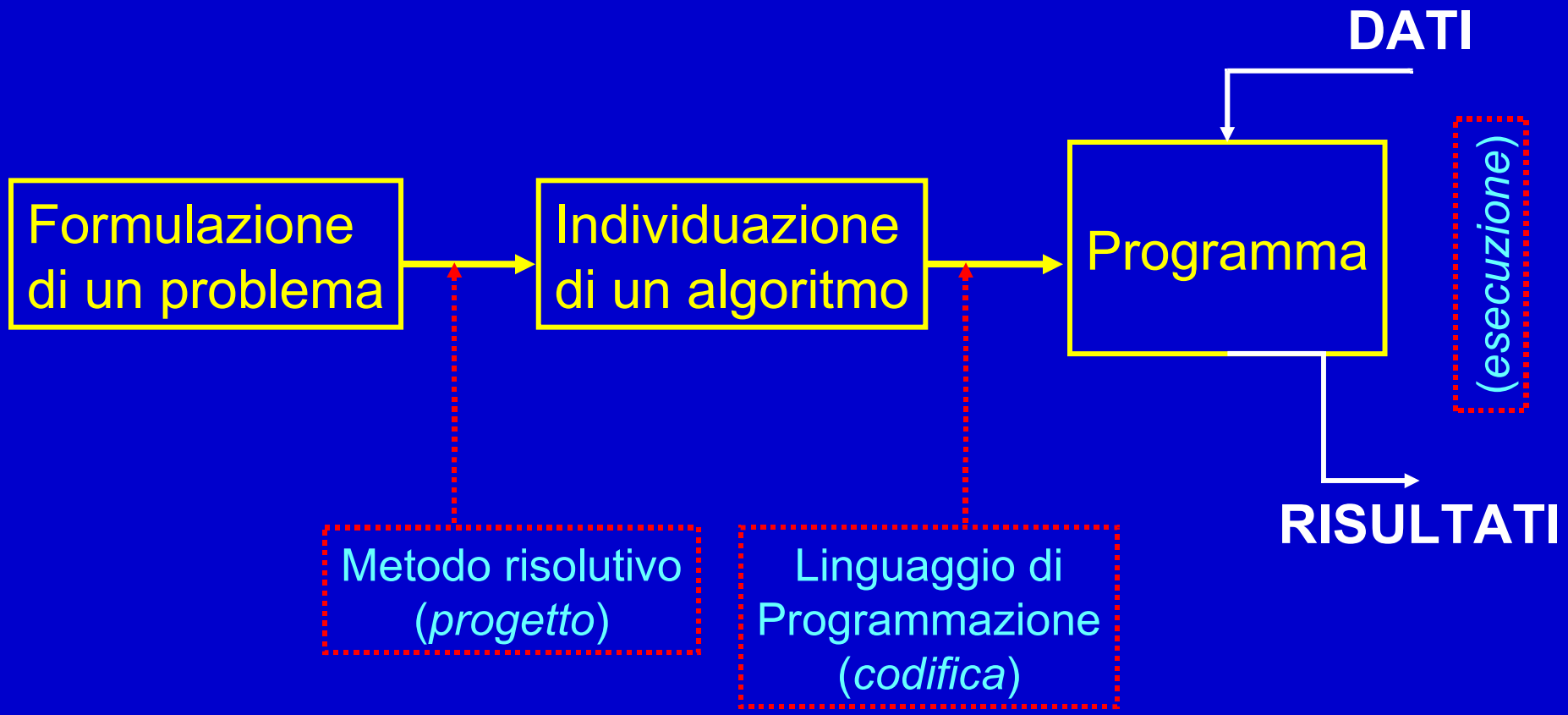
Algoritmi, programmi, traduttori, compilatori, ambienti di sviluppo

Riassumendo…



Esempio 1

Formulazione del problema

Stampare la somma di due numeri dati

Individuazione di un algoritmo

- Leggere il primo numero (p.es., da tastiera)
- Leggere il secondo numero (p.es., da tastiera)
- Effettuare la somma
- Stampare il risultato (p.es., su video)

Caratteristiche di un algoritmo

- **Eseguibilità**: ogni azione deve essere *eseguibile* da parte dell'esecutore dell'algoritmo in un tempo finito
- **Non-ambiguità**: ogni azione deve essere *univocamente interpretabile* dall'esecutore
- **Terminazione**: il numero totale di azioni da eseguire, per ogni insieme di dati di ingresso, deve essere finito

Algoritmo (*cont.*)

Quindi, l'algoritmo deve:

- essere *applicabile a qualsiasi insieme di dati di ingresso* appartenenti al *dominio di definizione* dell'algoritmo
- essere costituito da operazioni appartenenti ad un determinato insieme di operazioni fondamentali
- essere costituito da regole non ambigue, cioè interpretabili in modo univoco qualunque sia l'esecutore (persona o "macchina") che le legge

ALTRE PROPRIETA'

- **Determinismo**
- **Efficienza**
- **Terminazione**

Algoritmi equivalenti

Due algoritmi si dicono **equivalenti** quando:

- hanno lo stesso dominio di ingresso
- hanno lo stesso dominio di uscita
- in corrispondenza degli stessi valori nel dominio di ingresso *producono gli stessi valori* nel dominio di uscita

Due algoritmi equivalenti:

- forniscono lo **stesso risultato**
- ma possono avere **diversa efficienza (numero di passi da effettuare, quantità di memoria occupata)**
- e possono essere **profondamente diversi !**

Costo computazionale 1/3

- L'efficienza in termini di tempo di esecuzione di un algoritmo dipende dal suo **costo computazionale**, ossia dal numero di passi da effettuare per ottenere l'obiettivo per cui è stato definito
  - Spesso chiamata un po' impropriamente anche *complessità* dell'algoritmo
- Però il numero esatto di passi elementari effettuati da un algoritmo per risolvere un dato problema può variare in base ai particolari valori di ingresso, o in funzione di come si è definito l'insieme delle azioni elementari che si possono eseguire

Costo computazionale 2/3

- Pertanto, per misurare il costo di un algoritmo indipendentemente dai precedenti dettagli si utilizza il concetto di **ordine di costo**
  - Dato il numero di elementi *N* su cui un algoritmo lavora (*dimensione del problema*), l'algoritmo ha costo di ordine *O(f(N))* se esistono un valore di *N* ed una costante *c* tali che per **tutti** i valori di *N* maggiori di tale valore il numero di passi effettuati dall'algoritmo non è mai maggiore di *c * f(N)*
- Al di là dei dettagli tecnici (che gli informatici vedranno nel corso di *Algoritmi e strutture dati*), possiamo fare le seguenti importanti considerazioni su alcuni ordini di costo
  - *O(N)*: l'algoritmo effettua un numero di passi proporzionale al numero di elementi su cui lavora

Costo computazionale 3/3

- $O(N^2)$: l'algoritmo effettua un numero di passi pari al quadrato del numero di elementi su cui lavora
- $O(N^i)$: ordine di costo *polinomiale*, su problemi di grosse dimensioni ogni volta che *i* aumenta il tempo di esecuzione diviene enormemente più lungo
- $O(a^N)$: ordine di costo *esponenziale*, al crescere di *N* il tempo di esecuzione dell'algoritmo diviene rapidamente così lungo da renderne impossibile il completamento in tempi ragionevoli

- All'estremo opposto abbiamo:
  - $O(1)$: ordine di costo *costante*, il numero di passi effettuati dall'algoritmo è indipendente dal numero di elementi su cui lavora

Esecutore ed istruzioni

- Un algoritmo è di una qualche utilità se esiste un esecutore in grado di eseguirlo
- Un esecutore può essere istruito in modo molto efficace per eseguire un algoritmo se
  1) Può essere programmato mediante un insieme di istruzioni
  2) Tale insieme di istruzioni ha le seguenti caratteristiche:
     - Le istruzioni sono sufficienti per eseguire i passi dell'algoritmo che si vuol fare eseguire
     - La sintassi e la semantica delle istruzioni sono complete e non ambigue

Automa esecutore

- Un automa esecutore per un dato algoritmo è un'entità con le precedenti caratteristiche ed è quindi in grado di eseguire tale algoritmo

Come realizzare l'automa?

- **M**ediante **congegni meccanici**
  - macchina aritmetica (1649) di Blaise Pascal
  - macchina analitica di Charles Babbage (1792-1871)
- Mediante **un modello matematico**
  - funzionale (Hilbert, (1842-1943), Church, Kleene)
  - operazionale (Turing, 1912-1954)
  - sistemi di riscrittura (Post, Markov,...)

Linguaggio di programmazione

- Un insieme di istruzioni con le precedenti caratteristiche costituisce come sappiamo un linguaggio di programmazione
- In definitiva, quando programmiamo per esempio in linguaggio C/C++ presumiamo la presenza di un esecutore in grado di eseguire appunto un programma in linguaggio C/C++

PER CODIFICARE UN ALGORITMO…

... in un dato linguaggio di programmazione

- Bisogna conoscere la **sintassi** del linguaggio
- Bisogna conoscere la **semantica** del linguaggio
- Bisogna conoscere almeno un **ambiente di programmazione** per tale linguaggio
  - **Editor, compilatore, ...**

Esempio 1 (*cont.*)

Codifica in un programma (*in linguaggio C++*)

```
main()
  {
    int A, B, ris;
    cout<<"Immettere due numeri: ";
    cin>>A;
    cin>>B;
    ris=A+B;
    cout<<"Somma:"<<ris<<endl;
   }
```

Esempio 1 (*cont.*)

E se l'avessimo scritto in linguaggio C?

```
main()
  {
    int A, B, ris;
    printf("Immettere due numeri: ");
    scanf("%d", &A);
    scanf("%d", &B);
    ris=A+B;
    printf("Somma: %d\n", ris);
   }
```

Problema ed elaboratore

- Ma un tipico elaboratore è in grado di eseguire direttamente un programma scritto in linguaggio C/C++?
- Le azioni elementari che un elaboratore può effettuare si chiamano tipicamente **istruzioni macchina**
- Il componente di un elaboratore che si occupa dell'esecuzione di tali istruzioni è tipicamente chiamato (**micro**)**processore**

Linguaggio macchina

- L'insieme delle istruzioni di un processore, nonché la loro sintassi e semantica determinano il **linguaggio macchina** di quel processore
- Le istruzioni di un processore lavorano tipicamente su celle o insiemi di celle di memoria, ed eseguono operazioni elementari come somme, moltiplicazioni, copie di celle di memoria e simili
- In un tipico linguaggio macchina non esistono i concetti di variabile, tipo di dato, programmazione strutturata e così via

Registri

- Un processore è tipicamente dotato di un certo insieme di registri
  - Un registro è una speciale cella di memoria (tipicamente di dimensioni maggiori di un byte) interna al processore
- Uno degli scopi di tali registri è contenere gli operandi delle istruzioni aritmetiche
  - Infatti tali istruzioni tipicamente non possono utilizzare direttamente le celle della memoria principale dell'elaboratore come operandi
  - E' quindi spesso necessario copiare prima gli operandi all'interno dei registri

Linguaggi macchina ed assembly

- I programmi in linguaggio macchina non sono in generale codificati in formato testo
- Ogni istruzione è invece individuata da una data sequenza di byte (ossia da un valore numerico)
- Anche gli operandi su cui lavora sono a loro volta individuati in qualche modo mediante delle sequenze di byte
- In definitiva, un programma in linguaggio macchina non è altro che una sequenza di byte, ossia di numeri

Linguaggio assembly

- Per poter rappresentare in modo leggibile per un essere umano un programma in linguaggio macchina si utilizza tipicamente un linguaggio chiamato *assembly*
- Dato un linguaggio assembly corrispondente ad un dato linguaggio macchina
  - Per ogni istruzione del linguaggio macchina esiste una corrispondente istruzione nel linguaggio assembly
  - In assembly tale istruzione non è più individuata da un numero, ma da una stringa; tipicamente un nome abbreviato che ricorda lo scopo dell'istruzione stessa

Esempio

- Il seguente frammento di codice scritto in un ipotetico linguaggio assembly semplificato (per un ipotetico processore reale) realizza le operazioni necessarie per calcolare il risultato dell'espressione aritmetica 2+3 e memorizzarlo in un registro del processore stesso

```
LDA 3     # memorizza il valore 3
          # nel registro AX
ADD 2     # somma 2 al valore memorizzato nel
          # registro AX
```

Eseguibilità e portabilità

- Il linguaggio macchina è il linguaggio di un processore
- Il linguaggio macchina di un dato processore è direttamente eseguibile da un elaboratore basato su quel processore, senza alcuna intermediazione
- Processori differenti hanno linguaggi macchina differenti
- Pertanto, un programma scritto nel linguaggio macchina di un processore non è eseguibile su di un altro processore!
  - Si dice che non è portabile (*portabile* = può essere eseguito su piattaforme diverse)

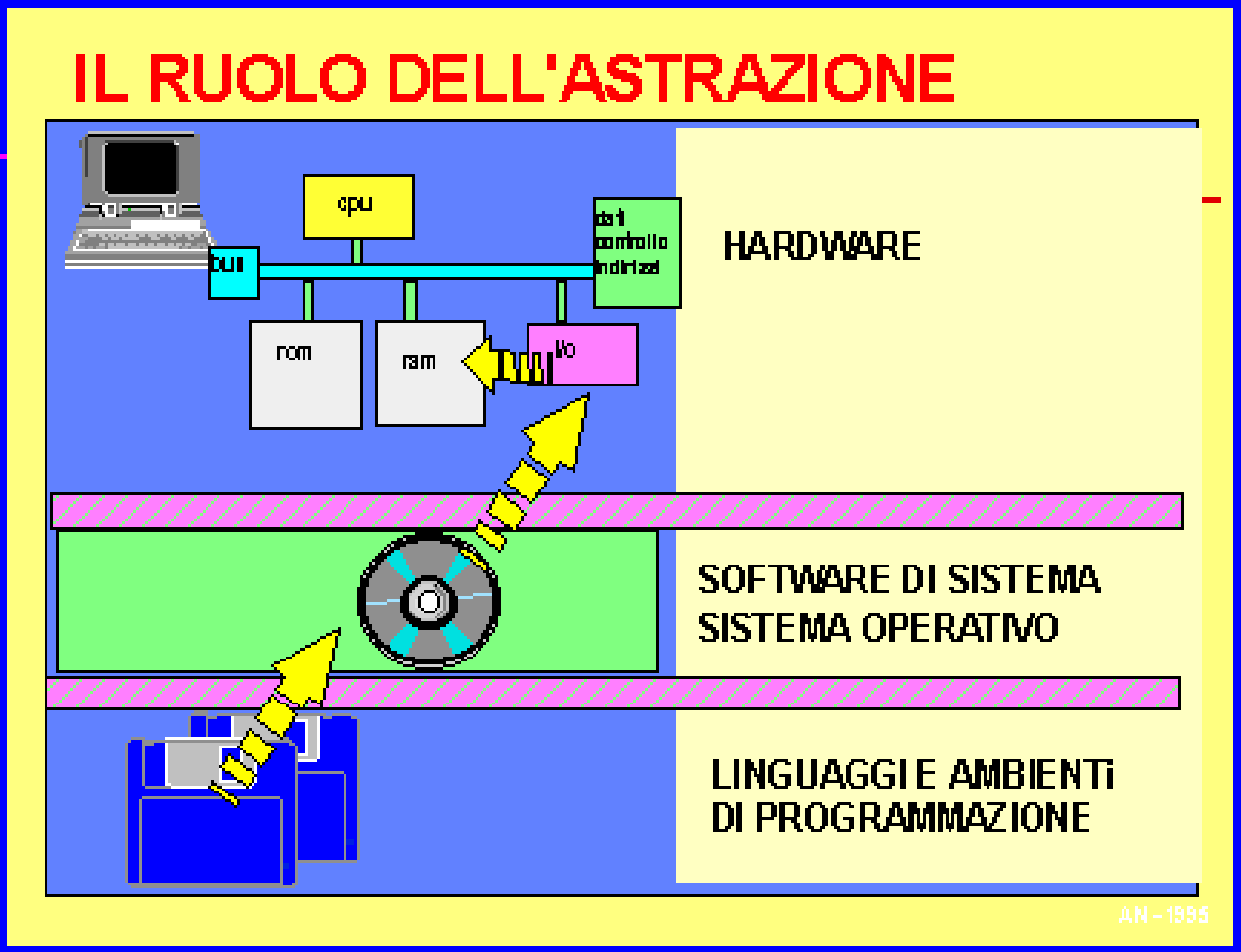
Livello di Astrazione per la Codifica di un Algoritmo

- **Dato l'elaboratore su cui si intende eseguire un certo algoritmo, è una buona idea codificare direttamente nel linguaggio macchina del suo processore tale algoritmo?**
  - SI, se l'algoritmo è tale da poter essere implementato in tale linguaggio con sufficiente semplicità ed in modo tale da ottenere una implementazione molto *efficiente*
    - Inoltre la portabilità deve essere un obiettivo secondario
  - NO, se il processore ha istruzioni di livello *troppo basso* per trovarsi nella precedente situazione
- Nel secondo caso è molto più efficace lavorare ad un più adeguato *livello di ASTRAZIONE*

Astrazione

- Ricordiamo che una definizione formale di processo di astrazione è quella di attività di **aggregazione** di informazioni e dati, e di **sintesi** di modelli concettuali che ne enucleano le proprietà **rilevanti** escludendo i dettagli inessenziali
- Il livello di astrazione di un concetto o di un linguaggio di programmazione è il livello a cui si trova tale concetto o linguaggio di programmazione nell'ambito del processo di astrazione precedentemente definito

Astrazione: come si vede la “macchina”



Linguaggi di alto livello (di astrazione)

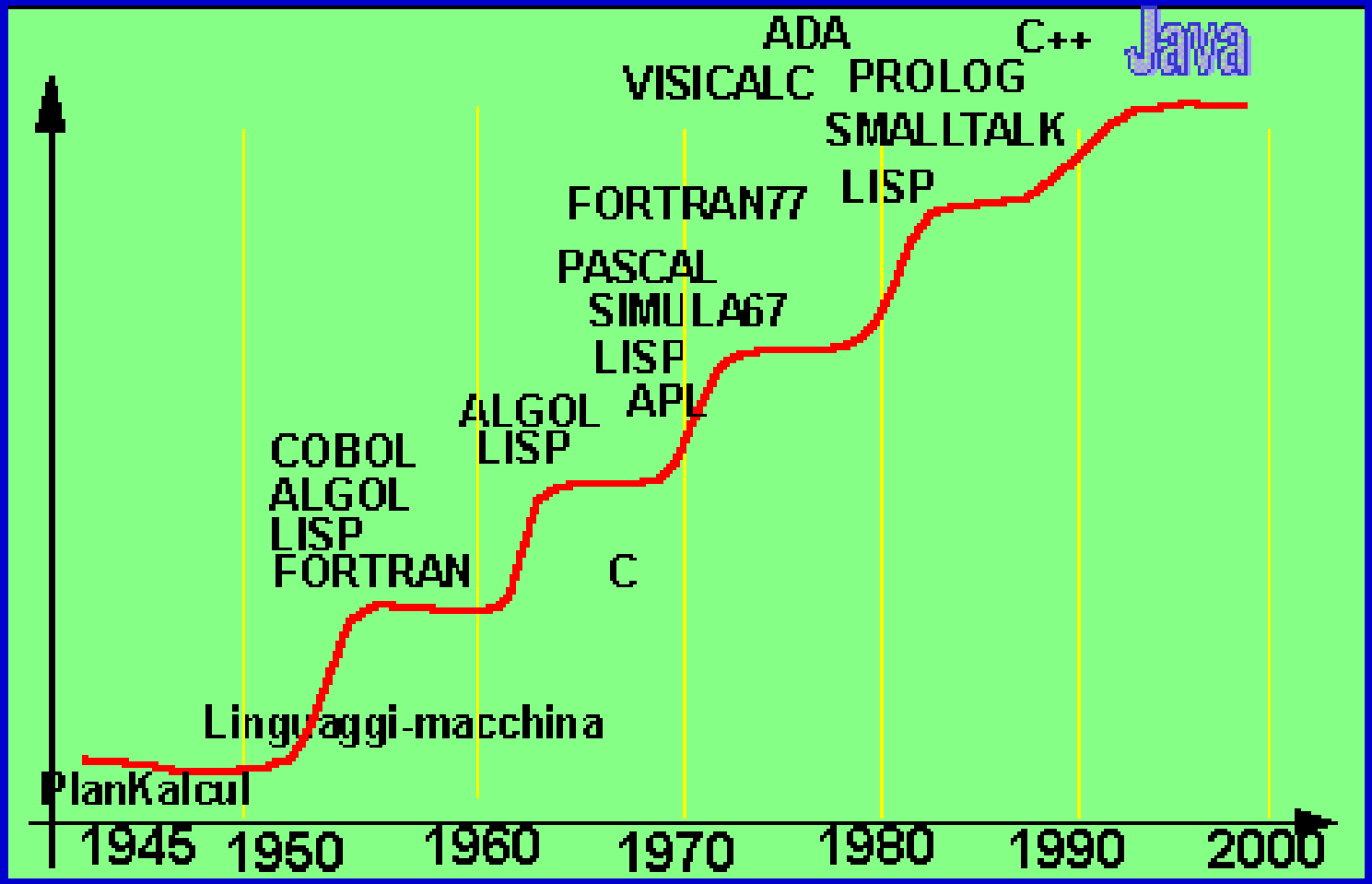
- Si basano su un ipotetico elaboratore dotato di un processore le cui istruzioni non sono quelle di un tipico processore reale, ma quelle del linguaggio stesso
  $\rightarrow$ realizzano una "***macchina virtuale***"
- Supportano concetti ed astrazioni
  Esempio: variabili, tipo di dato, funzioni
- Promuovono metodologie per agevolare lo sviluppo del software da parte del programmatore
  Esempio: programmazione strutturata e/o ad oggetti
- Hanno capacità espressive molto superiori rispetto a quelle del linguaggio macchina
  Esempio: costrutti iterativi
- **Esistono centinaia di linguaggi di programmazione!**
  (anche se pochi sono ampiamente utilizzati)

Linguaggi di alto livello (*cont.*)

Categorie di linguaggi ad alto livello

Ogni categoria determina uno specifico stile di programmazione!

Evoluzione dei linguaggi



Perché esistono tanti linguaggi?

- **Contesto applicativo:**
  - Scientifico: **Fortran**
  - Gestionale: **Cobol**
  - Sistemi Operativi: **C**
  - Applicazioni (non di rete): **C++**
  - Applicazioni di rete: **Java**

Problema

- Praticamente non esistono processori il cui linguaggio macchina raggiunga un livello di astrazione confrontabile con quello di un linguaggio di programmazione di alto livello
- Pertanto, non esistono processori in grado di eseguire più o meno direttamente programmi scritti in linguaggi ad alto livello

Traduzione

- Affinché un programma scritto in un qualsiasi linguaggio di programmazione ad alto livello sia eseguibile su un dato elaboratore, ***occorre tradurlo*** dal linguaggio di programmazione originario al linguaggio macchina del processore montato su tale elaboratore
  - Questa operazione viene normalmente svolta da speciali programmi, detti **traduttori**

Processo di traduzione

Programma originario

```
main()
{ int A;
  ...
  A=A+1;
  if ...
```

Programma tradotto

```
00100101
   ...
11001..
1011100 ...
```

I traduttori convertono il testo dei programmi scritti in un particolare linguaggio di programmazione, ***programmi sorgenti***, nella corrispondente rappresentazione in linguaggio macchina, ***programmi eseguibili***

Tipi di Traduttori

Due categorie di traduttori:

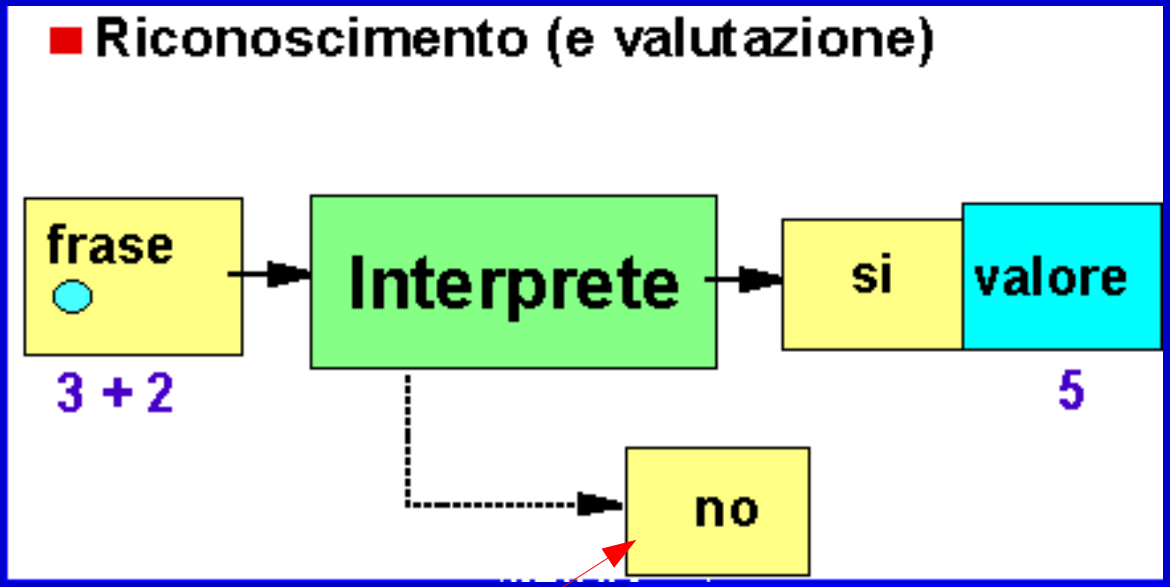
- ***Compilatori*** traducono l'intero programma (senza eseguirlo!) e producono in uscita un il programma convertito in linguaggio macchina (un file eseguibile)
- ***Interpreti*** traducono ed eseguono immediatamente ogni singola istruzione del *programma sorgente* (tipicamente non viene prodotto alcun file)

Quindi,

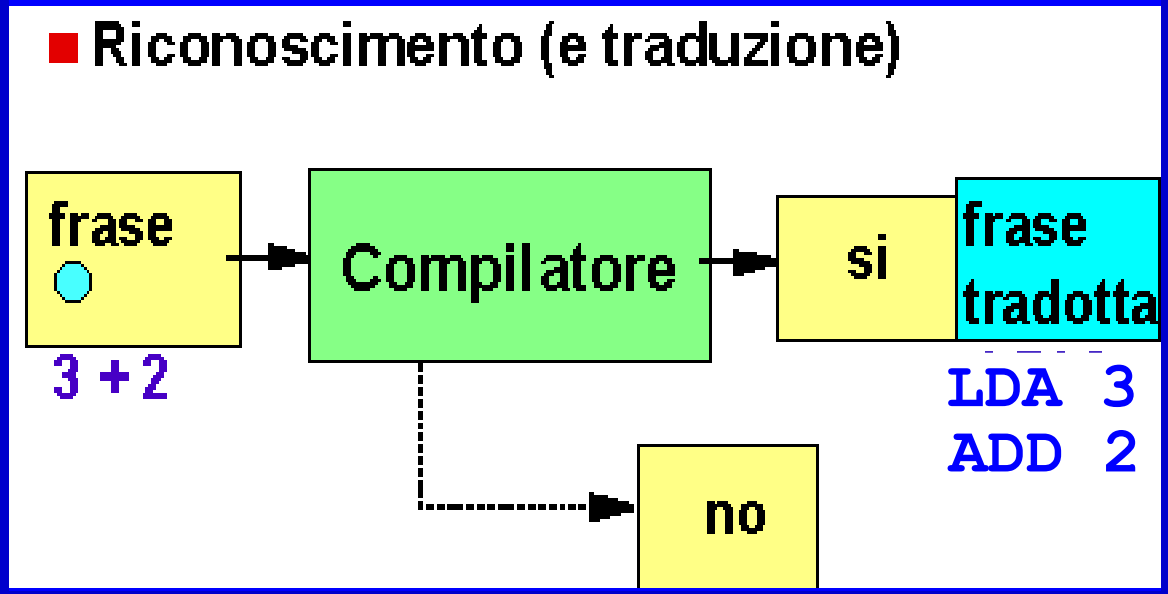
- **Nel caso del compilatore**, lo schema traduzione-esecuzione viene percorso una volta sola prima dell'esecuzione e porta alla creazione del fle che sarà poi eseguito senza altri interventi
- **Nel caso dell'interprete**, lo schema traduzione-esecuzione viene ripetuto tante volte quante sono le istruzioni del programma che saranno eseguite

  Ad ogni attivazione dell'interprete su di una particolare istruzione segue l'esecuzione dell'istruzione stessa.

Compilatore e Interprete



Frase non corretta



Compilatore e Interprete (*cont.*)

- Sebbene in linea di principio un qualsiasi linguaggio possa essere tradotto sia mediante **compilatori** sia mediante **interpreti**, nella pratica si tende verso una differenziazione già a livello di linguaggio:
  - Tipici linguaggi interpretati: **Basic**, **Javascript, Perl**, ...
  - Tipici linguaggi compilati: **C/C++**, **Fortran**, **Pascal**, **ADA**, …

    (*NOTA:* **Java** *costituisce un caso particolare*, anche se si tende a considerarlo interpretato)

Compilatore e Interprete (*cont.*)

- L'esecuzione di un programma *compilato* è tipicamente più veloce dell'esecuzione di un programma *interpretato*
  - *Siccome la traduzione è effettuata una sola volta prima dell'esecuzione, di fatto poi si va ad eseguire direttamente il programma in linguaggio macchina*

- Un programma sorgente da interpretare è tipicamente più **portabile** di un programma da compilare

Librerie, header file e file pre-compilati

- Gli header file contengono spesso solo delle dichiarazioni
- Come sappiamo, le dichiarazioni ci permettono di utilizzare degli oggetti
- Occorre però che poi questi oggetti siano definiti da qualche parte!
- In effetti le funzioni e gli oggetti di una libreria sono definiti in ancora altri file
  - Tali file sono tipicamente pre-compilati nel caso di linguaggi compilati
- In pratica, per fornire una certa libreria vengono forniti tanto gli header file, quanto dei file pre-compilati contenenti il codice macchina delle funzioni e degli oggetti forniti

Collegamento delle librerie 1/2

- Quindi, se in un programma includiamo correttamente gli header file di una certa libreria e ne usiamo le funzioni o gli oggetti, il tutto funziona se
  - Il compilatore **collega** il nostro programma ai file pre-compilati contenenti il codice macchina necessario
- E' per esempio quello che è accaduto automaticamente sempre per la libreria di ingresso/uscita
  - Trattandosi di una libreria utilizzata molto spesso, i compilatori sono tipicamente pre-configurati per collegare il programma ai file pre-compilati di tale libreria

Collegamento delle librerie 2/2

- A volte però il compilatore può non essere già predisposto a collegare il nostro programma ai file pre-compilati di determinate librerie non utilizzate spesso
- In quel caso dobbiamo istruirlo noi, passando ad esempio opzioni aggiuntive
  - Lo abbiamo fatto quando abbiamo aggiunto l'opzione *-lm* per far collegare il nostro programma ai file pre-compilati della libreria matematica

Fasi della compilazione 1/3

- Tipicamente un compilatore ottiene un programma eseguibile da un programma sorgente attraverso varie fasi di compilazione
  1) Preprocessing: il testo del programma sorgente viene modificato in base a delle semplici direttive. Ne abbiamo già visto esempi in C/C++ con le direttive **`#include`** e **`#define`**

Fasi della compilazione 2/3

2) Traduzione (spesso chiamata compilazione, come vedete c'è spesso confusione con i termini): genera un programma in linguaggio macchina a partire dal programma in linguaggio sorgente
   - Il componente di un compilatore che realizza questa fase è tipicamente chiamato **traduttore**
   - Il programma generato nella fase di traduzione non è però tipicamente eseguibile, perché manca fondamentalmente il codice delle funzioni e degli oggetti non definiti nel programma stesso
   - Un programma in linguaggio macchina di questo tipo è tipicamente chiamato **file oggetto**

Fasi della compilazione 3/3

3)Collegamento: si unisce il file oggetto con i file pre-compilati delle librerie (ed eventualmente con altri file oggetto nel caso di programmi sviluppati su più file sorgente)

- Il risultato è il programma (file) **eseguibile**
- Il componente di un compilatore che realizza questa fase è tipicamente chiamato **collegatore** o **linker**

Fasi di sviluppo

- Lo sviluppo di un programma passa attraverso varie fasi
  - Progettazione
  - Scrittura (codifica)
  - Compilazione (per i linguaggi compilati)
  - Esecuzione
  - Collaudo (testing)
  - Ricerca ed eliminazione degli errori (debugging)
  - ...
- Si chiama tipicamente **ambiente di programmazione** (sviluppo) per un dato linguaggio o insieme di linguaggi, l'insieme degli strumenti (*tool*) di supporto alle varie *fasi di sviluppo* dei programmi scritti con tali linguaggi

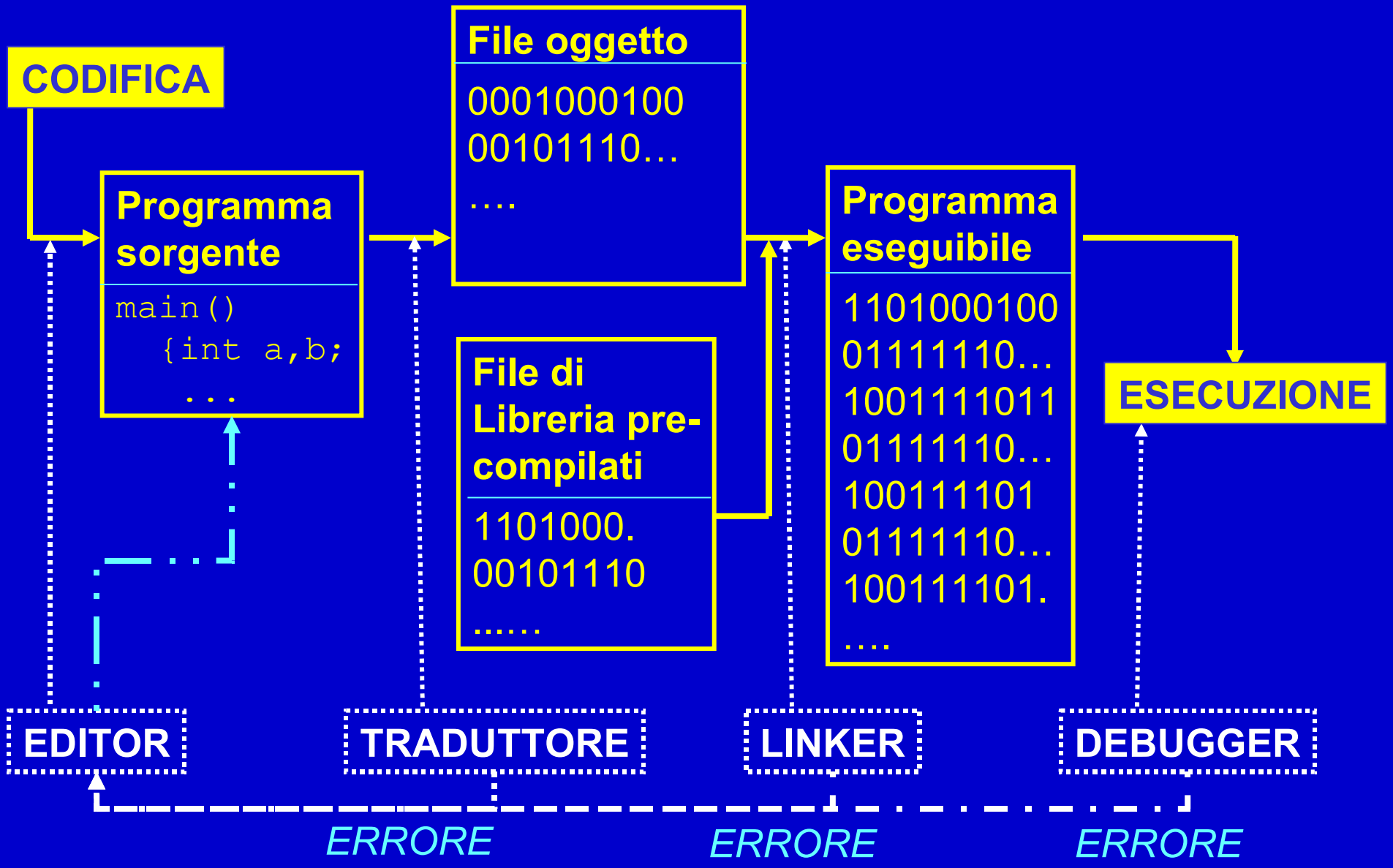
Ambiente di programmazione

- ***Editor*: serve per creare file di testo. In particolare, l'editor consente di scrivere il *programma sorgente*.**
- ***Traduttore** (spesso chiamato anche compilatore)*: opera la traduzione di un programma sorgente scritto in un linguaggio ad alto livello in un *programma oggetto*.**
- ***Linker*** ("*collegatore*"): nel caso in cui la costruzione del programma eseguibile richieda l'unione di più moduli (compilati separatamente), provvede a collegarli per formare un unico file *eseguibile.*
  - Spesso traduttore e linker (e pre-processore) sono componenti di un dato ***compilatore***

- ***Interprete*: traduce ed esegue direttamente ciascuna istruzione del *programma sorgente*, istruzione per istruzione.**
  È alternativo al compilatore.

- ***Debugger***: consente di eseguire passo-passo un programma, controllando via via quel che succede, al fine di scoprire ed eliminare errori non rilevati in fase di compilazione.

Sviluppo ed esecuzione di un programma



Tipi di ambienti di sviluppo 1/2

- Si possono considerare fondamentalmente due tipologie di ambienti di sviluppo
  - Ambienti dati dalla somma di componenti più o meno indipendenti
    - I sistemi operativi UNIX costituiscono spesso dei veri e propri ambienti di sviluppo di questo genere
      - sono dei sistemi in cui è possibile avere molta scelta per i singoli strumenti di sviluppo

Tipi di ambienti di sviluppo 2/2

- Ambienti di sviluppo integrati (Integrated Development Environment, IDE)
  - Ambienti in cui i singoli strumenti sono integrati gli uni con gli altri e si dispone di un'unica interfaccia per gestire tutte le fasi (editing, compilazione, esecuzione, debugging, ...)

Confronto tra i tipi di ambiente

- Il vantaggio principale degli IDE è probabilmente la praticità e semplicità d'uso
  - Uso di una sola interfaccia (tipicamente grafica)
  - Possibilità di salvare, compilare ed eseguire premendo un solo bottone
  - Posizionamento automatico nei punti del programma in cui si trovano gli errori
- Vantaggi dei sistemi non IDE
  - Si distinguono meglio le varie fasi dello sviluppo (utilità fondamentalmente didattica)
  - Si ha maggiore indipendenza da uno specifico strumento (si può usare/cambiare lo strumento preferito per ciascuna fase dello sviluppo)