



**Corso di Laurea Magistrale in Ingegneria Informatica
A.A. 2010-2011**

Linguaggi Formali e Compilatori

Introduzione

Giacomo PISCITELLI

Scheduling

Orario delle lezioni

Lunedì	ore 17.30 - 19.10	aula	10
Martedì	ore 15.50 - 17.30	aula	8
Giovedì	ore 14.10 - 15.50	aula	5

Chiarimenti e informazioni

Orario ricevimento

Giovedì ore 11.30 – 13.15

Per appuntamento

e-mail: piscitel@poliba.it

Obiettivi

- ⇒ Cos'è un linguaggio di programmazione.
- ⇒ Come sono definiti i linguaggi di programmazione e come si definiscono i programmi.
- ⇒ Come questi linguaggi sono “implementati”, ovvero come vengono eseguiti su una macchina (tipicamente, un computer).
- ⇒ Quali sono le tecniche di traduzione per i moderni linguaggi di programmazione sia di tipo *general purpose* che per applicazioni specifiche.

Requisiti preliminari

- ✓ Conoscenza dei linguaggi di programmazione
- ✓ Conoscenza dei paradigmi di programmazione
- ✓ Conoscenza delle diverse modalità di allocazione della memoria
- ✓ Capacità di sviluppare applicazioni software anche complesse
- ✓ Conoscenza delle principali strutture dati e degli algoritmi per la loro gestione

Course Outline

Linguaggi di programmazione

Tecniche di implementazione dei linguaggi

Linguaggi e macchine astratte

Traduttori, compilatori e interpreti

Alfabeto, stringhe, vocabolario

Linguaggio, approcci alla definizione

Grammatiche e classificazione

Grammatiche regolari e context-free

Backus-Naur Form (BNF)

Automi e Macchine di Turing

Struttura di un compilatore

Analisi lessicale (scanning)

Implementazione di scanner

Analisi sintattica (parsing)

Top-Down Parsing, LL Parsing

Bottom-Up Parsing, LR Parsing

Implementazione di scanner

Analisi Semantica

Syntax-Directed Translation

Attribute Definitions

Rappresentazione intermedia

Generazione del codice

Tecniche di ottimizzazione del codice

Modalità di esame

elaborato

gruppi di 2 – 3 persone

esame orale

discussione del progetto
contenuti teorici del corso

Materiale didattico

Testi

A.H. Aho, M.S. Lam, R. Sethi, J.D. Ullman,
“Compilers: principles, techniques & tools”, second edition
“Compilatori: principi, tecniche e strumenti”, seconda edizione,
Pearson/Addison-Wesley, 2007/2009.

*Il libro è spesso chiamato **Dragon Book** (il libro del dragone) a causa dell'immagine della copertina che, almeno nell'edizione originaria, raffigura il Cavaliere della Programmazione che lotta con il Dragone del design dei compilatori.*

T.W. Pratt, M.V. Zelkowitz,
“Programming Languages: Design and Implementation”,
Prentice Hall

Lucidi e materiale del docente

Altri testi

David Gries
“Principi di progettazione dei compilatori”
Collana di Informatica, Franco Angeli Editore

J.E. Hopcroft, R. Motwani, J.D. Ullman
“Automi, Linguaggi e calcolabilità” vol. primo: Metodi sintattici
Addison-Wesley, 2003

Cos'è un Linguaggio di Programmazione (LdP)?

Una **qualsiasi notazione per descrivere algoritmi e strutture dati** può essere considerata un **linguaggio di programmazione**.

Le “frasi” di questi linguaggi sono programmi validi.

Per eseguire un programma, esso deve essere tradotto in una forma che può essere capita da un calcolatore → **traduttore**

Per realizzare un traduttore si fa uso di principi e tecniche utili non solo per costruire compilatori, ma anche in molte altre aree dell'informatica: linguaggi di programmazione, architettura, teoria dei linguaggi, algoritmi, ingegneria del software, ecc.

Quanti sono i linguaggi di programmazione?

Risposta:

1348

O almeno tanti sono i linguaggi che compaiono (07/09/2010) all'URL:

<http://99-bottles-of-beer.ls-la.net/>

(ma molti linguaggi non sono inclusi)

È necessario qualche criterio per catalogare i diversi linguaggi!

Sul sito 99-bottles-of-beer usano l'ordinamento alfabetico. . .

Non è comunque pensabile conoscerli tutti!

Perché studiare i linguaggi di programmazione?

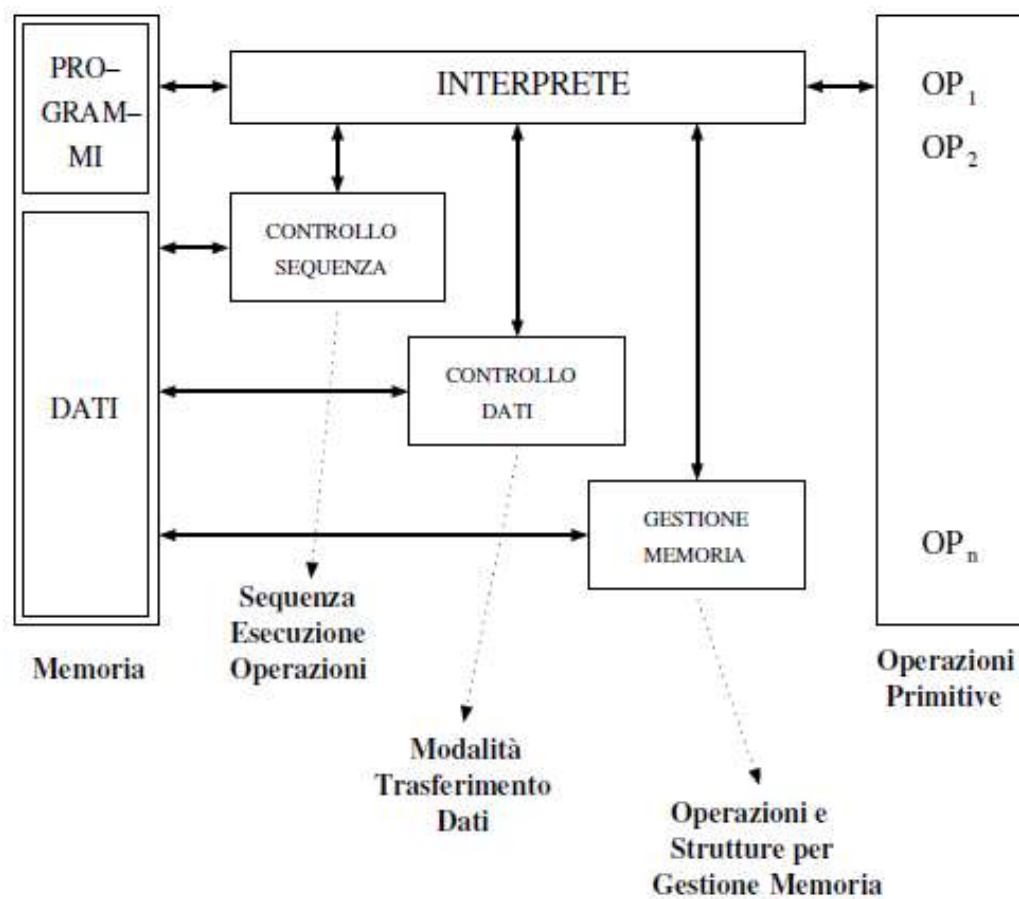
- ⇒ per sfruttare al meglio i LdP conosciuti e migliorare la capacità di sviluppare algoritmi efficienti
- ⇒ per ampliare il vocabolario dei costrutti disponibili
- ⇒ per comprendere somiglianze e diversità fra i LdP
- ⇒ per facilitare la scelta del LdP più adatto
- ⇒ per facilitare l'apprendimento di nuovi LdP
- ⇒ per facilitare il progetto di nuovi LdP

Linguaggi e Macchine

$$L \Leftrightarrow M_L$$

L : linguaggio di programmazione

M_L : macchina che ha L come linguaggio macchina



Il caso **HARDWARE**

Macchina HW convenzionale (o di von Neumann)

PROGRAMMI

→ sequenze di istruzioni in linguaggio macchina HW

DATI

→ registri, RAM (byte-parole), memoria di massa (settori)

OPERAZIONI MACCHINA

→ operazioni aritmetico-logiche, lettura/scrittura RAM, . . .

CONTROLLO SEQUENZA

→ strutture dati: program counter, operazioni di salto, . . .

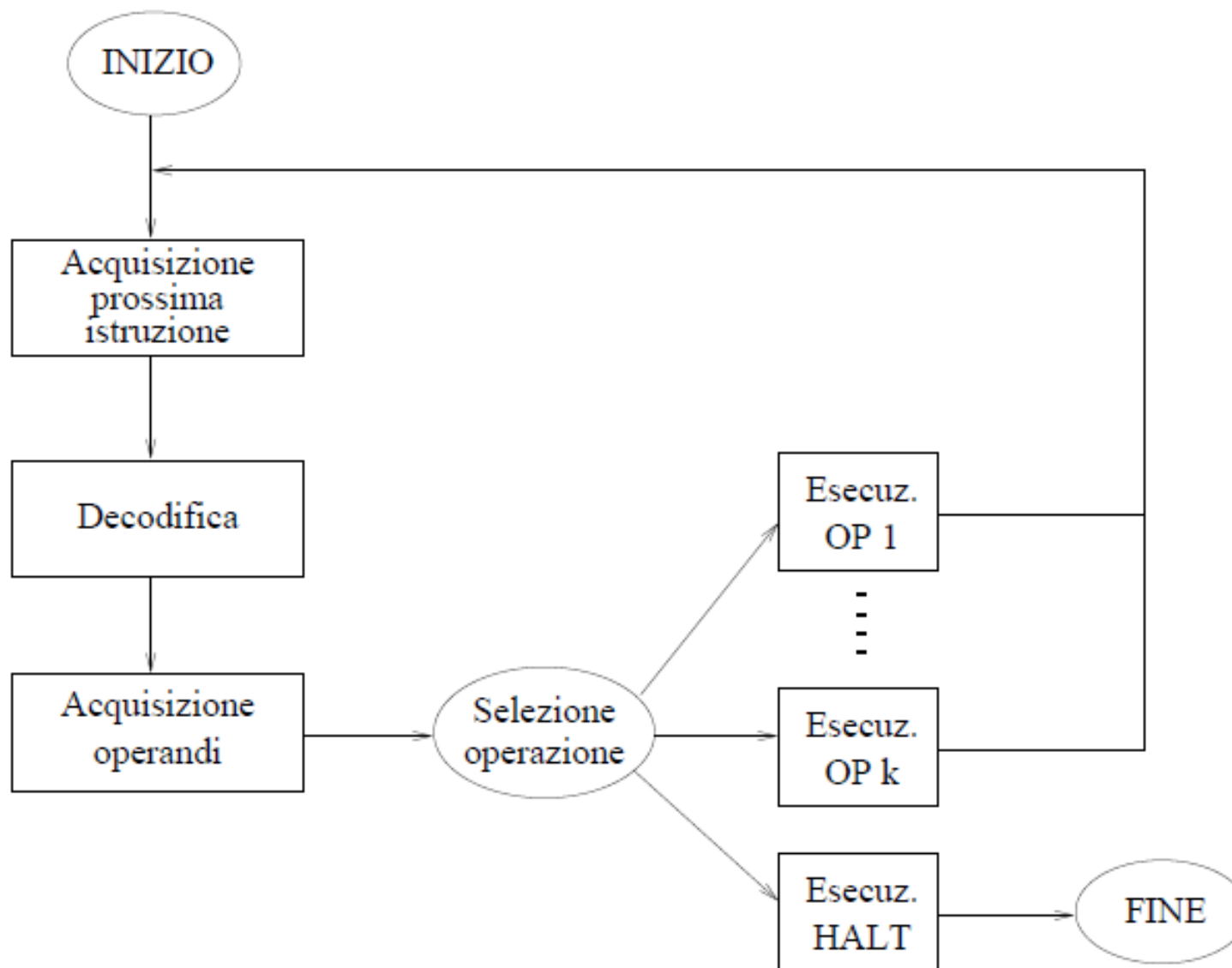
GESTIONE MEMORIA

→ stack, . . .

TRASFERIMENTO DATI

→ modalità di indirizzamento, registro indice, . . .

Interprete



Linguaggi e Macchine Astratte

Ogni linguaggio induce una macchina astratta definita dall'implementazione del linguaggio:

$$L \Rightarrow M_L$$

L : linguaggio

M_L : macchina che ha L come linguaggio macchina

Ogni macchina astratta definisce un “linguaggio macchina”:

$$M \rightarrow L_M$$

M : macchina

L_M : linguaggio interpretato da M (programmi in $L_M \Leftrightarrow$ dati primitivi di M)

Paradigmi di linguaggi

È possibile catalogare i linguaggi in vari paradigmi:

Linguaggi imperativi (o procedurali)

Esempi: Pascal, C, FORTRAN, Perl

Linguaggi applicativi (o funzionali)

Esempi: LISP, ML

Linguaggi orientati agli oggetti

Esempi: C++, Java, Smalltalk, Python

Linguaggi basati su regole (o dichiarativi)

Esempi: Prolog, YACC, Make

Linguaggi per sistemi distribuiti

Esempi: Lotos, CSP, PICT

Cosa distingue i diversi paradigmi?

Caratteristiche di un LdP

Un linguaggio di programmazione è definito da:

- ⇒ **lessico**: le parole chiave, i simboli e le regole per la costruzione dei nomi
- ⇒ **sintassi**: la forma di un programma
- ⇒ **semantica**: il significato di un programma
- ⇒ **implementazione**: come funziona un programma
 - tramite **compilazione** (traduzione) in un altro linguaggio
 - tramite **interpretazione**

Gerarchia di Macchine

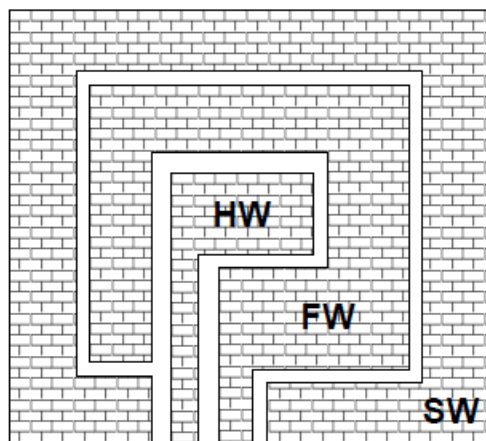
La macchina astratta di un LdP è normalmente composta da una gerarchia di macchine:

- M4 Macchina Virtuale sviluppata dal Programmatore**
Implementata dal modello di esecuzione sviluppato in codice C
- M3 Macchina Virtuale C**
Implementata da librerie di run-time
- M2 Macchina Virtuale del Sistema Operativo**
Implementata da programmi in linguaggio macchina eseguiti dal FW
- M1 Macchina Virtuale Firmware (FW)**
Implementata da microcodice eseguito dall'hardware
- M0 Macchina Virtuale Hardware (HW)**
Implementata da dispositivi fisici

Realizzazione delle Macchine Astratte

La realizzazione di una macchina virtuale può essere:

- **diretta** via hardware (HW)
- **emulata** via firmware (FW)
- emulata via software (SW)
- una opportuna **combinazione** di HW, FW e SW

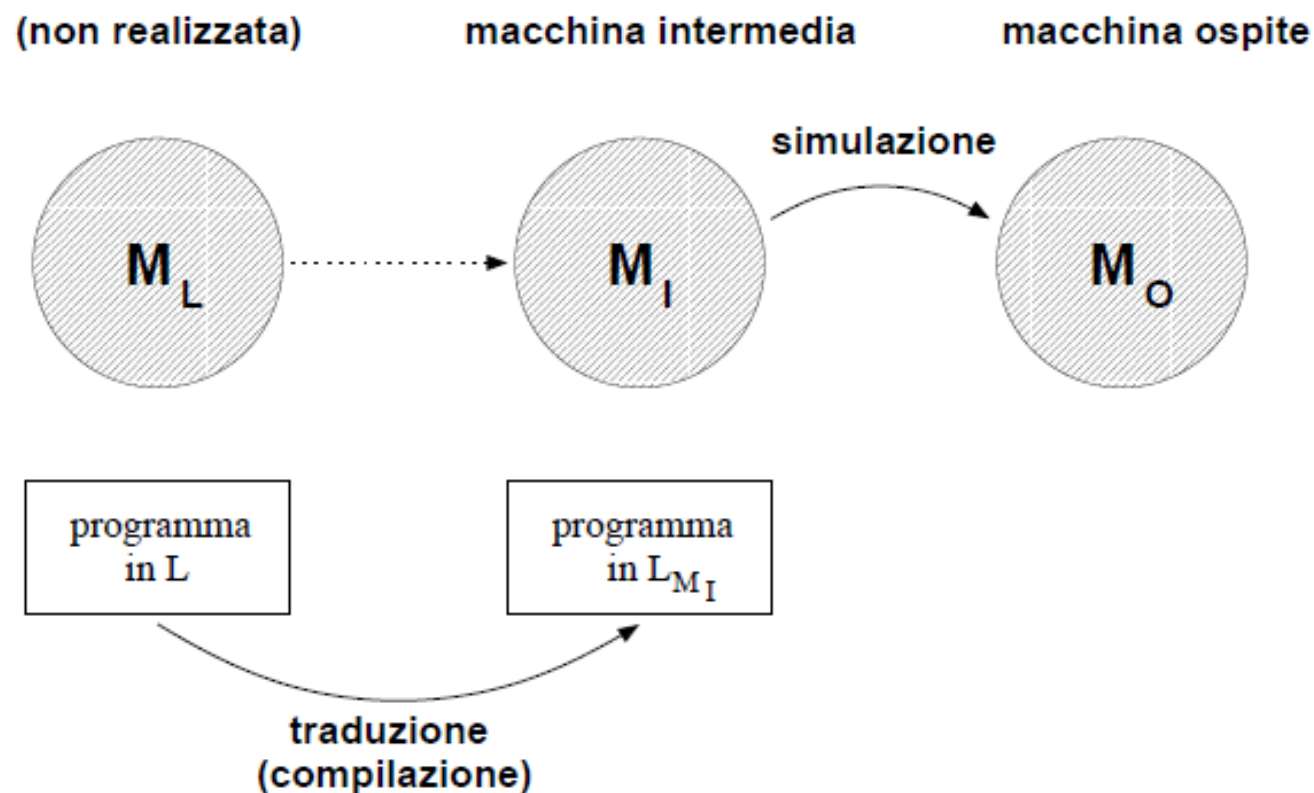


Implementazione di un LdP

L'implementazione del linguaggio L prevede:

1. La **definizione** di M_L
2. La “**implementazione**” di M_L
 - ✓ **direttamente** via HW
 - ✓ **tramite emulazione** di M_L su una macchina ospite M_O :
 - interpretazione
 - traduzione

Emulazione di M_L su M_O



$M_L = M_I \Rightarrow$ implementazione puramente interpretativa

$M_I = M_O \Rightarrow$ implementazione puramente compilativa

Interpretazione e Compilazione

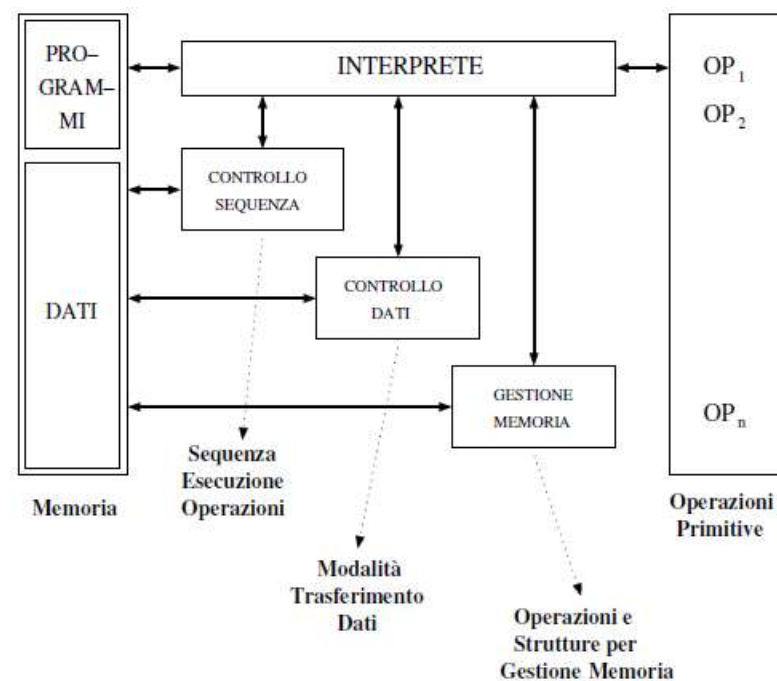
Per i linguaggi di alto livello non esistono implementazioni puramente interpretative o **compilative**: esse sono ottenute **combinando opportunamente traduzione e interpretazione**.

Nelle implementazioni **compilative** la parte di M_I differente da M_O è il cosiddetto **supporto a run time** (ad esempio in C il supporto per effettuare I/O).

Nelle implementazioni **interpretative** (anche le più spinte) c'è almeno una fase di traduzione di un programma in L in una rappresentazione interna.

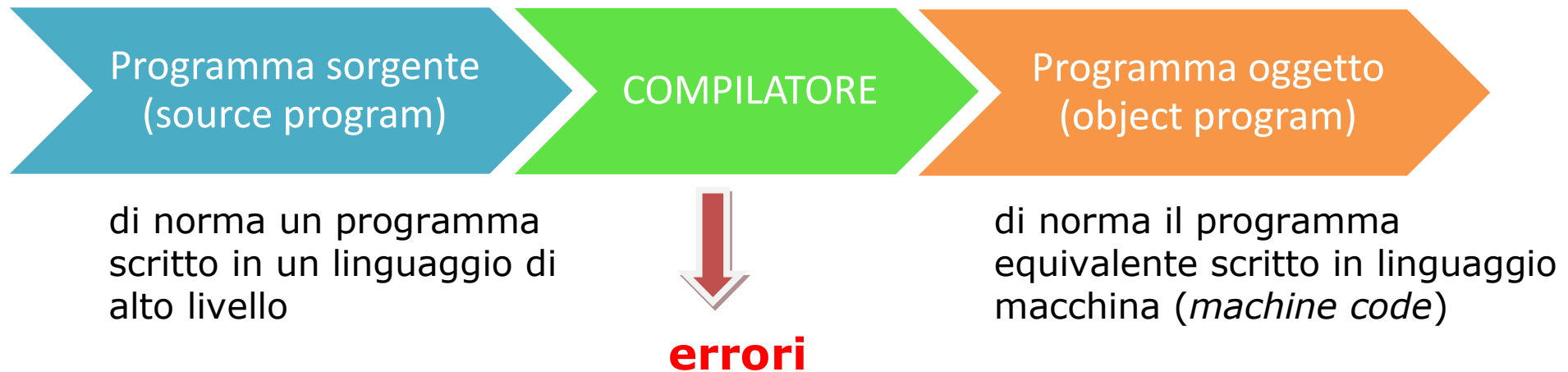
Implementazione di un Interprete

Sviluppo di un programma in L_{M_O} che simula la macchina M_I



Compilatori

I compilatori sono fondamentali nei moderni computer: essi agiscono come traduttori che consentono di trasformare un linguaggio di programmazione *human-oriented* in uno *computer-oriented*.



La maggior parte degli utenti vedono un compilatore come una "*black box*".

I compilatori consentono ai programmatori di ignorare i dettagli *machine-dependent* della programmazione.

TOKEN, PATTERN, LESSEMA

What's a Token?

- A syntactic category
 - In English:
noun, verb, adjective, ...
 - In a programming language:
Identifier, Integer, Keyword, Whitespace, ...

Un **token** è una coppia costituita da (*token name*, *optional attribute value*).

Un **pattern** è la descrizione della forma che possono assumere i **lessemi** di un token.

Un **lessema** è una sequenza di caratteri del *source program* che rispetta il **pattern** di un token ed è identificato dallo scanner come istanza di quel token.

Struttura di un compilatore

Ci sono due fasi fondamentali in un compilatore: **Analisi** e **Sintesi**

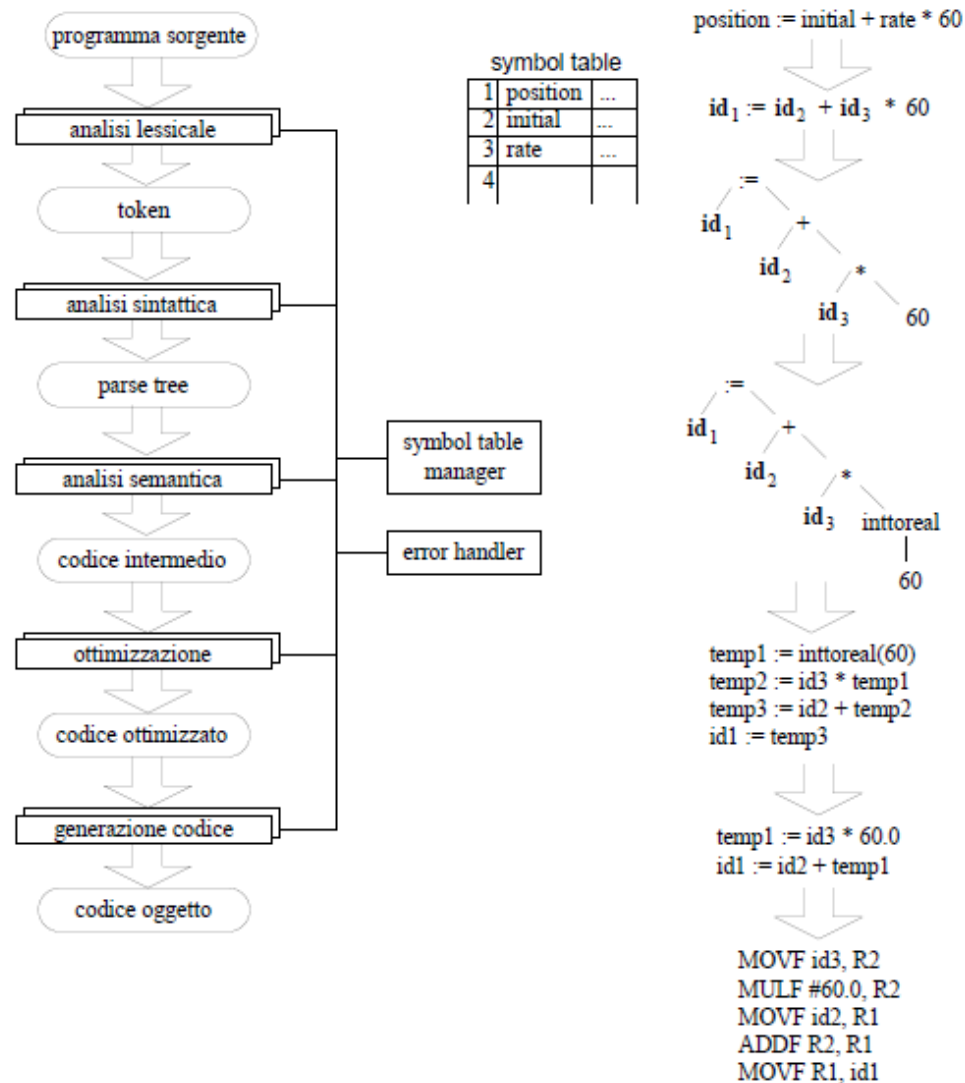
Nella fase di **analisi**, viene creata una rappresentazione intermedia del programma sorgente.

Sono parti di questa fase l'**Analizzatore lessicale** (trasforma il codice sorgente in una forma compatta e uniforme, costituita da **lessemi**, cioè insiemi di caratteri che hanno lo stesso significato (come identificatori, operatori, *keywords*, numeri, delimitatori, etc), raggruppati a loro volta in elementi lessicali detti **token**), l'**Analizzatore Sintattico** (il procedimento di costruzione della derivazione di una frase rispetto ad una data grammatica), l'**Analizzatore Semantico** (controllo del significato delle istruzioni presenti nel codice in ingresso) e il **Generatore di codice intermedio** (traduzione delle frasi del programma da un linguaggio di alto livello in una rappresentazione intermedia).

Nella fase di **sintesi**, viene creato, partendo dalla rappresentazione intermedia, il programma target equivalente.

Sono parti di questa fase il **Generatore di Codice** (il codice in rappresentazione intermedia è mappato nel codice della macchina target) e l'**Ottimizzatore** (il codice macchina viene analizzato e trasformato in codice equivalente ottimizzato per una specifica architettura).

Implementazione di un Compilatore



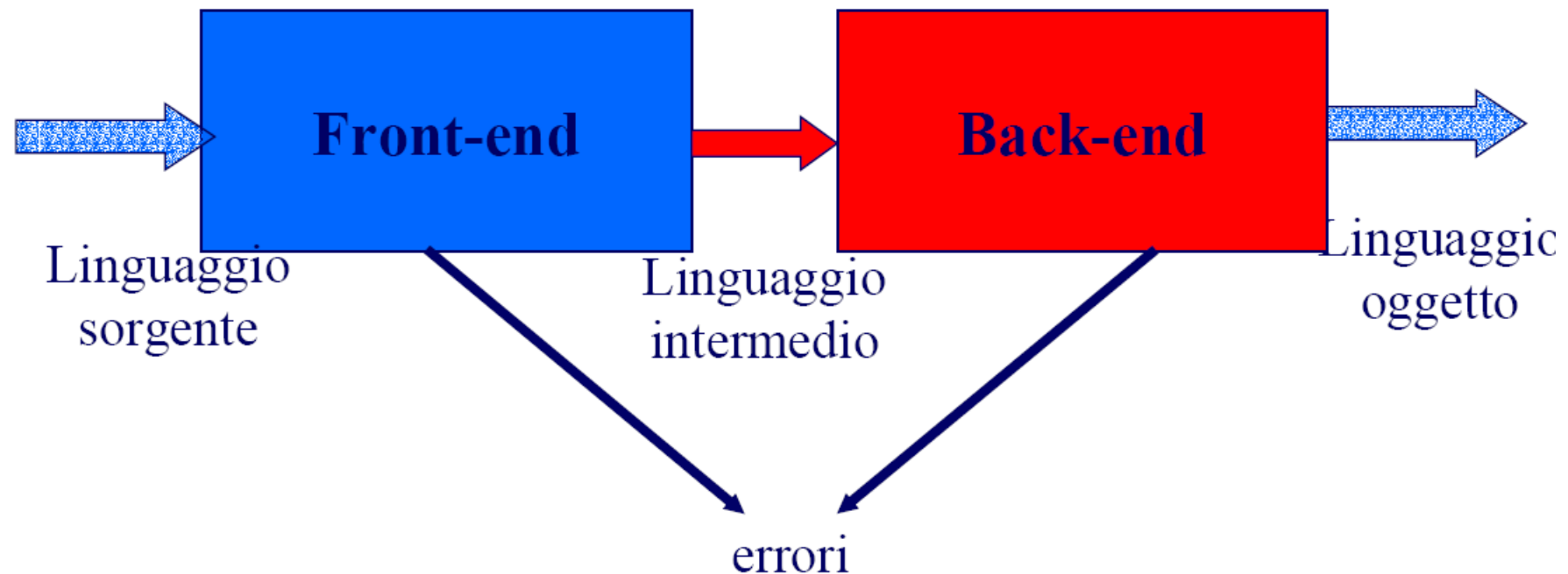
Ogni fase trasforma il programma sorgente da una rappresentazione in un'altra
 Nelle varie fasi si utilizzano: Gestore errori (**Error handler**) e Tabella simboli (**Symbol table**).

Compilatori a 2 passi: **FRONT-END** e **BACK-END**

I compilatori attuali dividono l'operazione di compilazione in due stadi principali: il **Front-end** e il **Back-end**.

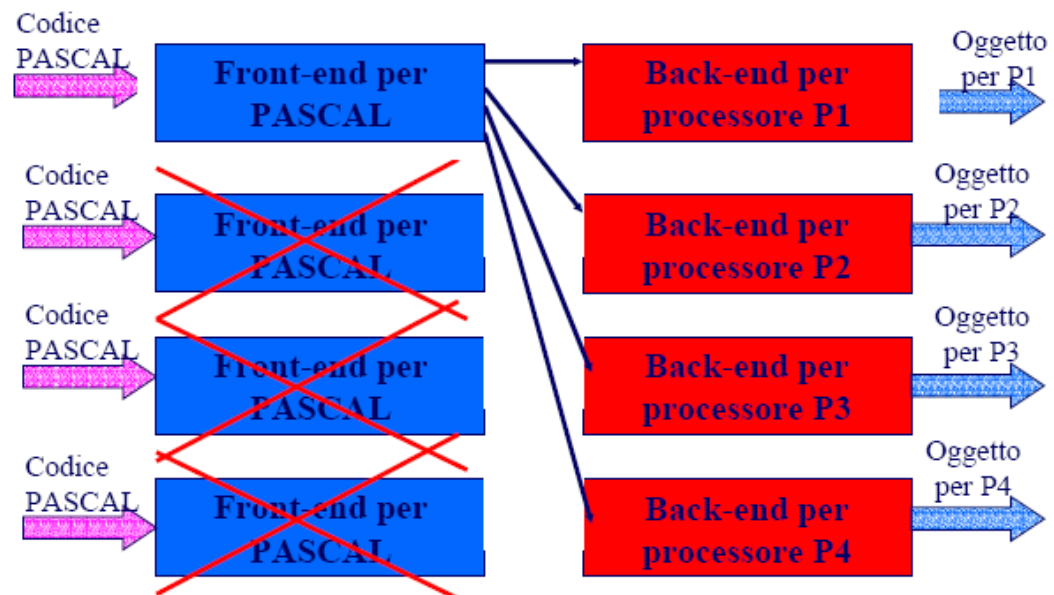
Il **Front-end** dipende dal linguaggio sorgente, ma è indipendente dalla macchina target. Il compilatore traduce il sorgente in un linguaggio intermedio.

Il **Back-end** è indipendente dal linguaggio sorgente, ma dipende dalla macchina target. Nello stadio di **back-end** avviene la generazione del codice oggetto e l'ottimizzazione.

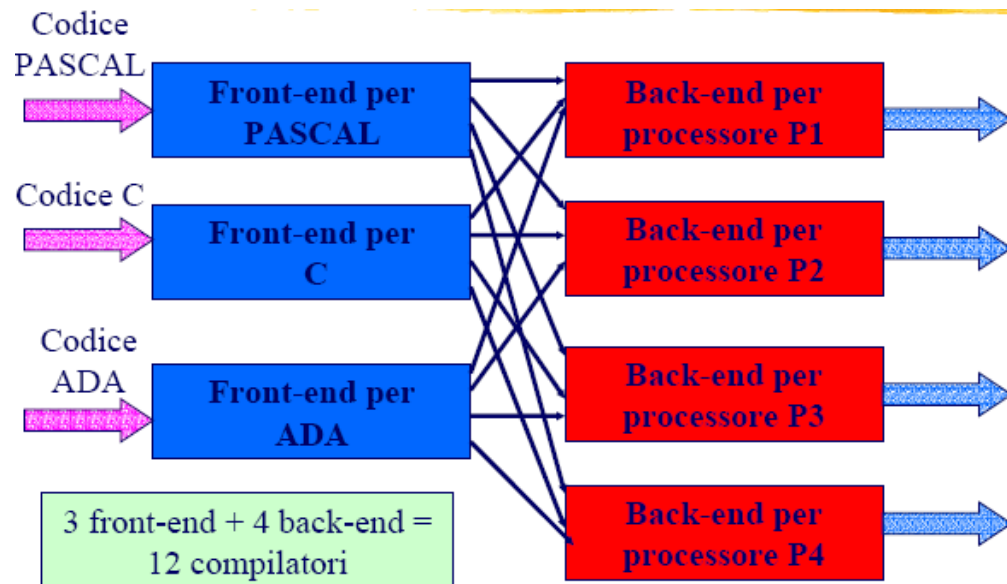


Front end e Back end si possono riutilizzare separatamente

Compilatori a due passi: il vantaggio del riutilizzo separato



Retargeting



**Multiple
Front-end**