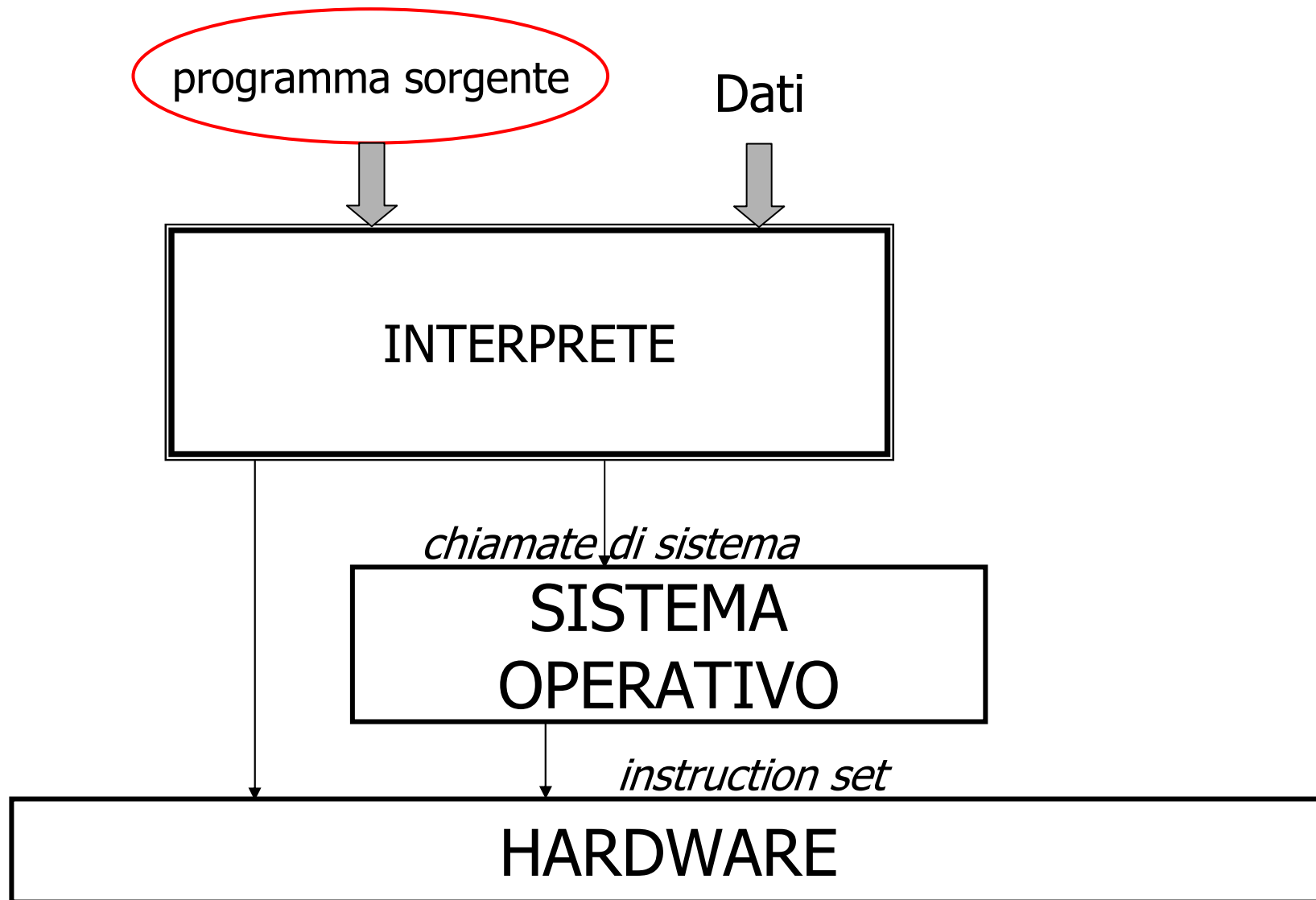
# Domanda

- Quali proprietà fondamentali distinguono un compilatore (o un interprete) da un altro?
  - ✓ Il linguaggio di alto livello
  - 
  - ✓ L'insieme delle istruzioni macchina (ISA) (semplificando, il tipo di processore)
  - ✓ Il sistema operativo

# Compilatore, sistema operativo, hardware



# Interprete, sistema operativo, hardware



# Confronto fra compilatori e interpreti

- *Velocità* di esecuzione: i programmi compilati hanno in genere prestazioni migliori (nella compilazione si possono attuare processi di *ottimizzazione* dell'eseguibile)
- *Messa a punto* del programma: gli interpreti permettono di correggere gli errori non appena vengono scoperti, senza bisogno di ricompilare interamente il programma

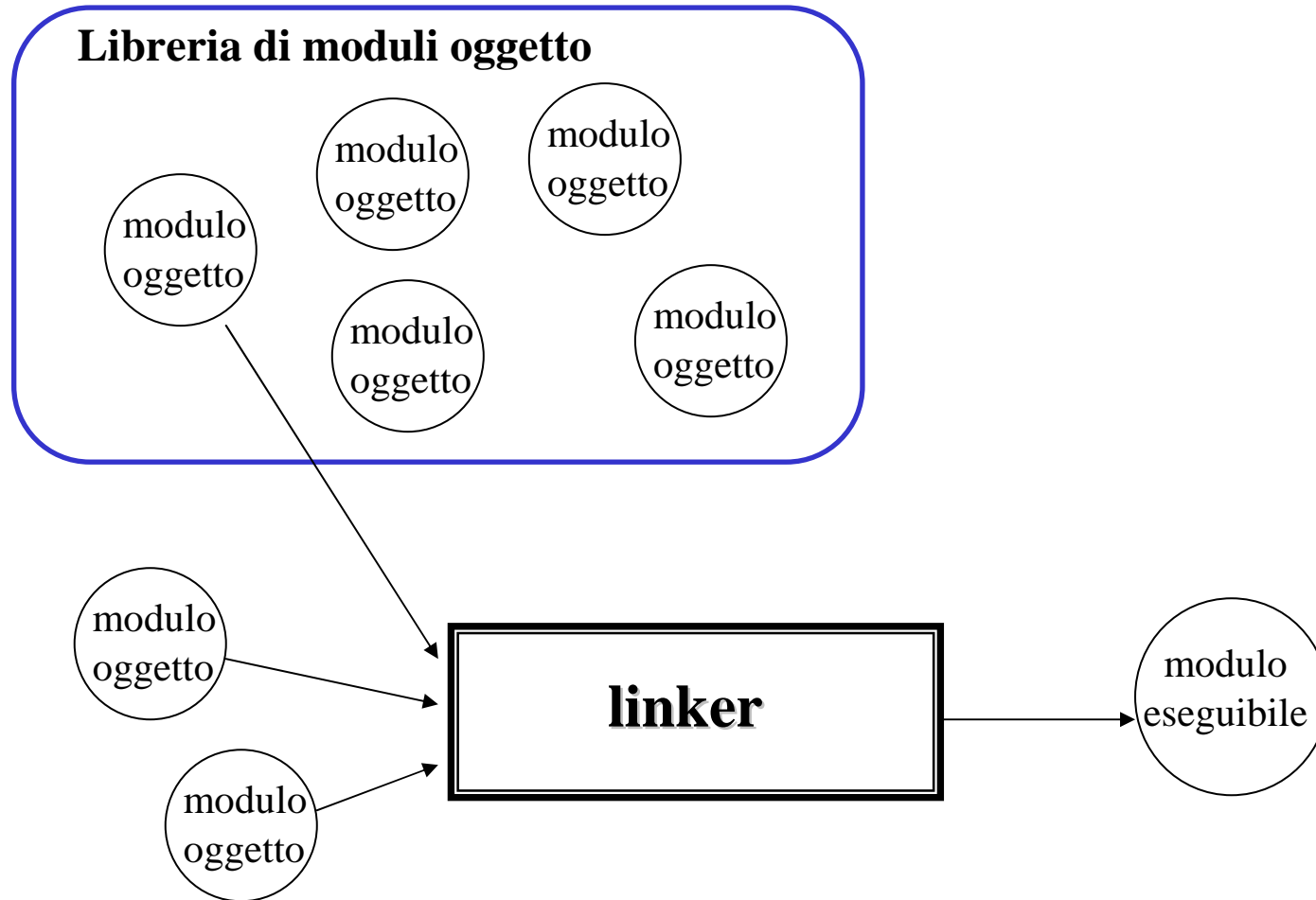
# Confronto fra compilatori e interpreti: portabilità

- Si definisce *portabilità* la possibilità di utilizzare un programma su piattaforme hardware/software diverse da quella dove è stato sviluppato
- Tre modalità:
  - *portabilità del file eseguibile*: richiede calcolatori con hardware e sistema operativo dello stesso tipo per il quale è stato compilato. Semplice ma restrittivo.
  - *portabilità tramite ricompilazione*: richiede l'esistenza di un compilatore e di librerie per la nuova piattaforma e lo svolgimento della ricompilazione. Non fattibile dall'utente medio.
  - *portabilità tramite interpretazione*: richiede l'esistenza di un interprete per la nuova piattaforma. Semplice (purchè l'interprete sia già installato) e non restrittivo.

# Compilatore e linker

- I compilatori consentono tipicamente la compilazione separata di parti di programmi (*moduli*)
- I diversi moduli possono essere progettati, costruiti e messi a punto separatamente, e archiviati in opportune *librerie*
- Nel momento in cui un programma deve essere eseguito, un programma apposito, detto *linker*, si occupa di ritrovare e collegare opportunamente fra loro i moduli oggetto
- Il risultato del linker è un unico modulo, detto *modulo eseguibile*, pronto per il caricamento in memoria e l'esecuzione

# Il ruolo del linker



**Alcuni ambienti di sviluppo includono gli strumenti di creazione, traduzione ed esecuzione dei programmi**

