Linguaggi di Programmazione e Compilatori
Introduzione al corso

Doc. Maria Rita Di Berardini

Dipartimento di Matematica e Informatica
Università di Camerino

Informazioni Generali

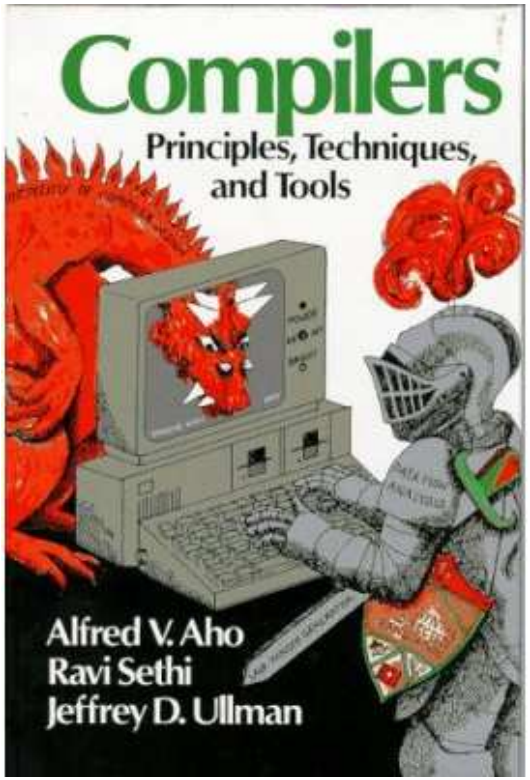
- Ricevimento: Mercoledì ore 15:00 – 17:00
- Email: mariarita.diberardini@unicam.it
- Ufficio: Polo Informatico, primo piano

Programma

- Introduzione: linguaggi di programmazione, paradigmi, storia
- Processo di traduzione: compilazione ed interpretazione
- Struttura generale di un compilatore: fasi di compilazione, primi concetti
- Analisi Lessicale: riconoscimento di tokens, espressioni regolari, automi, algoritmi
- Analisi Sintattica: grammatiche libere, analisi top-down, analisi bottom-up, tabelle LR(1), LL(1), LALR(1), SLR(1)
- Analisi statiche tramite attributi e regole della grammatica: type checking, ...
- Generazione del codice a triple (tipo assembly) – cenni
- Cenni sull'ottimizzazione del codice

Materiale Didattico

- Dispense, lucidi ed esercizi messi a disposizione dal docente
- Testo di riferimento: "**Compilers, principles, techniques and tools**" Alfred V.Aho,Ravi Sethi, Jeffrey D. Ullman. Addison-WesleyPub



Esame

- Prova scritta e possibilità di una prova orale integrativa
- Appelli 8: 2 giugno/luglio, 2 settembre/ottobre, 2 dicembre/gennaio, 2 marzo/aprile
- Per i non frequentanti: materiale (dispense, lucidi ed esercizi) + ricevimenti dedicati (da concordare con il docente)

Obiettivi Didattici

- Un pò di storia

Obiettivi Didattici

- Un pò di storia
- Classificazione in base al livello di astrazione

Obiettivi Didattici

- Un pò di storia
- Classificazione in base al livello di astrazione
- Paradigmi di Programmazione

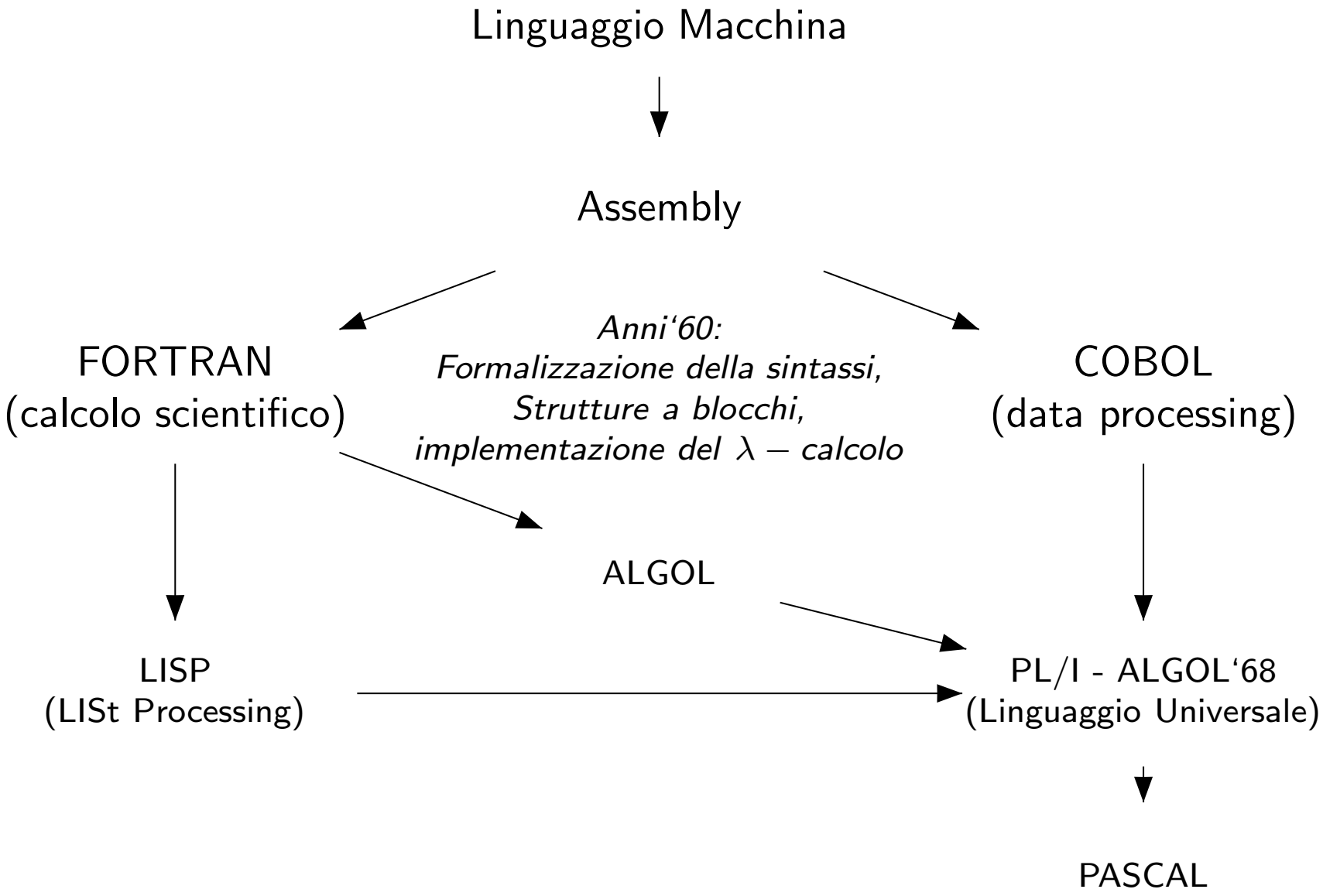
Obiettivi Didattici

- Un pò di storia
- Classificazione in base al livello di astrazione
- Paradigmi di Programmazione
- Processo di traduzione: compilazione ed interpretazione

Part I

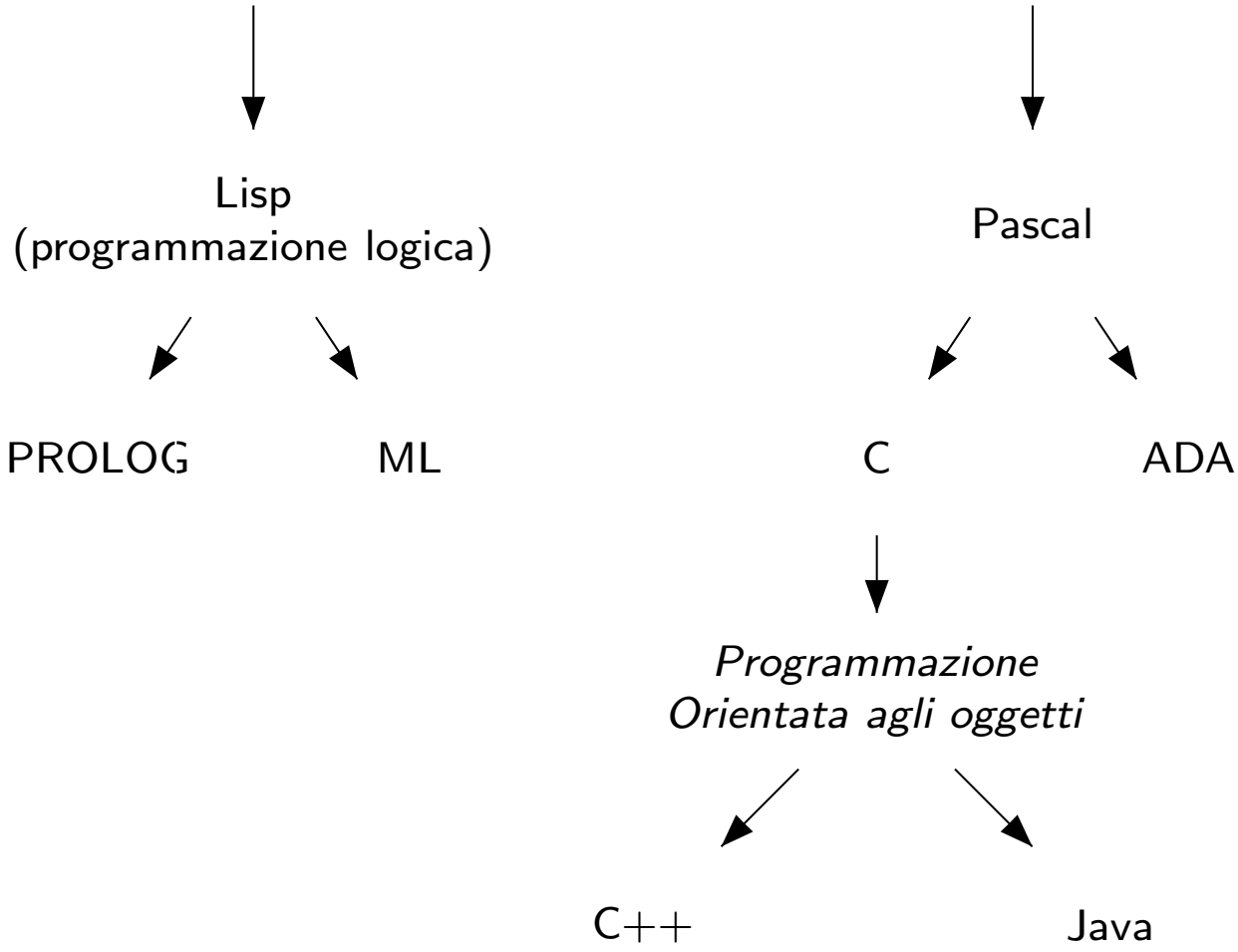
Linguaggi di Programmazione

Storia dei Linguaggi di Programmazione



Storia dei Linguaggi di Programmazione

Era Moderna: Programmazione strutturata, Metodologie, Astrazione, Linguaggi ad alto livello per la programmazione di sistema



Classificazione dei linguaggi di programmazione

- Una prima classificazione dei linguaggi di programmazione può essere fatta in base al loro livello di astrazione
- Si usano i termini linguaggi di "basso livello" e di linguaggi di "alto livello" per riferirsi, rispettivamente, alla vicinanza alla macchina hardware e alla vicinanza all'utente umano

Linguaggio Macchina

- Efficiente: puó essere interpretato ed eseguito senza "mediazioni" perchè le sue istruzioni corrispondono ad operazioni direttamente eseguibili dall'hardware della macchina
- Dipendente dall'architettura della macchina
  - Ogni processore ha un suo linguaggio macchina
  - Occorre conoscere l'architettura del processore
  - I programmi non sono portabili
- I programmi in linguaggio macchina sono sequenze di 0 ed 1 (binarie), sostanzialmente illegibili per l'uomo. Es: un programma per il calcolo della somma tra due numeri

codice operativo	*operando*
00000010	000000011011100
00000110	000000011111100
00000100	000000011011100

Linguaggio Assembler

- Costituisce un primo parziale passo verso la semplificazione della programmazione
- Le istruzioni binarie sono sostituite con dei **codici mnemonici** più facilmente comprensibili dall'uomo
- Ad esempio il programma precedente puó essere espresso come segue

codice operativo	*operando*	*codice mnemonoco*
00000010	000000011011100	LOAD 220
00000110	000000011111100	SUM 252
00000100	000000011011100	MEM 220

Linguaggi di alto livello

- In seguito alla realizzazione dei primi computer commerciali si é reso necessario sviluppare linguaggi di programmazione il cui uso risultasse più semplice rispetto al linguaggio assembler
- I linguaggi di programmazione di alto livello, mediante opportuni meccanismi di astrazione, permettono di usare costrutti che prescindono dalle caratteristiche fisiche della macchina
- Evoluzione dei linguaggi verso:
  1. astrazione dalla macchina,
  2. semplificazione della scrittura dei programmi e
  3. similarità con il ragionamento umano
- Sono più simili ai linguaggi naturali rispetto alle sequenze (almeno) apparentemente senza senso di 0 e 1 del linguaggio macchina, ma anche rispetto al linguaggio assembler

Linguaggi di alto livello

- Consentono l'utilizzo di simboli matematici e parole chiave tipiche dei linguaggio naturale
- Appositi software provvedono a tradurre le istruzioni di un linguaggio di alto livello nel loro equivalente in codice eseguibile dalla macchina
- Sono meno efficienti rispetto al linguaggio assembler e al linguaggio macchina

Programma per la somma di due numeri

- linguaggio macchina

```
00000010   0000000011011100
00000110   0000000011111100
00000100   0000000011011100
```

- linguaggio assembler

```
LOAD   220
SUM    252
MEM    220
```

- linguaggio di alto livello

```
a = a + b
```

dove `a` e `b` sono delle variabili, ossia dei nomi simbolici con cui identifichiamo una locazione di memoria astraendo dai dettagli di basso livello relativi agli indirizzi di memoria

Paradigmi di Programmazione

- Con il termine paradigma di programmazione si intende il "modo" in cui vengono specificati i programmi
- Non si tratta tanto del tipo di linguaggio usato, quanto del contesto più ampio al quale un dato linguaggio appartiene
- Parliamo di come viene organizzata la programmazione, con quali caratteristiche: stile, livello di dettaglio, "forma mentis" del programmatore
- Quattro principali paradigmi di programmazione: imperativa, funzionale, logica ed ad oggetti

Programmazione Imperativa

- Paradigma classico: i programmi sono delle sequenze di comandi che agiscono sui dati o sull'ordine di esecuzione delle istruzioni
- In genere viene studiato per primo ed è per questo (ma non solo) che viene percepito come il "più naturale"
- Esempi di linguaggi imperativi: Assembly, FORTRAN, C, COBOL, Pascal
- Costrutti tipici: assegnamento, cicli, if-then-else, procedure/funzioni con passaggio di parametri
- Il programmatore deve definire le strutture dati e gli algoritmi che operano su di esse
- Eseguire un programma significa eseguire una particolare sequenza di istruzioni

Programmazione Funzionale

- Un programma è una definizione di funzione nel senso matematico del termine
- Ordine superiore: gli argomenti delle funzioni definite possono essere sia valori di tipi primitivi che altre funzioni
- Un interprete si occupa di valutare, in base alle funzioni di base e a quelle definite dall'utente, una qualsiasi espressione ben formata
- L'algoritmo di valutazione non viene scritto dal programmatore ma varia a seconda del particolare linguaggio usato
- Esempi di Linguaggi Funzionali: ML, Lisp, CAML
- L'esecuzione del programma consiste nella chiamata di una funzione con i relativi parametri per ottenere un risultato

Un esempio in CAML

```
#let rec fatt = function
#    0 -> 1
#    | n -> n * fatt n-1;;
fatt:  int -> int = <fun>

#let rec map = function
#    [], f -> []
#    | (x::tail), f -> (f x) ::  map tail, f;;
map:  'a list * ('a -> 'b) -> 'b list = <fun>

#let map [2; 0; 3] fatt;;
    - int list = [2; 1; 6]

#let map [0; 3] (function x->x+1);;
    - int list = [1; 4]
```

Programmazione Funzionale

- Primo linguaggio funzionale: λ-calcolo (nucleo della programmazione funzionale)
- Molto importante soprattutto dal punto di vista teorico (la versione non tipata del λ-calcolo è Turing-equivalente)
- Altri linguaggi: Miranda, Haskell, Scheme, Standard ML (diverse implementazioni tra cui il CAML) ed il Lisp
- Si differenziano per l'efficienza, nella vastità delle librerie, nel tipo di valutazione
- Approccio dichiarativo: consente di fornire una descrizione del problema, attraverso un insieme di funzioni, più che l'algoritmo per risolverlo
- Le moderne implementazioni sono piuttosto efficienti

Programmazione Logica

- Un programma è un insieme di predicati della logica del primo ordine
- I predicati sono descritti come della clausole di Horn

$$q(Y_1, \ldots, Y_h) \leftarrow p_1(X_1^1, \ldots, X_n^1) \wedge \ldots \wedge p_k(X_1^k, \ldots, X_m^k)$$

dove $n, m, k, h \geq 0$, $q, p_1, \ldots, p_k$ sono simboli di predicato ed X_j^i, Y_i sono termini di un linguaggio

- Termini: variabile X oppure $f(X_1, \ldots, X_n)$ dove, f è una funzione di arietà n ed $X_1, \ldots, X_n$ sono dei termini

Programmazione Logica: un esempio

```
#append([], Xs, Xs).
#append([X | Xs], Ys,[X | Zs]) :-
            append(Xs, Ys, Zs)
```

una clausola sempre vera

:- sta per ←

```
?- append([1,2],[2,4],X).
X = [1,2,2,4]

?- append(X,[3],[2,5,3])
X = [2,5]

 ?- append(X,Y,[3,4,5]).
  X = [], Y=[3,4,5] ;
  X = [3], Y=[4,5] ...
```

```
?- append(X,Y,[3,4|Z]).
 X = [3], Y = [4|Z] ;
 X = [3,4], Y = Z
```

Programmazione Logica

- Anche in questo caso il formalismo è dichiarativo: i programmi sono definizioni, in questo caso di relazioni e non di funzioni
- Il motore di calcolo è un *dimostratore di teoremi* che cerca di dimostrare un *goal* (una clausola Horn senza la conseguenza, ad esempio: `?- append(X,Y,[3,4,5]))` a partire di predicati del programma
- Il "risultato" è ottenuto mediante dagli assegnamenti alle variabili libere del goal (nel nostro esempio `X` e `Y`) che il dimostratore esegue durante la visita (con backtracking) dell'albero di dimostrazione per il goal
- Linguaggio principale: PROLOG (diverse implementazioni)

Programmazione Ad Oggetti

- Di recente formalizzazione; ha avuto molto successo
- I programmi definiscono delle astrazioni degli elementi del dominio applicativo del programma, tali astrazioni vengono dette **classi**
- Le classi contengono informazioni sui dati ma anche il codice per gestirli (in genere di tipo imperativo)
- Un programma è una insieme di **oggetti** (istanze di classi) che possono interagire gli uni con gli altri scambiandosi dei messaggi
- L'esecuzione del programma è il risultato dell'interazione degli oggetti che lo costituiscono
- Esempi di linguaggi ad oggetti: C++, Java, Smalltalk, Eiffel

Programmazione Concorrente

- Un programma concorrente consiste di diversi processi (programma in esecuzione) che cooperano in un ambiente per raggiungere un risultato comune
- I processi possono avere meccanismi diversi di comunicazione (canali, memoria condivisa, ...)
- L'attenzione della programmazione concorrente non è tanto sul calcolo che effettuano i singoli processi, ma sulle loro interazioni
- Numerosi formalismi possibili
  - Algebre di Processo: CCS, CSP, ...
  - Set di costrutti concorrenti che si aggiungono a linguaggi classici: Linda, Pascal o C concorrenti
- La formalizzazione è importante per effettuare verifiche formali di proprietà (es. mutua esclusione, raggiungibilità di stati, invarianze) di programmi concorrenti

Traduzione

- Linguaggi ad alto livello (Lisp, Fortran, Algol, Pascal, C, Java, ...)
  - Più vicini al linguaggio naturale
  - Più vicini al linguaggio naturale
  - Più semplici da utilizzare
  - Più flessibili e potenti
  - Indipendenti dalla particolare architettura della macchina
  - Non possono essere direttamente eseguiti su un certa macchina, ma devono essere tradotti in linguaggio macchina
- Traduttori
  - generano codice in linguaggio macchina a partire da codice scritto in un linguaggio ad alto livello
  - Si distinguono in **compilatori** ed **interpreti**

Part II

Macchine Astratte: definizione e tipi di implementazione

La nozione di Macchina Astratta

- I meccanismi di astrazione giocano spesso un ruolo fondamentale in informatica in quanto, isolando gli aspetti rilevanti in un particolare contesto, permettono di dominare la complessità della maggiorparte dei sistemi di calcolo
- Una delle nozioni più generali che coinvolgono l'astrazione è quello di macchina astratta
- La nozione di macchina astratta è strettamente collegata a quello di linguaggio di programmazione
- Ci permette di descrivere cosa sia l'implementazione di un linguaggio, senza addentrarci nei dettagli di una particolare implementazione
- Descriveremo in termini generali i concetti di interprete e compilatore di un linguaggio

La nozione di Macchina Astratta

- Intuitivamente, una macchina astratta è una astrazione del concetto di macchina fisica, intesa come un calcolatore che consente di eseguire degli algoritmi
- Per poter essere eseguiti, questi algoritmi devono essere codificati in termini di costrutti di un linguaggio di programmazione, ossia devono essere rappresentati mediante istruzioni di un opportuno linguaggio di programmazione $\mathcal{L}$
- Il linguaggio $\mathcal{L}$ sarà definito da una specifica sintassi e semantica
- Non serve specificare ulteriormente la natura di $\mathcal{L}$; basta sapere che la sintassi di $\mathcal{L}$ definisce un numero finito di costrutti che possono essere usati per comporre programmi (il ruolo dell'astrazione ...)
- Un programma di $\mathcal{L}$ non è altro che un insieme finito di istruzioni di $\mathcal{L}$

La nozione di Macchina Astratta

Definition

Sia $\mathcal{L}$ un dato linguaggio di programmazione. Una macchina astratta per $\mathcal{L}$ (denota con $\mathcal{M}_{\mathcal{L}}$) è un qualsiasi insieme di strutture dati ed algoritmi in grado di memorizzare ed eseguire algoritmi scritti in $\mathcal{L}$

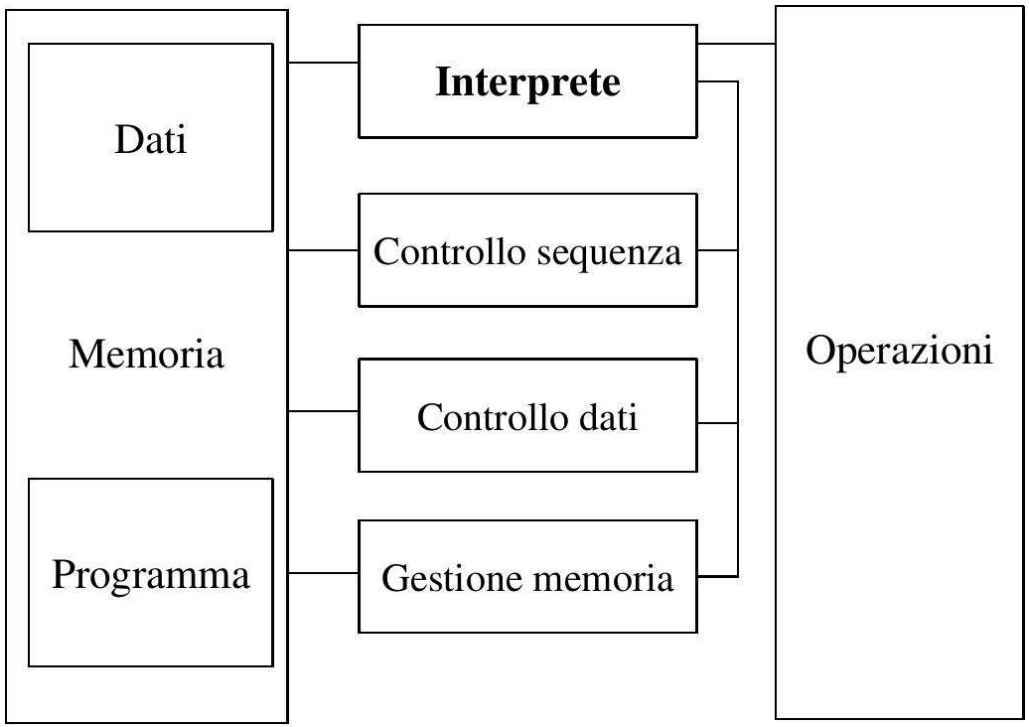
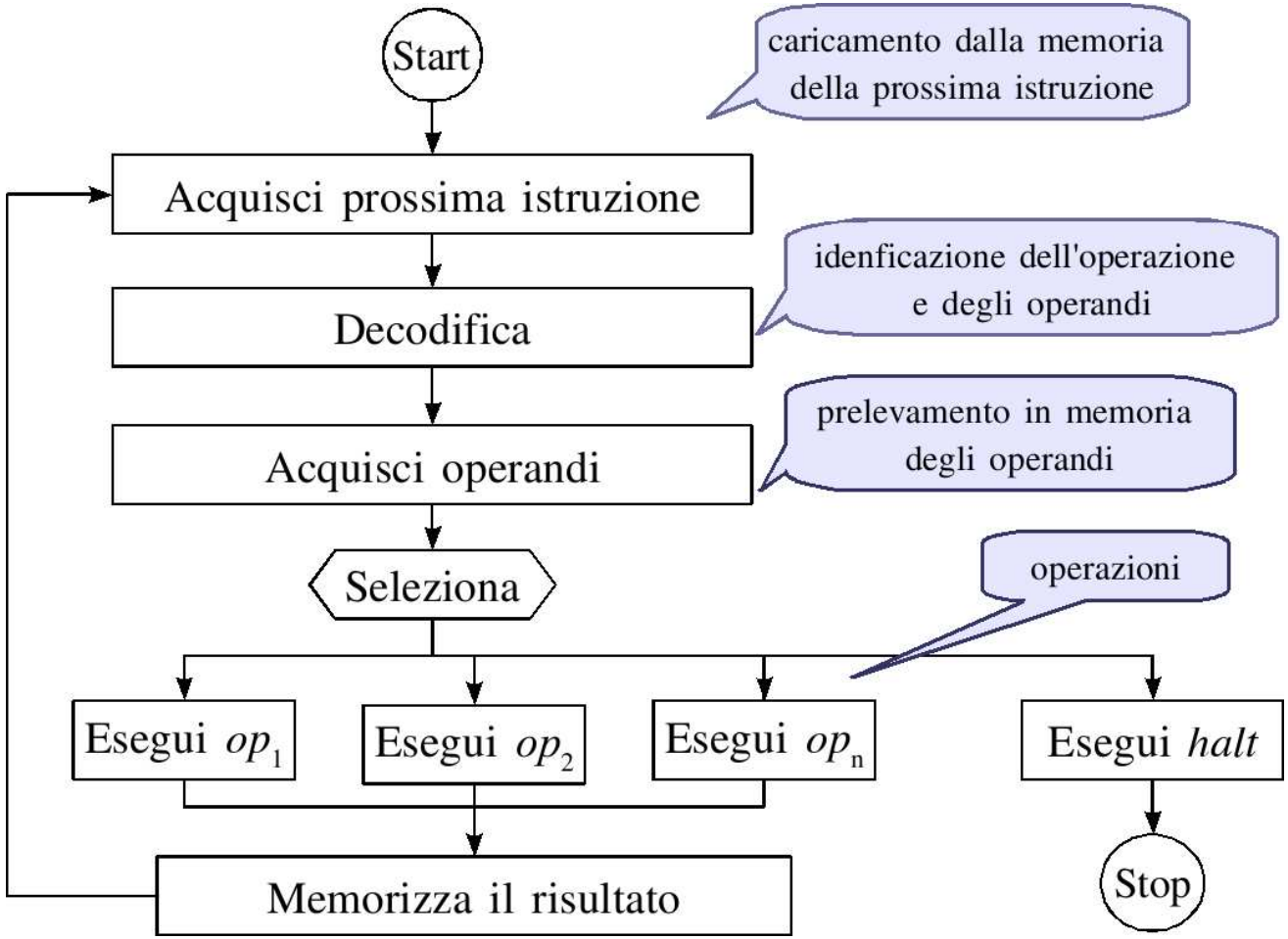


Figure: Struttura di una macchina astratta

La nozione di Macchina Astratta

- Una generica macchina astratta $\mathcal{M}_{\mathcal{L}}$ è composta da una memoria e da un interprete
- La memoria serve per immagazinare dati e programmi; l'interprete è la componente che esegue le istruzioni che costituiscono i programmi
- L'interprete esegue delle operazioni specifiche che dipendono dal particolare linguaggio $\mathcal{L}$ che deve essere interpretato, tuttavia possiamo identificare delle operazioni comuni a tutti gli interpreti:
  - Elaborazione dati primitivi
  - Controllo della sequenza di esecuzione delle istruzioni
  - Trasferimento dati
  - Gestione della memoria

Ciclo di esecuzione di un generico interprete



Una macchina fisica come una macchina astratta

- Operazioni Primitive: operazioni aritmetico-logiche realizzate dalla ALU, ossia operazioni aritmetiche su interi e numeri in virgola mobile, operazioni booleane, shift, e confronto
- Controllo della sequenza: la struttura principale è il registro contatore di programma (PC); principali operazioni: incremento e salti
- Trasferimento dati: le strutture canoniche sono i registri della CPU che interfacciano con la memoria princale, il registro indirizzo dati (MAR, *Memory Address Register*) ed il registro dei dati (MDR, *Memory Data Register*); operazioni per modificare il contenuto di questi registri ed per realizzare le diverse modalità di indirizzamento

Una macchina fisica come una macchina astratta

- Gestione della memoria: nelle usuali macchine a registri la gestione della memoria è statica; nelle macchine a pila troveremo la struttura dati pila con le relative operazioni di push e pop
- il ciclo di controllo dell'interprete è realizzato tramite un insieme di dispositivi che costituisco l'Unità di Controllo che consentono di eseguire il ciclo *fetch-decode-execute* analogo a quello di un generico interprete

Implementazione di un linguaggio

- Una macchina astratta $\mathcal{M}_{\mathcal{L}}$ è per definizione un dispositivo che consente di eseguire programmi scritti nel linguaggio $\mathcal{L}$
- Una macchina corrisponde univocamente ad un linguaggio, il suo linguaggio macchina
- Al contrario, dato un linguaggio $\mathcal{L}$ vi sono molte macchine astratte che hanno $\mathcal{L}$ come proprio linguaggio macchina
- Tali macchine differiscono per il modo in cui è realizzato l'interprete e per le strutture dati che utilizzano
- Implementare un linguaggio di programmazione $\mathcal{L}$ significa realizzare una macchina astratta che abbia $\mathcal{L}$ come linguaggio macchina

Realizzazione di una macchina astratta

Una macchina astratta è realizzata mediante tre tecniche principali (o eventuali loro combinazioni)

1. Realizzazione in *hardware*
2. Simulazione mediante *software*
3. Simulazione (emulazione) mediante *firmware*

Realizzazione in hardware

- È, in linea di principio, sempre realizzabile: si tratta di realizzare una macchina fisica il cui linguaggio macchina coincida con $\mathcal{L}$
- Ha il vantaggio di essere molto efficiente perchè i programmi di $\mathcal{L}$ vengono eseguiti direttamente su dispositivi fisici
- I costrutti di un linguaggio $\mathcal{L}$ ad alto livello sono molto lontani dalle funzionalità elementari dei dispositivi fisici, il costo di realizzazione tramite hardware è in generale molto alto e, spesso, richiede l'uso di hardware specifico per la macchina che vogliamo realizzare
- È una soluzione che offre scarza flessibilità
- Viene usato solo per linguaggi di basso livello i cui costrutti sono molti vicini alle operazioni che possiamo implementare direttamente in hardware

Simulazione tramite software

- Consiste nel realizzare le strutture dati e gli algoritmi di $\mathcal{M}_{\mathcal{L}}$ tramite programmi scritti in un altro linguaggio $\mathcal{L}'$ che possiamo supporre già implementato
- Possiamo realizzare la macchina $\mathcal{M}_{\mathcal{L}}$ mediante opportuni programmi in $\mathcal{L}'$ che interpreterano i costrutti di $\mathcal{L}$ simulando, così, le funzionalità di $\mathcal{M}_{\mathcal{L}}$
- Massima flessibilità; implementare nuove caratteristiche di $\mathcal{L}$ equivale a modifica e/o riscrive i programmi in $\mathcal{L}'$ che simulano $\mathcal{M}_{\mathcal{L}}$
- Minore efficienza rispetto alla soluzione via hardware

Simulazione tramite firmware

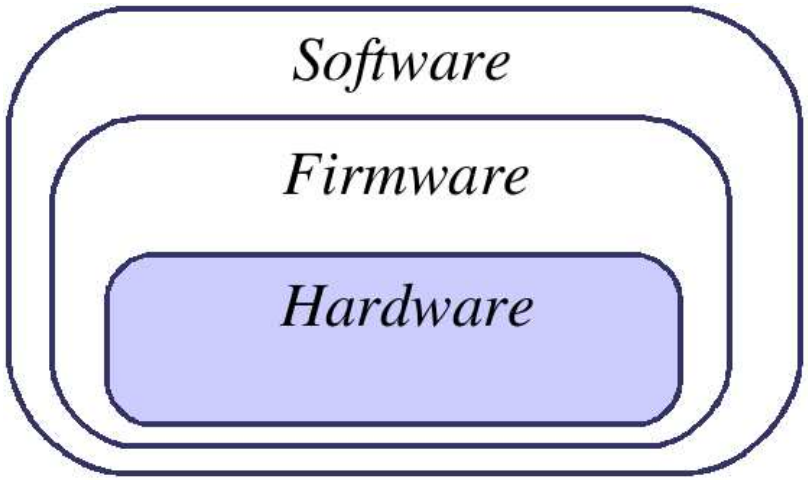
- Simulazione (emulazione) di strutture dati e algoritmi della macchina astratta $\mathcal{M}_{\mathcal{L}}$ tramite dei microprogrammi
- Le techine di microprogrammazione vennero introdotte negli anni '60 per permettere a calcolatori diversi di condividere uno stesso insieme di istruzioni (IBM 360)
- I microprogrammi usano un linguaggio di livello molto basso costituito da microistruzioni
- Le microistruzioni specificano semplici operazioni di trasferimento dati fra registri, da e per la memoria principale, ed operazioni aritmetiche
- Una classe di particolari microistruzioni consente di realizzare il ciclo dell'interprete.
- Un microprogramma non risiede in memoria principale ma su speciali memorie di sola lettura

Simulazione tramite firmware

- Concettualmente è molto simile alla simulazione tramite software: in entrambi i casi la $\mathcal{M}_{\mathcal{L}}$ è simulata mediante oppurtuni programmi che sono poi eseguiti dalla macchina fisica
- Nel caso della simulazione tramite software, la macchina astratta è simulato da programmi scritti in un linguaggio ad alto livello
- Nel caso della simulazione tramite firmware, la macchina astratta è simulato da microprogrammi
- Un microprogramma risiede su speciali memorie di sola lettura per poter essere eseguiti dalla macchina fisica ad alta velocità
- Questo tipo di realizzazione è più efficente rispetto ad una ottenibile via software ma più lenta rispetto ad una realizzazione in hardware
- La flessibilità di questa soluzione è inferiore rispetto a quella di una simulazione via software, ma superiore rispetto ad una realizzazione in hardware

Una soluzione "mista"

Spesso la realizzazione di una macchina astratta prevede una combinazione delle tre metodologie fornendo una gerarchia di macchine astratte

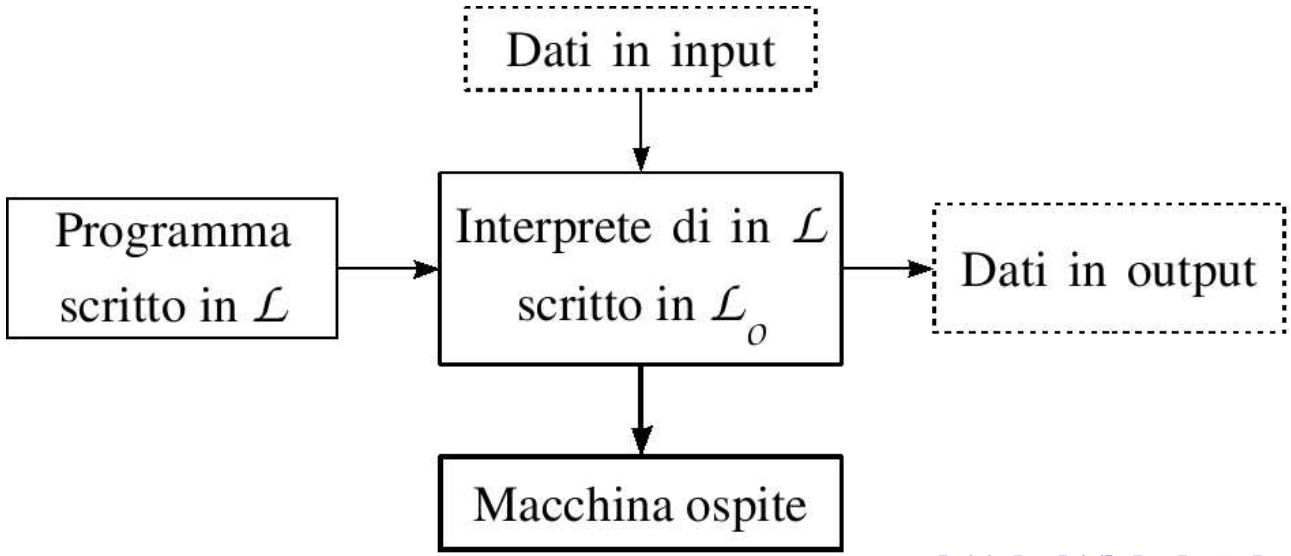


Possibili implementazioni di una MA

- Assumiamo di dover implementare un generico linguaggio $\mathcal{L}$ (ossia di dover realizzare una macchina astratta $\mathcal{M}_{\mathcal{L}}$)
- Assumiamo di avere a disposizione una macchina astratta ospite ($\mathcal{M}_{\mathcal{O}_{\mathcal{L}_{\mathcal{O}}}}$) già implementata che ci permette di usare direttamente i costrutti del suo linguaggio macchina ($\mathcal{L}_{\mathcal{O}}$)
- Intuitivamente, l'implementazione di $\mathcal{L}$ sulla macchina ospite $\mathcal{M}_{\mathcal{O}_{\mathcal{L}_{\mathcal{O}}}}$ avviene mediante una qualche **traduzione** di $\mathcal{L}$ in $\mathcal{L}_{\mathcal{O}}$
- Distinguiamo due modalità di implementazione concettualmente molto diverse fra loro
- *Implementazione interpretativa pura*: ogni costrutto di $\mathcal{L}$ viene tradotto tramite un corrispondente (equivalente) programma di $\mathcal{L}_{\mathcal{O}}$
- *Implementazione compilativa pura*: traduzione esplicita di programmi in $\mathcal{L}$ in corrispondenti programmi di $\mathcal{L}_{\mathcal{O}}$

Implementazione interpretativa pura

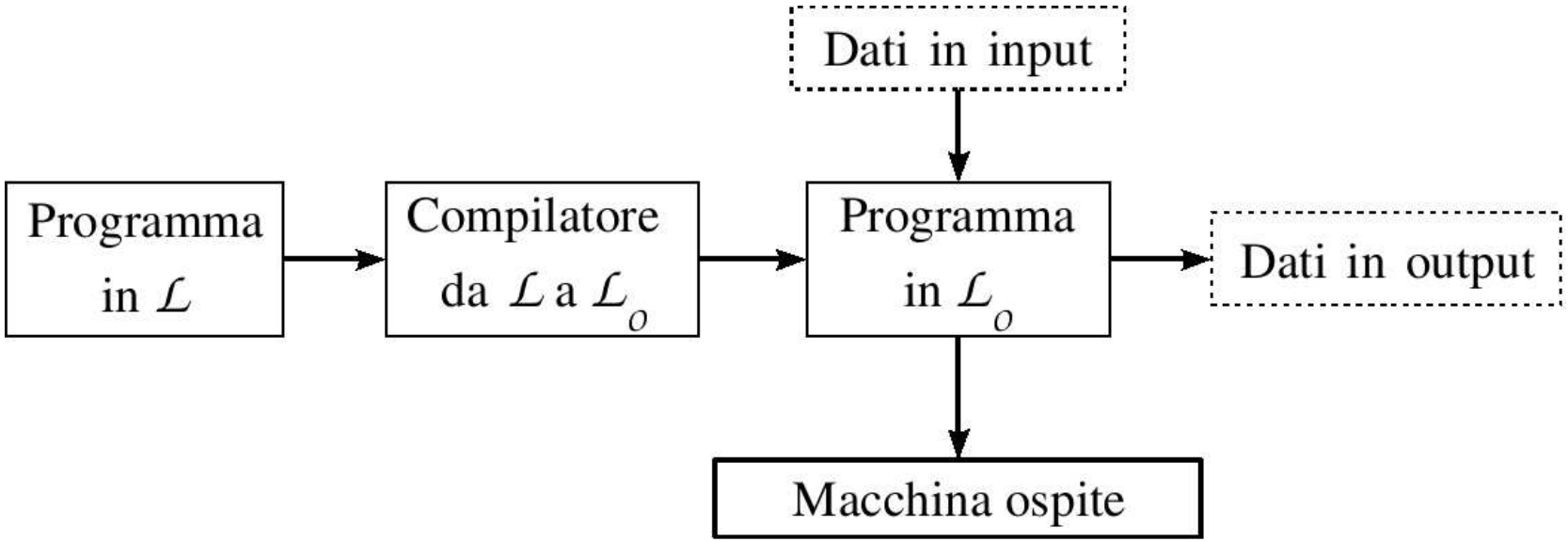
- Si realizza un programma, detto **interprete**, scritto in $\mathcal{L}_{\mathcal{O}}$ in grado di interpretare tutte le possibili istruzioni di $\mathcal{L}$
- Per eseguire un programma $\mathcal{P}^{\mathcal{L}}$ scritto in $\mathcal{L}$ con un dato in input D basterà eseguire, sulla macchina ospite, l'interprete con $\mathcal{P}^{\mathcal{L}}$ e D come dati in input
- Non viene effettuata una traduzione esplicita dei programmi scritti in $\mathcal{L}$: l'interprete, per poter eseguire un'istruzione del linguaggio $\mathcal{L}$ le fa corrispondere un certo numero di istruzioni di $\mathcal{L}_{\mathcal{O}}$



Implementazione compilativa pura

- Il linguaggio $\mathcal{L}$ viene implementato traducendo in maniera esplicita i programmi scritti in $\mathcal{L}$ in programmi scritti in $\mathcal{L}_{\mathcal{O}}$
- In questo caso, la traduzione è eseguita da uno specifico programma chiamato **compilatore**; $\mathcal{L}$ è detto linguaggio sorgente, mentre $\mathcal{L}_{\mathcal{O}}$ è detto linguaggio oggetto
- Per eseguire un programma $\mathcal{P}^{\mathcal{L}}$ (scritto in $\mathcal{L}$) con un dato in input D, dovremmo:
  - eseguire il compilatore con $\mathcal{P}^{\mathcal{L}}$ come input (compilazione di $\mathcal{P}^{\mathcal{L}}$) ed ottenere il programma compilato $\mathcal{P}_{\mathcal{C}}^{\mathcal{L}_{\mathcal{O}}}$
  - eseguire $\mathcal{P}_{\mathcal{C}}^{\mathcal{L}_{\mathcal{O}}}$ sulla macchina ospite con in input il dato D per ottenere il risultato desiderato
- Notare come, in questo caso, la fase di traduzione è separata da quella di esecuzione

Implementazione compilativa pura



Vantaggi/svantaggi dell'implementazione interpretativa

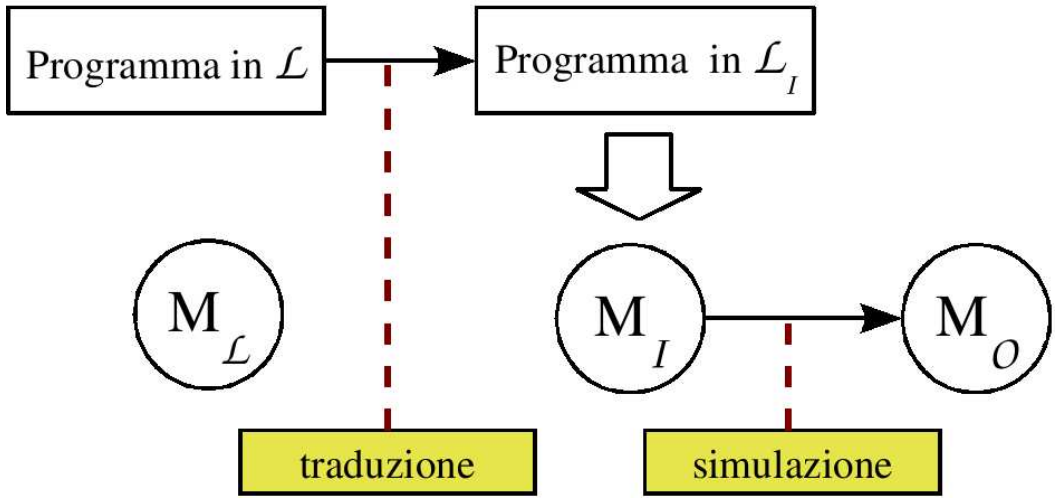
- Scarsa efficienza: la decodifica dei vari costrutti di $\mathcal{L}$ viene eseguita in fase di esecuzione del programma; ai tempi intrinsecamente richiesti dal programma bisogna aggiungere i tempi necessari per la decodifica
- Flessibilità: interpretare i costrutti del linguaggio in fase di esecuzione consente di interagire in modo diretto con l'esecuzione del programma (cosa particolarmente utile in fase di debugging del programma)
- In generale, la scrittura di un interprete è più semplice rispetto alla scrittura di un compilatore
- Permette di usare una quantità di memoria ridotta (il programma viene memorizzato solo nella sua versione sorgente)

Vantaggi/svantaggi dell'implementazione compilativa

- La traduzione del programma sorgente in programma oggetto avviene separatamente all'esecuzione di quest'ultimo; l'esecuzione del programma compilato è in generale più efficiente rispetto rispetto a quello che si otterebbe con una versione interpretativa
- Perdita di informazioni relative alla struttura del programma sorgente, il che rende più difficile l'interazione con il programma a run-time (se si verifica un errore a run-time diventa difficile individuarne la causa)
- Difficoltà nel realizzare strumenti di debugging

Traduzione e Macchine Intermedie

- I due tipi di implementazione (interpretativa e compilativa) visti finora sono due casi limite che nella realtà non esistono quasi mai
- In una situazione reale c'è quasi sempre una soluzione mista



- Una fase di **traduzione** in un linguaggio intermedio seguita da una fase di **simulazione** della macchina intermedia
- Interpretazione pura: $\mathcal{M}_{\mathcal{L}} = \mathcal{M}_{\mathcal{I}}$
- Compilazione pura: $\mathcal{M}_{\mathcal{I}} = \mathcal{M}_{\mathcal{O}}$

Linguaggi compilati, interpretati, semi-compilati

- Linguaggi compilati: implementati tramite un compilatore
- Linguaggi interpretati: implementati tramite un interprete
- Linguaggi semi-compilati: soluzione "mista"
- Un esempio di linguaggio semi-compilato **Java**: una prima fase di compilazione (il programma sorgente viene compilato per ottenere la sua rappresentazione intermedia – bytecode) è seguita da un fase di interpretazione (il bytecode viene successivamente tradotto ed eseguito da un interprete – JVM)