Linguaggi di programmazione 1

Corso di Laurea in Informatica
Facoltà di Scienze M.F.N.
Università Federico II

aa. 2009-2010

Note per gli studenti

- **Programma e contenuto del corso sono concertati tra i docenti dei due gruppi**
- **Quindi, a parte personalizzazioni di stile, non ci sono differenze fra i due corsi nè sui compiti che farete nè sugli obiettivi che ci proponiamo**
- **Non sono permessi cambi di gruppo**
  - **Noi terremo in considerazione solo la lista di quelli ufficiali per l'anno in corso**
- **Ciascun docente farà ricevimento ed esami ai soli membri del proprio gruppo**

Materiale didattico

- Libri di testo:
  a) Dershem - Jipping. Programming languages: structures and models.
  b) Wampler. The essence on object oriented programming with Java and UML.
  c) Fowler. UML distilled.
  d) Eckel. Thinking in Java.
  e) Gabbrielli, Martini. Linguaggi di programmazione: Principi e paradigmi
- Note distribuite dai docenti

Sito del docente

- http : // cs.na.infn.it / ~bonatti
  - Oppure Google *Piero Bonatti*
- Seguire il link “Teaching”

- Materiale protetto per motivi di copyright
  - Accesso con *secpriv* e *pgpp3p*
- Tenete d'occhio le *assenze programmate* e gli annunci per i corsi

Programma di L1 aa. 08-09

- Parte prima (concetti generali) – ~10 lezioni
  1. Paradigmi dei linguaggi di programmazione.
  2. Il modello imperativo.
  3. Il modello ad oggetti.
  4. Progettazione orientata ad oggetti e UML.
  5. Prova scritta intercorso.

- Parte seconda (Java) – ~16 lezioni
  1. Studio dei costrutti fondamentali: identificatori, parole chiavi, tipi; espressioni e controllo di flusso; ereditarietà; overloading e overriding;classi astratte , polimorfismo, arrays
  – Studio dei costrutti più avanzati: qualificatori di classi, metodi, attributi; interfacce; classi interne; gestione degli errori:eccezioni; programmi text-based; (threads; networking: molto improbabile).

Modalità d'esame

- Prova intercorso sulla prima parte del corso
- Esame “regolare”:
  - 2 scritti: su Java e sulla prima parte (equivalente a prova intercorso)
  - In caso di quasi sufficienza, o casi per noi dubbi, un orale

Modalità d'esame

- Differenze dagli anni precedenti
  - In caso di grave insufficienza non sarà possibile sostenere altre prove scritte fino alla *sessione successiva*
  - Non sarà possibile tenere frammenti di uno scritto e rifarne solo alcune parti
    - Resta possibile mantenere il voto di una prova scritta fino alla fine dell'anno accademico

Chi sono come trovarmi

- Piero Bonatti.
- Studio 0F34, edificio di biologia strutturale.
- Email :bonatti@na.infn.it
- Il ricevimento è su appuntamento via e-mail.

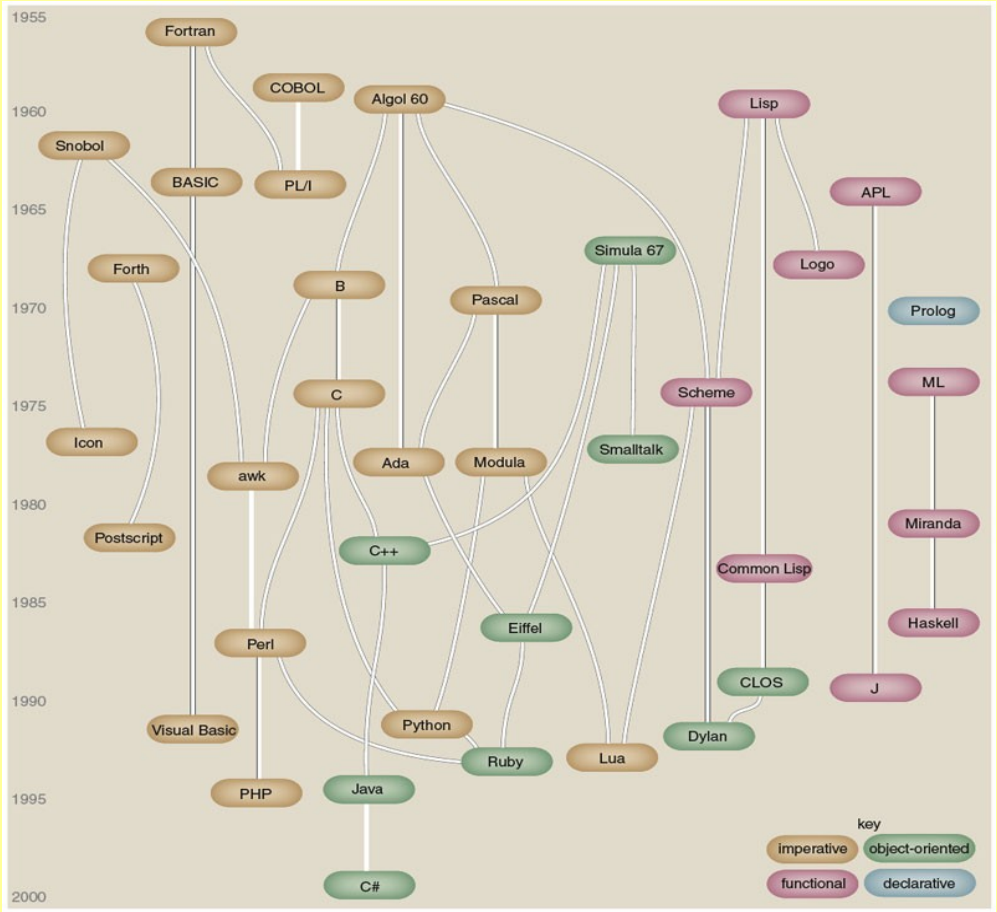
Obbiettivi del corso

- Spiegare le differenze tra i vari paradigmi di programmazione - in particolare dei paradigmi imperativo e ad oggetti - e sul loro impatto sullo stile di soluzione dei problemi.
- Capacità media di progettare ad oggetti.
- Capacità media di programmare in Java.

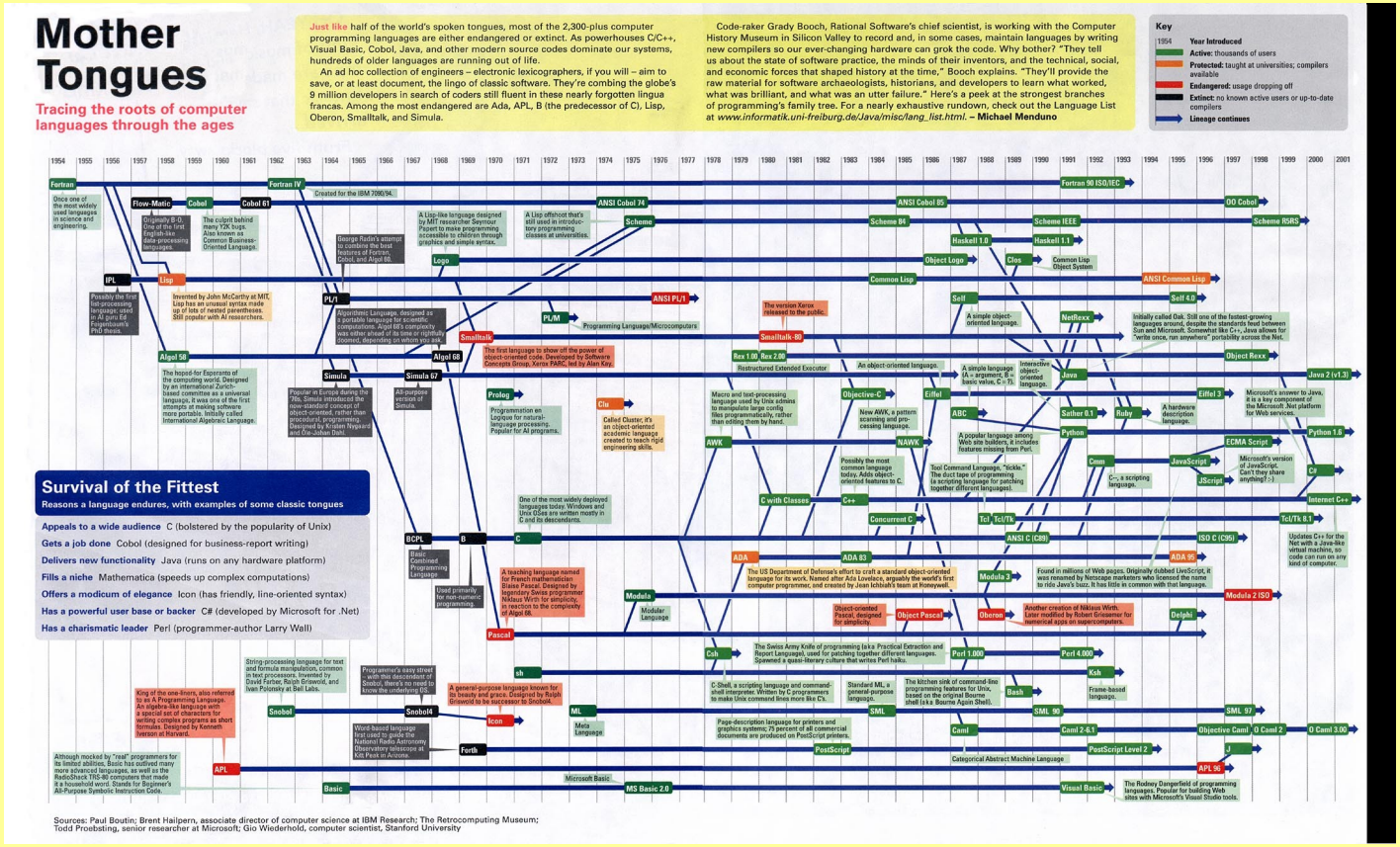
Obiettivi del corso a lungo termine

- Migliorare l'abilità nel risolvere i problemi.
- Imparare a usare meglio i linguaggi di programmazione.
- Imparare a scegliere più intelligentemente, in dipendenza del problema, il linguaggio di programmazione.
- Aumentare la capacità di imparare linguaggi di programmazione (che sono TANTI!).

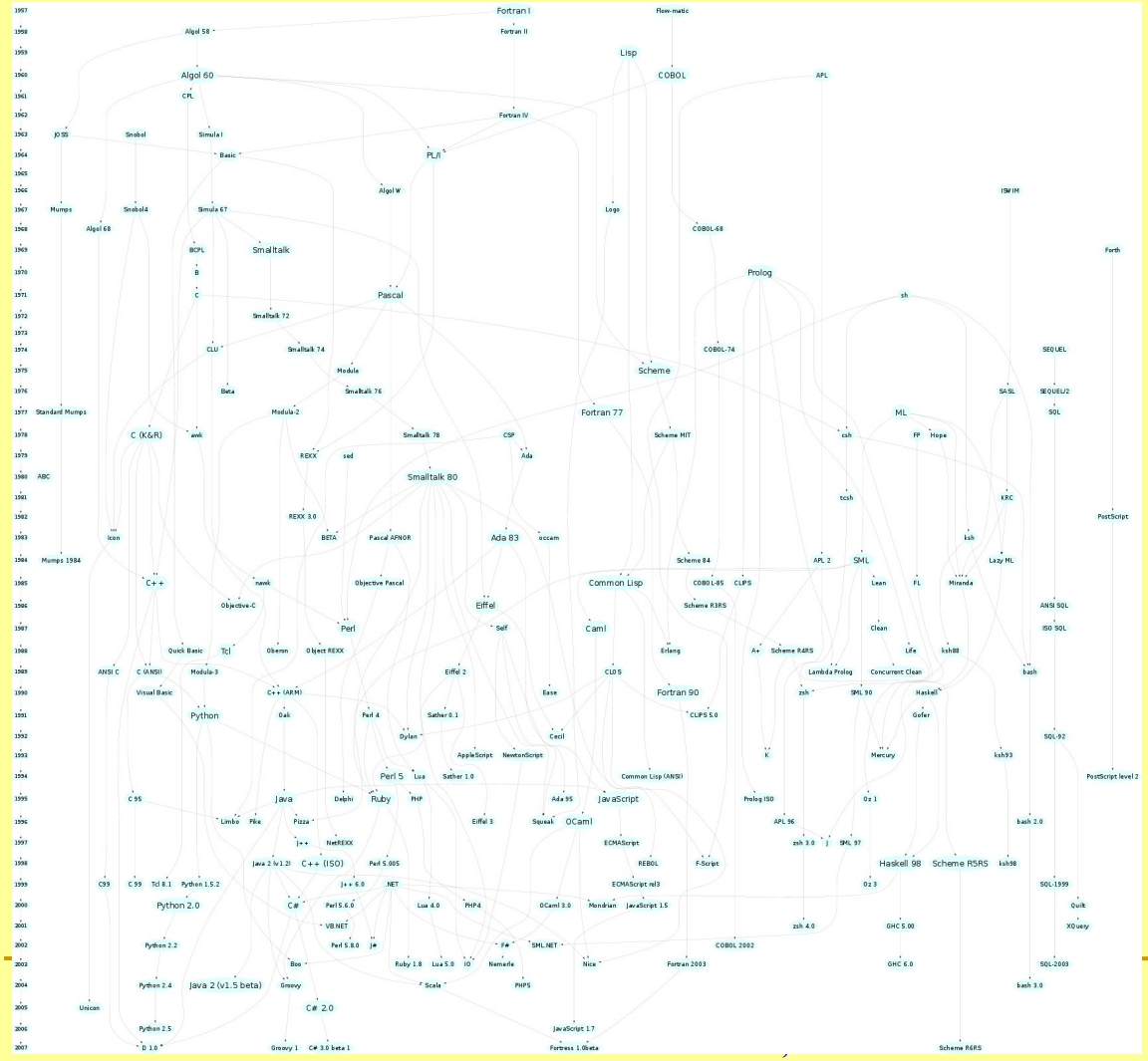
Quanti sono i linguaggi di programmazione? ~40?



Quanti sono i linguaggi di programmazione? ~80?



Quanti sono i linguaggi di programmazione?



- Questi sono ~100
- Secondo alcuni 2500
- Secondo altri >8K

- Come orientarsi?
- Come usarli *bene*?
- Come apprendere in fretta quelli nuovi?

- Occorre una comprensione astratta delle caratteristiche dei linguaggi per coglierne somiglianze/differenze

- e per comprendere lo scopo di ciascun costrutto (ovvero i principi del language design)

Sommario

- Terminologia
- Breve storia
- Paradigmi
- Macchina astratta
- Meccanismi di traduzione
- Processo di compilazione
- Caratteristiche dei linguaggi
- Criteri di scelta di linguaggi
- Modello imperativo
- Legami e data object
- Modifiche di un legame
- Tipi
- Esempio: il puntatore
- Ambito di validità dei legami
- Legame di nome
- Ereditarietà dei legami
- Ereditarietà statica
- Legame di locazione
- Esempio di allocazione dinamica
- Bibliografia
- Esempi

Terminologia

- **Linguaggio di programmazione**: un linguaggio che è usato per esprimere (mediante un programma) un processo con il quale un processore può risolvere un problema.
  - Con qualche ulteriore vincolo – vedi slides succ.
- **Processore:** intendiamo l' architettura hardware relativa all' esecuzione del processo descritto dal programma; non si deve necessariamente intendere come un singolo oggetto, ma come un' architettura di elaborazione.
- **Programma:** è l'espressione codificata di un processo.

Potenza di calcolo ed espressività

- **Linguaggio di programmazione**: un linguaggio che è usato per esprimere un processo con il quale un processore può risolvere un problema.
  - Con qualche ulteriore vincolo
- **Linguaggi general purpose**
  - Possono essere usati per affrontare qualunque problema/applicazione
  - SQL è un linguaggio di programmazione?
  - HTML è un linguaggio di programmazione?

Potenza di calcolo ed espressività

- SQL è un linguaggio di programmazione general purpose? **NO**
  - Le query sono calcolabili, il risultato è *decidibile*
- HTML è un linguaggio di programmazione general purpose? **NO**
  - Simile
- Un linguaggio di programmazione è di solito computazionalmente completo (Turing equivalente)
  - Deve poter esprimere *tutte* le funzioni calcolabili
  - Esistono diversi modi per raggiungere questo obbiettivo (corrispondono a diversi *paradigmi*)

Correlare paradigmi a linguaggi

- Uno specifico linguaggio di programmazione implementa un modo di pensare il processo di computazione (**paradigma di computazione**)'.
- Un ***paradigma di computazione*** definisce strutture e fattori che accomunano linguaggi di programmazione apparentemente diversi e che differiscono sostanzialmente dai linguaggi che appartengono ad altri paradigmi.
  - Similitudini di sostanza sotto differenze sintattiche

Paradigmi in pillole

- **Imperativo:** Esecuzione sequenziale che realizza, tramite lo statement di assegnazione, modifiche della memoria: Si pensa il programma come **sequenza di azioni.** Servono costrutti per **iterazioni**
- **Funzionale:** I programmi sono **funzioni.** I "dati" sono gli argomenti su cui si valutano le funzioni. Niente variabili! Niente assegnamenti!
  - dunque niente iterazioni (sarebbero inutili). *Rimpiazzate con?...*
- **Logico:** I programmi sono **teorie logiche.** L'esecuzione è la prova nella teoria-programma del teorema il cui enunciato è il problema.
  - Si dice *cosa* si vuole ottenere, non *come* ottenerlo
  - Gli algoritmi possono essere *nondeterministici* (la valutazione no...)
- **Orientato ad oggetti:** Il programma è un insieme di **classi** ognuna della quali definisce il *tipo* (cioè operazioni e dominio dei valori possibili) per lo "stato" degli oggetti sue istanze. L' esecuzione è uno scambio di messaggi fra oggetti che induce l'esecuzione di "operazioni" da parte degli oggetti riceventi e possibilmente un *cambiamento del loro stato*.
  - Nei linguaggi più comuni le operazioni somigliano al modello imperativo
  - Ma esistono linguaggi O.O. di sapore funzionale e logico
- **Parallelo:** Programmi che descrivono entità distribuite che sono **eseguite contemporaneamente**

Paradigmi: qualche esempio informale

Approccio imperativo

function definition

```
function factI (n)
  local accumulator = 1
  for i = 1,n do
    accum = accumulator*i
  end
  return accum
end
```

trace of execution

```
factI(4):
          accumulator = 1
i = 1     accumulator = 1 * 1
i = 2     accumulator = 1 * 2
i = 3     accumulator = 2 * 3
i = 4     accumulator = 6 * 4
return 24
```

Approccio funzionale

Notare come *n* assuma il ruolo di *i* e...

ricorsione invece di iterazione

function definition

```
function factR (n)
  if n == 1 then
    (return)1
  else
    (return)n*factR(n-1)
  end
end
```

trace of execution

```
factR(4) =
4 * factR(3) =
      3 * factR(2) =
            2 * factR(1) =
                        1
                        1
                  1
            2
      6
24
```

Paradigmi: qualche esempio

- La funzione *member*
  - Risponde alla domanda: l'elemento X appartiene alla lista L?
- Assumiamo per le liste le seguenti funzioni
  - head(L): restituisce il primo elemento della lista
    - Es: head([1,2,3]) = 1
  - tail(L): restituiamo la sottolista ottenuta rimuovendo il primo elemento da L
    - Es: tail([1,2,3]) = [2,3]
  - empty(L): restituisce *true* se L è vuota, *false* altrimenti

Paradigmi: qualche esempio

- La funzione *member*, versione procedurale

```
procedure member(X,L)
    local L1 = L
    while not empty(L1) and not X=head(L1)
        do L1 = tail(L1)
    if not empty(L1) then return true /* dev'essere X=head(L1) */
    else return false
```

- Esempi di esecuzioni
  - member(2,[1,2,3]) restituisce *true*
  - member(0,[1,2,3]) restituisce *false*

Paradigmi: qualche esempio

- La funzione *member*, procedurale, in C

```
bool member(X,L) {
    List L1 = L;
    while( ! empty(L1) && ! X=head(L1))
        L1 = tail(L1);
    if (! empty(L1))  return true;
    else return false;  }
```

- La struttura del programma resta pressochè identica

Paradigmi: qualche esempio

- La funzione *member*, versione funzionale

```
function member(X,L)
    if  empty(L)  then  false
    else if  X == head(L)  then true
    else  member(X, tail(X))
```

- NB: niente variabili, niente assegnamenti
- Esempi di esecuzioni: come prima
  - member(2,[1,2,3]) restituisce *true*
  - member(0,[1,2,3]) restituisce *false*

Paradigmi: qualche esempio

- Si potrebbe usare il C in stile funzionale:

```
bool member(X,L) {
    return (empty(L)) ?  false :
                (X == head(L))?  true :
                   member(X, tail(X)) }
```

- Verificare similitudine strutturale con il codice della slide precedente
- Cosa non abbiamo mostrato del paradigma funzionale:
  - Soprattutto funzioni anonime, di ordine superiore, e particolari trattamenti del sistema di tipi

Paradigmi: qualche esempio

- La funzione *member*, versione logica

  member(X,[X|L]).

  member(X,[Y|L]) <= member(X,L).

- Da notare:
  - I parametri di *member* possono essere *pattern*
- Esempi di esecuzioni: come prima
  - member(2,[1,2,3]) restituisce *yes (true)*
  - member(0,[1,2,3]) restituisce *no (false)*
- *E inoltre...*

Paradigmi: qualche esempio

- La funzione *member*, versione logica

  member(X,[X|L]).

  member(X,[Y|L]) <= member(X,L).

- Esempi di esecuzioni *di nuovo tipo* (~query)
  - member(X,[1,2,3]) restituisce
    - X=1; X=2; X=3; no (si comporta come un *generatore*)
  - member(1,L) restituisce
    - L=[**1**|L_0] ; L=[Y_1,**1**|L_1] ; L=[Y_1,Y_2,**1**|L_2] ; .... (!!)
- *Invertibilità: nessuna distinzione tra input e output; un solo predicato ma molte funzioni*

Cosa dimostra?

- Che il paradigma del linguaggio può influenzare *fortemente* il modo in cui si risolve un problema
- Non è l'unico aspetto determinante: altri importanti sono
  - Sistema di tipi supportato
  - Supporto delle eccezioni
  - Modello di concorrenza e sincronizzazione

Breve storia incompleta: solo nomi

1954 FORTRAN (FORmula TRANslation)	USO SCIENTIFICO
1960 LISP (LISt Processing)	CONCEPITO PER INTELL: ARTIF:
1960 COBOL	(Common Business Oriented Language)
ALGOL 60 (Algorithmic Oriented Language)	USO SCIENTIFICO
PL/1 (Programming Language 1)	"GENERAL PURPOSE"
Simula 67	SIMULAZIONE
ALGOL 68	"GENERAL PURPOSE"
PASCAL	
APL	
BASIC	
1970/80 PROLOG	
SMALLTALK	
C	
MODULA/2	
ADA	
Etc…….	

Breve storia dei concetti principali introdotti dai progenitori dell'oggi

Fortran: nato per manipolazione algebrica,introduce: variabili, statment di assegnazione,concetto di tipo , subroutine,iterazione e statment condizionali,go to, formati di input e output.

- Gestione solo statica della memoria, no ricorsione, no strutture dinamiche, no tipi definiti da utente

Cobol: indipendenza dalla macchina,e statment "English like". Orientato ai database. Introduce il record.

Algol60: indipendenza dalla macchina e definizione mediante grammatica(bakus-naur form),strutture a blocco, supporto generale dell'iterazione e ricorsione, ...

Breve storia dei concetti principali introdotti dai progenitori dell'oggi

Lisp: primo vero linguaggio di manipolazione simbolica,paradigma funzionale, non c'è lo statment di assegnazione,e quindi concettualmente non c'è "il valore" ovvero l' idea di cambiare lo stato della memoria.

non c'è differenza concettuale fra funzione e dato: dipende dall'uso.

- Prima versione essenzialmente non tipata

Prolog: primo (e principale) linguaggio di programmazione logica (paradigma logico). Tra le caratteristiche innovative: invertibilità, programmazione in stile nondeterministico (generate and test)

- Essenzialmente non tipato; estensioni (tipi e altro) mediante metaprogrammazione

Concetti principali introdotti dai progenitori dell'oggi

Simula 67: classe come incapsulamento di dati e procedure, istanze delle classi(oggetti):anticipatorio del concetto di tipo di dato astratto implementati in Ada e Modula2, e dl concetto di classe di Smalltalk e C++.

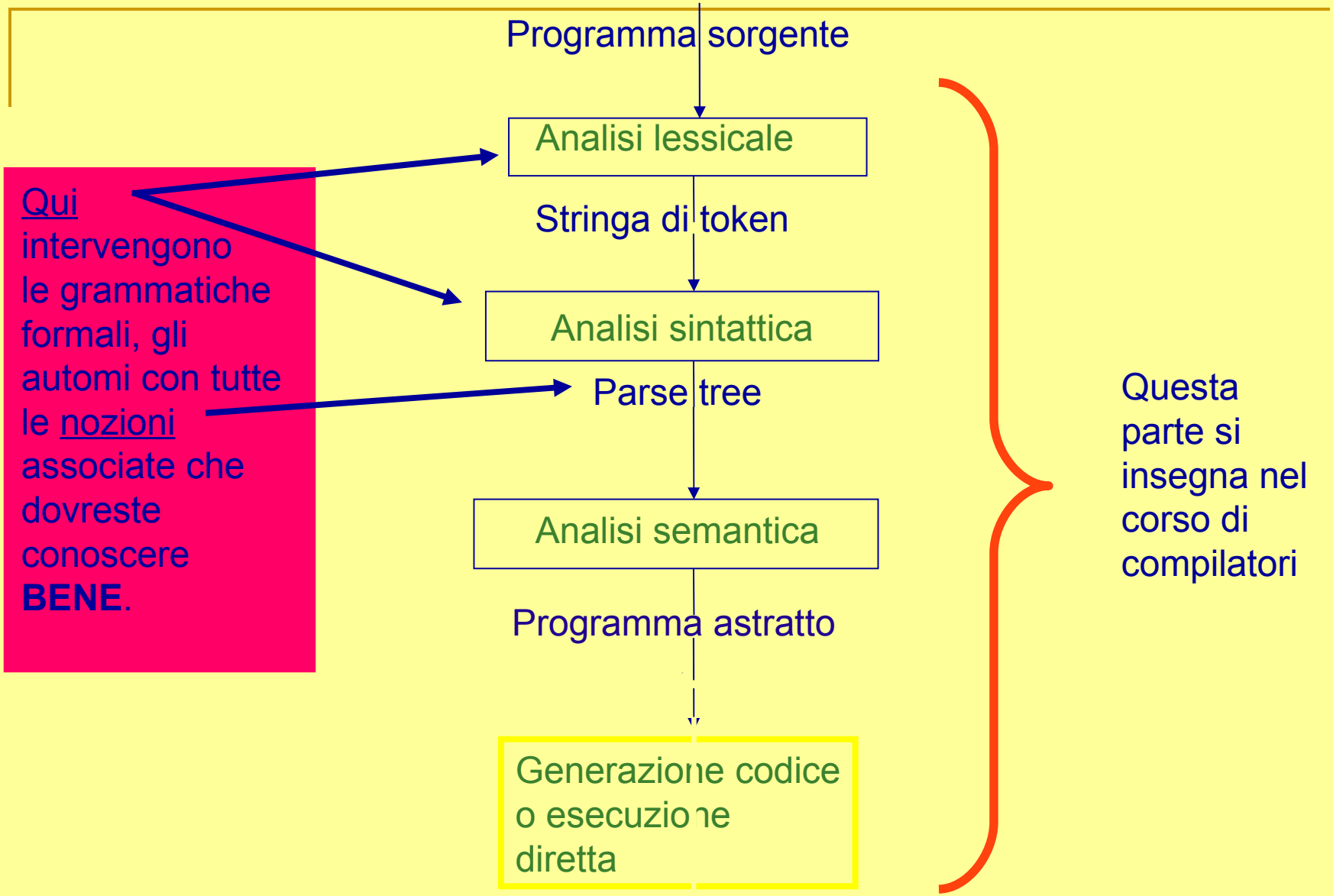
PL/1: abilità ad eseguire procedure specificate quando si verifica una condizione eccezionale;"multitasking",cioè specificazione di tasks che possono essere eseguiti in concorrenza.

Pascal: programmazione strutturata, tipi di dato definiti da utente,ricchezza di strutture dati…

- Ma ancora niente encapsulation; si dovrà aspettare Modula...

Scelto il linguaggio, enunciato il problema, dato l' algoritmo scritto il programma …

- Cosa succede??


Programma sorgente
Analisi lessicale
Stringa di token
Analisi sintattica
Parse tree
Analisi semantica
Programma astratto
Generazione codice o esecuzione diretta
Qui intervengono le grammatiche formali, gli automi con tutte le nozioni associate che dovreste conoscere **BENE**.
Questa parte si insegna nel corso di compilatori

Meccanismi di traduzione : importante comprendere le differenze fra di essi

- **Interpreti**: traducono ed eseguono un costrutto alla volta
  - Ad es. Scripting languages, Python, Javascript, PHP,...
- **Compilatori**: prima traducono l'intero programma; poi eseguono la traduzione, detta *programma oggetto*. (Il codice oggetto può essere eseguito anche più volte senza ricompilare).
  - Ad es. C/C++, Pascal, ...
- **Soluzioni ibride** (ad es. Prolog, ML, Java, C#)
  - Compilazione in un linguaggio macchina *virtuale* (intermedio)
  - Che viene interpretato da un programma (macchina virtuale)

Approfondiamo un poco il concetto di macchina astratta

- Dato un linguaggio di programmazione L, una **macchina astratta** per L (in simboli, M_L) è un qualsiasi insieme di strutture dati e algoritmi che permettano di memorizzare ed eseguire programmi scritti in L.

Elementi di una macchina astratta

- Essenzialmente memoria e interprete:

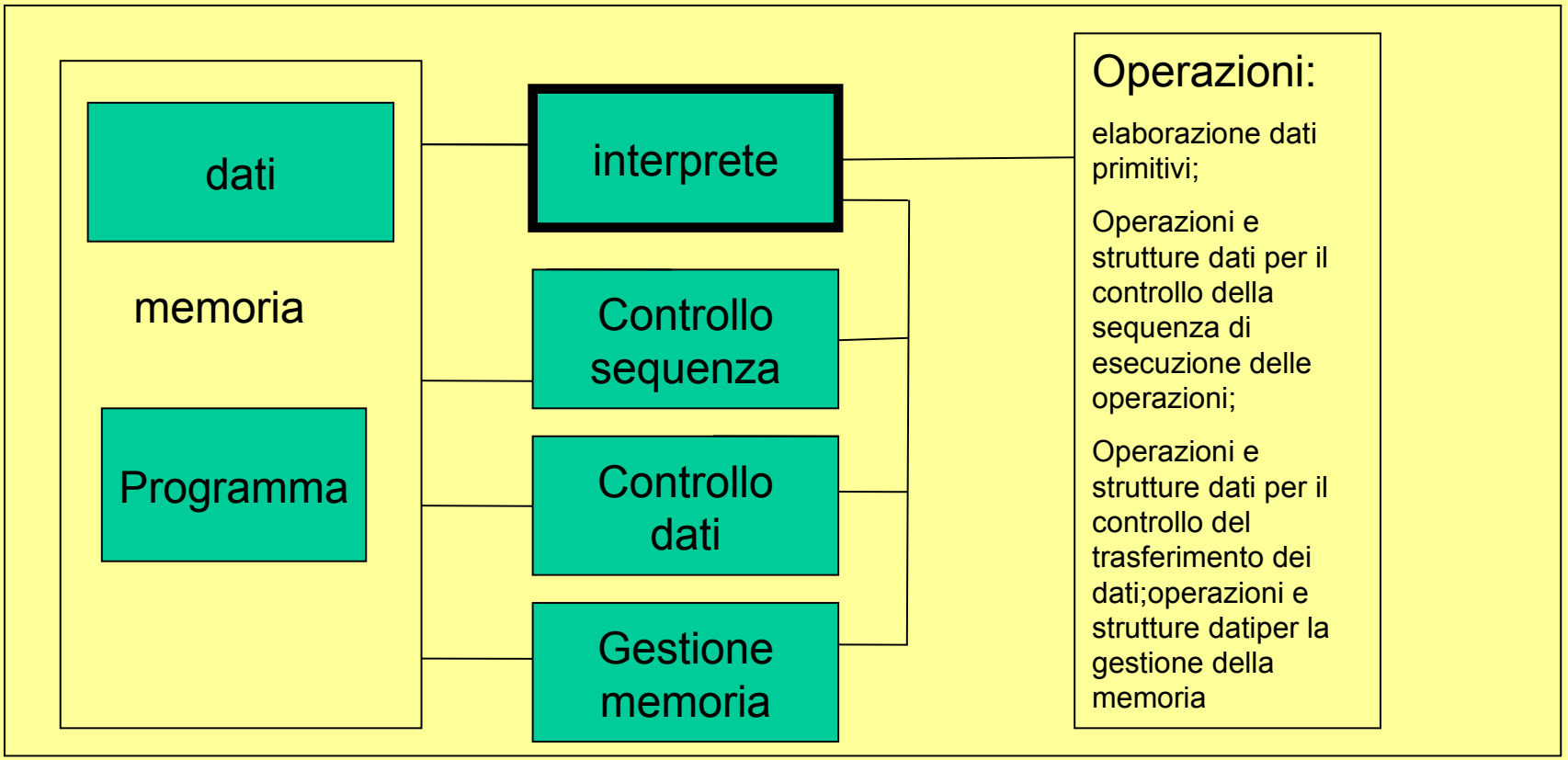
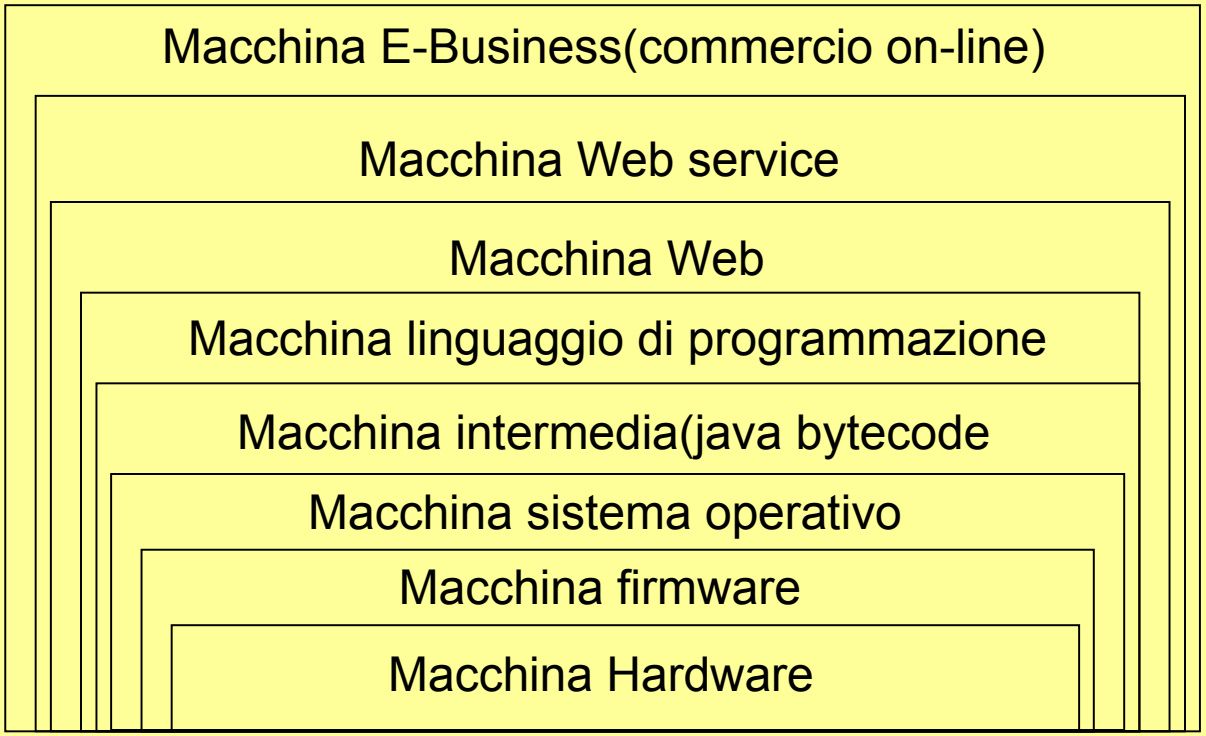
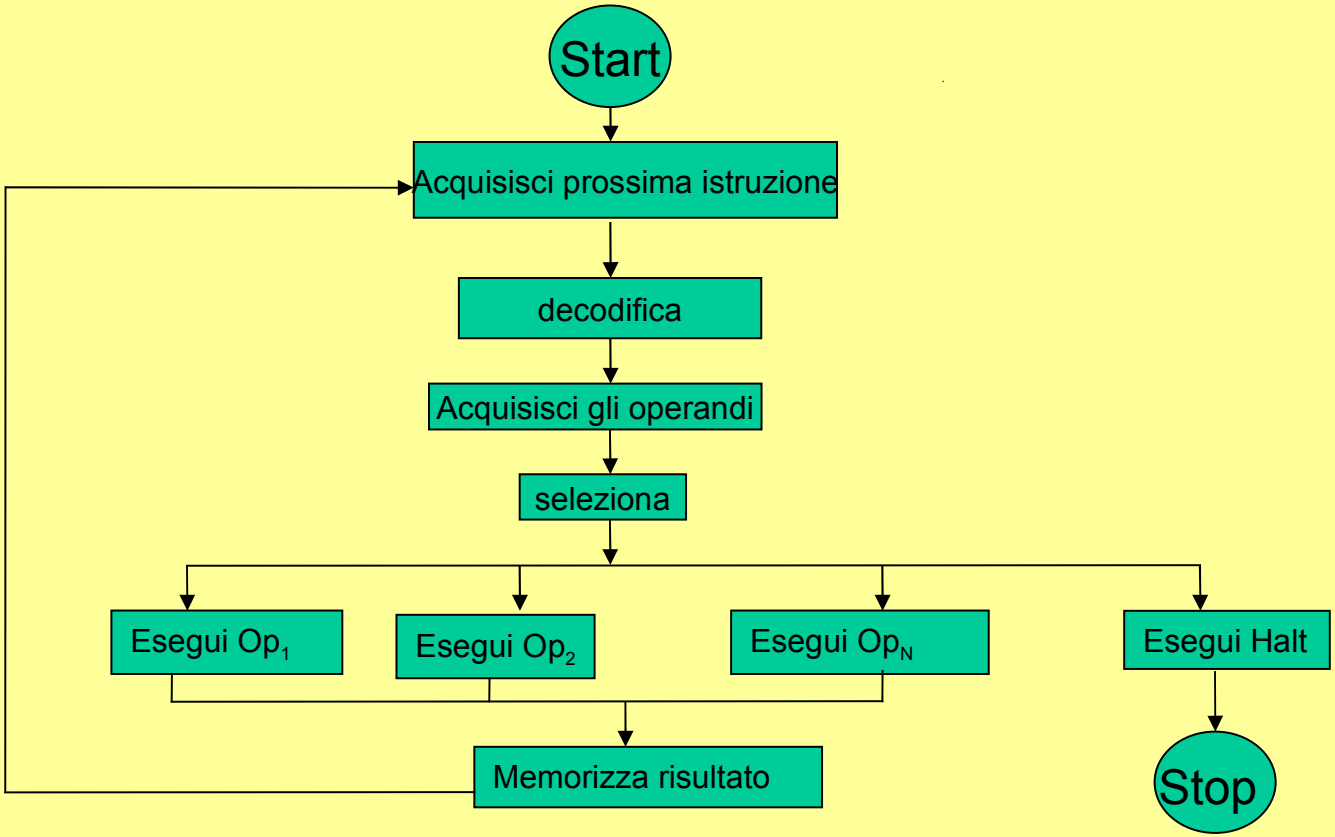


Fig.Struttura macchina astratta

Scommettiamo che avete già l' esperienza di "macchina Astratta"?



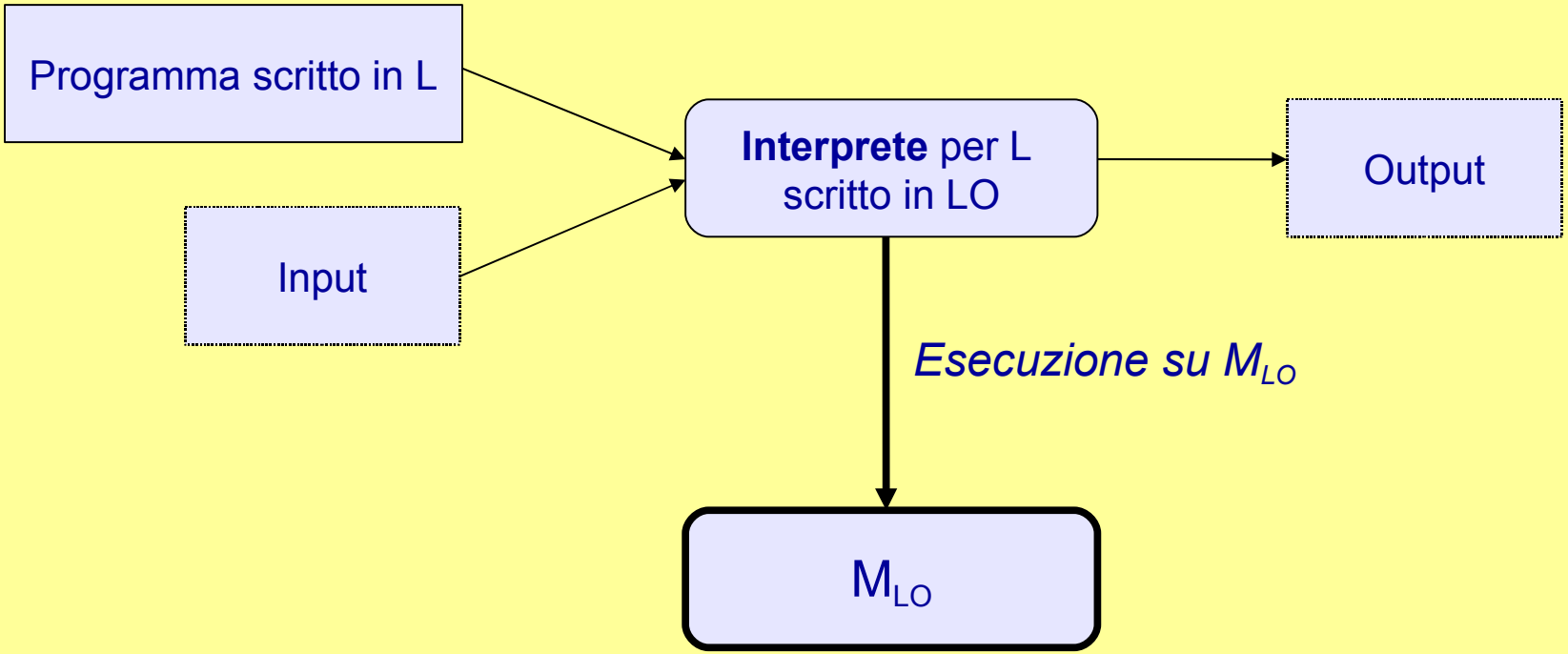
Ciclo esecuzione interprete



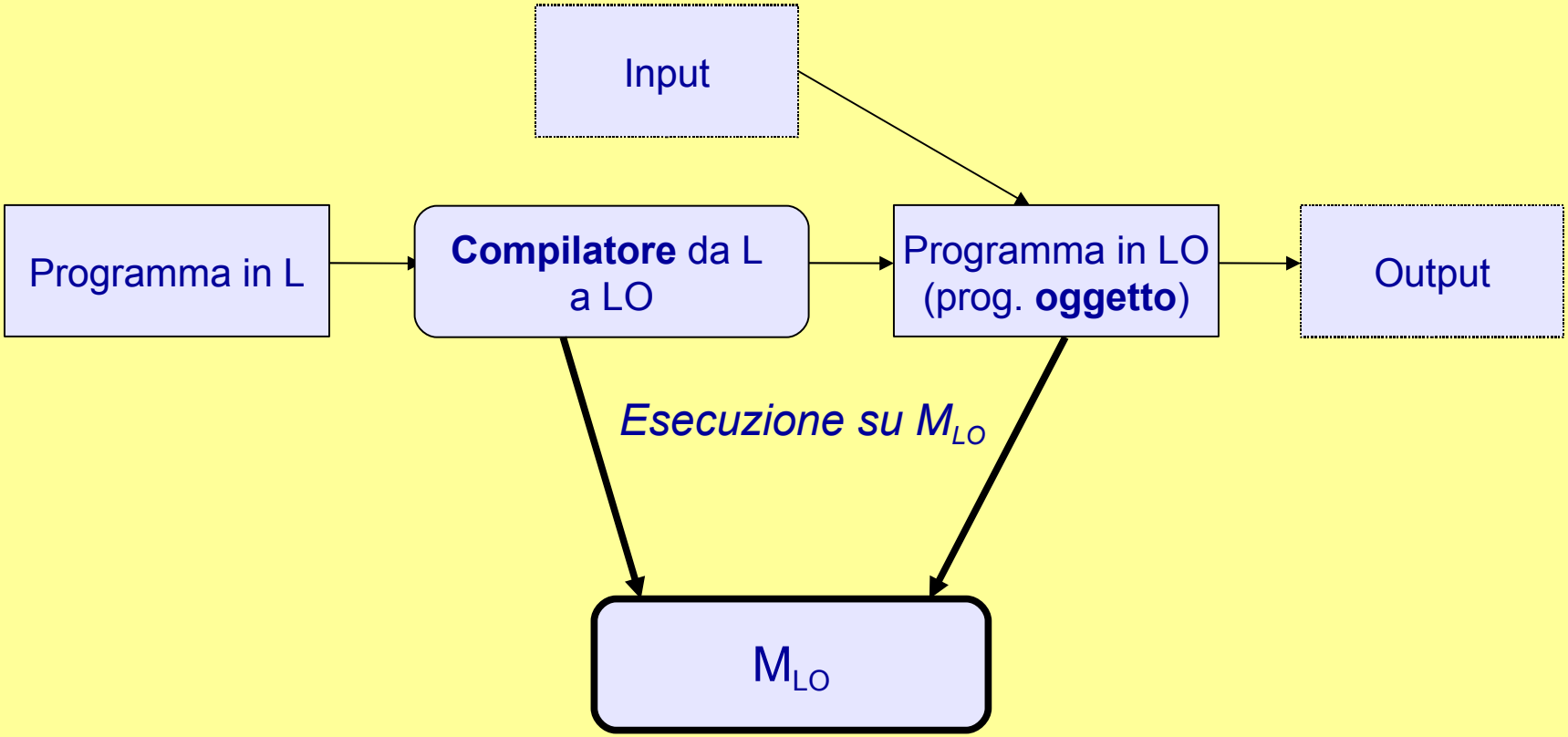
Possibili tecnologie per la realizzazione di una macchina astratta

- Hardware
  - Si era pensato per Lisp e Prolog
  - Costo/prestazioni non conveniente
- Firmware
  - Soluzione spesso adottata per flessibilità o semplicità/economicità di progetto
- Software
  - Ad es. macchina astratta Java, o Warren Abstract Machine (Prolog)

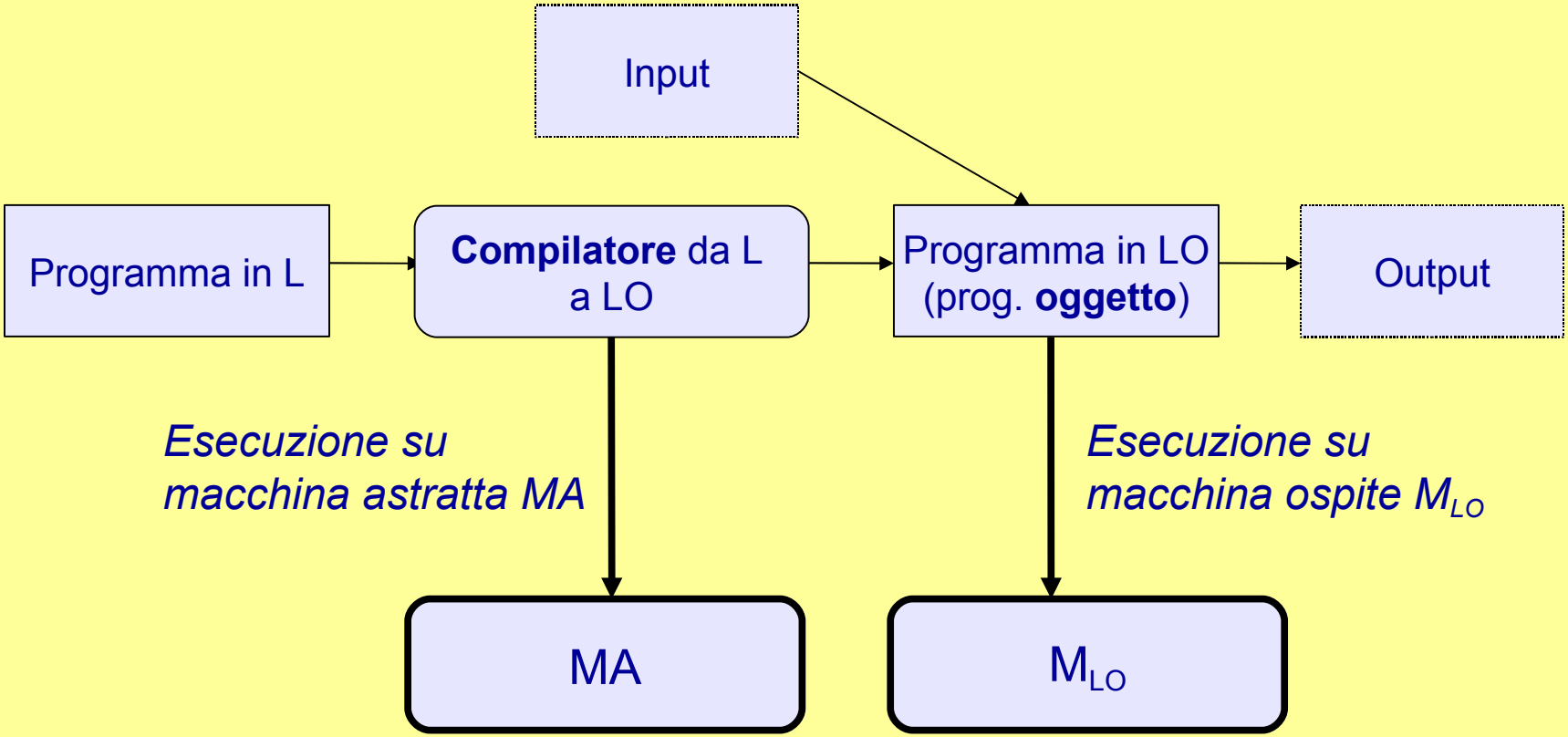
Implementazione interpretativa pura



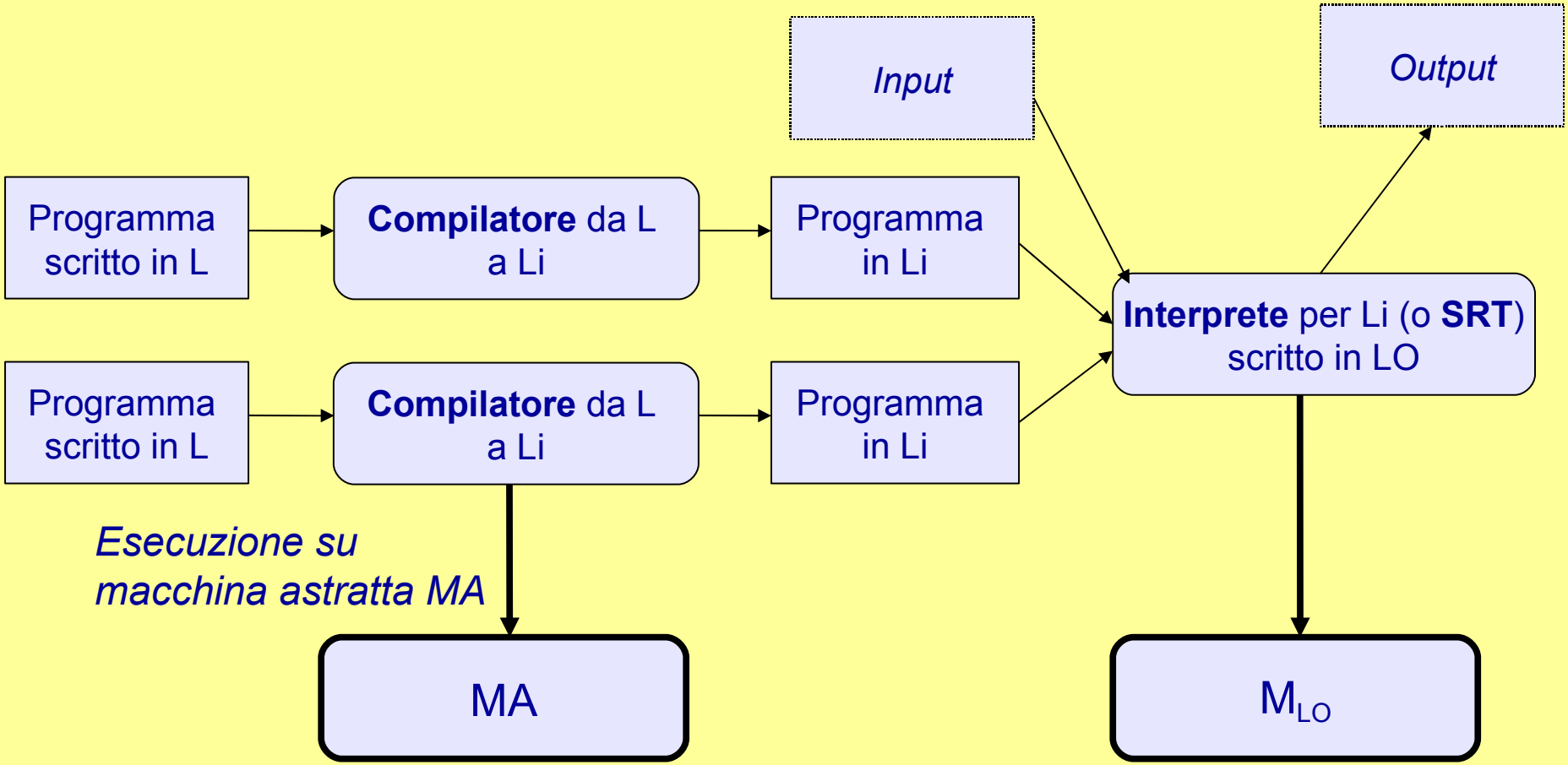
Implementazione compilativa pura – caso più comune



Implementazione compilativa pura – caso più generale (ad es. Cross compilation)



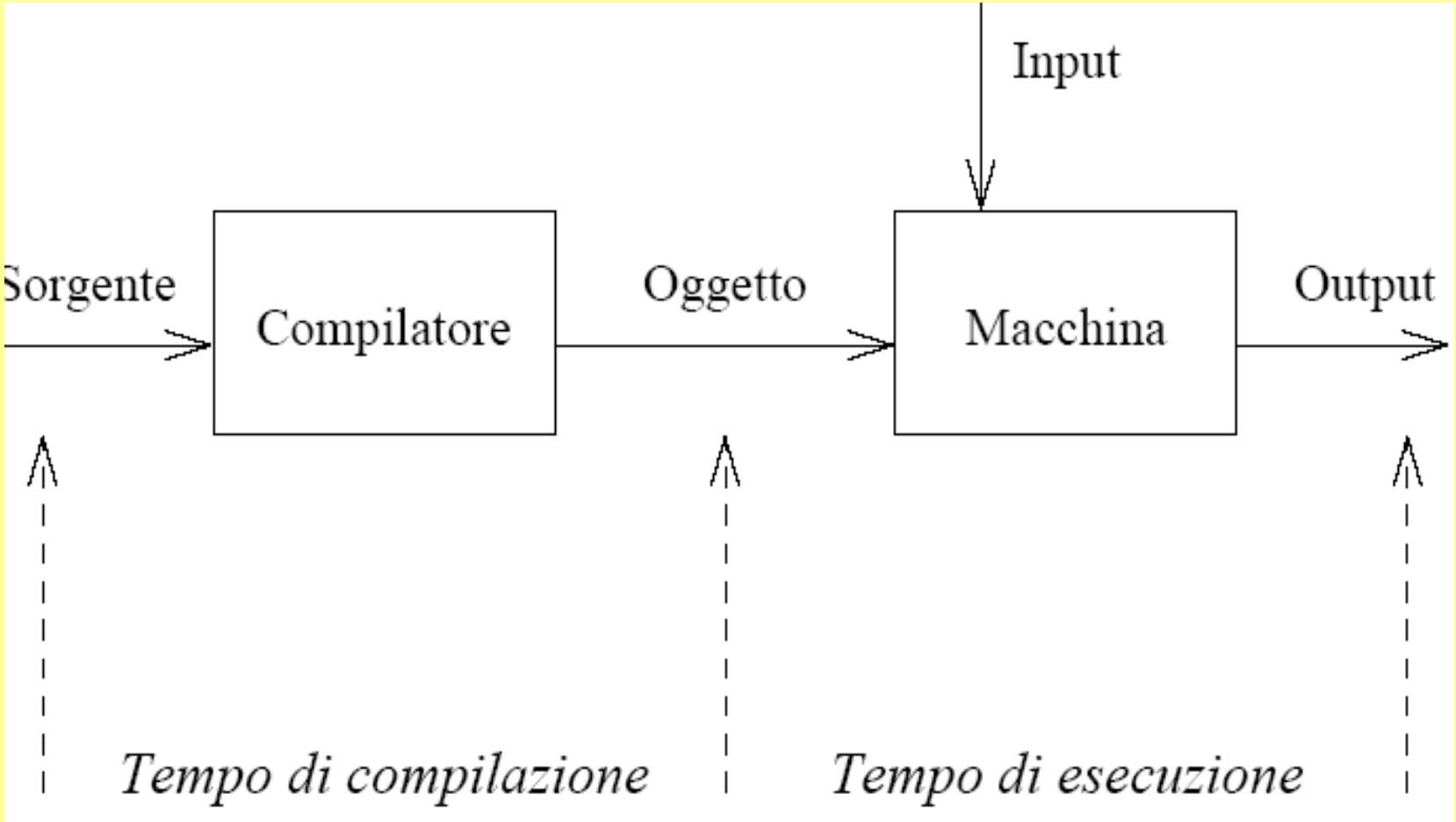
Implementazione con macchina intermedia



Supporto a run time (SRT)

- Funzionalità aggiuntive (risp. A M_{LO})
  - Funzioni di basso livello / interfacce col S.O.; ad es. Funzioni di I/O
  - Gestione della memoria; garbage collection; gestione dell'heap; gestione dello stack
- Non necessariamente una macchina astratta radicalmente diversa
  - A volte solo un pacchetto di funzioni o librerie aggiunte automaticamente al codice oggetto

Tempi di compilazione e di esecuzione



Meccanismi di traduzione : importante comprendere le differenze fra di essi

- **Interpreti**: traducono ed eseguono un costrutto alla volta
  - PRO: alleggerisce debug (ma i tipi contano molto!), *just-in-time programming; portabilità; mobilità*
- **Compilatori**: prima traducono l'intero programma; poi la traduzione. (Il codice oggetto può essere eseguito anche più volte senza ricompilare).
  - PRO: velocità di esecuzione finale; molti controlli possono essere fatti una volte per tutte a tempo di compilazione; sono possibili altre ottimizzazioni
- **Soluzioni con macchina intermedia**
  - Cercano di ottenere i benefici di entrambe le soluzioni

Meccanismi di traduzione : importante comprendere le differenze fra di essi

- **Interpretato, Compilato o Ibrido?**
- Scelta “strategica”
  - Non dipende dalla struttura del linguaggio quanto dall'uso
  - Ad es. PHP viene talvolta compilato (lato server)
- **In linea di principio per ogni linguaggio si può realizzare sia un interprete che un compilatore, che una soluzione ibrida**

Terminologia & verifica

- *Input*
  - Da command line; attraverso apposite istruzioni
  - Gli input vanno al compilatore? Al programma oggetto? All'interprete? Dipendono dalla particolare esecuzione?
- *Tempo di compilazione -Tempo di esecuzione*
  - Al crescere della mole di dati che un programma deve elaborare, cresce anche il tempo di compilazione? E il tempo di esecuzione?
- *Compilatore – Macchina virtuale – Codice oggetto – Processore*
  - Chi esegue chi?