# Linguaggi

VR443470

marzo 2023

# Indice

# 1 Introduzione

## 1.1 Nascita dei linguaggi

Il problema **principale dell'informatica** consiste nel voler far eseguire ad una macchina **algoritmi** che manipolano **dati**. Con il termine "macchina" si intende un dispositivo **programmabile**, ovvero un calcolatore, che può eseguire un insieme di istruzioni chiamate programmi, ricevuti come input (**macchine universali**).

In particolare, i computer moderni hanno un'architettura che nasce da quella di Von Neumann.

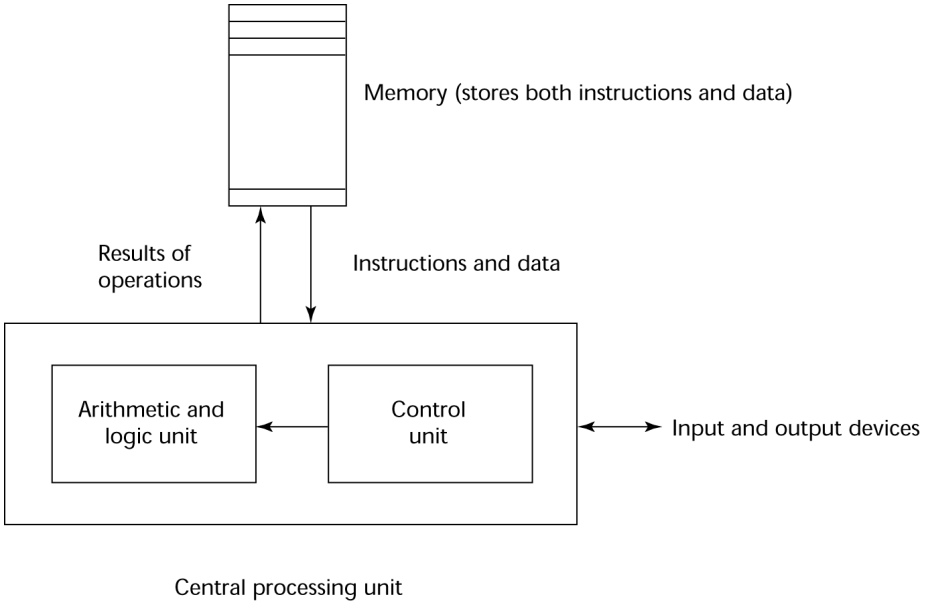



Figura 1: Architettura di Von Neumann.

Per operare su tali macchine, fu necessario inserire una CPU per eseguire algoritmi e operare sui dati in memoria. Il primo linguaggio che consente di programmare tale architettura è quello basato sull'implementazione dell'architettura stessa, ovvero la programmazione con schede perforate per esempio.

## 1.2 Definizioni

### 1.2.1 Algoritmo

Un **algoritmo** **è una sequenza finita di passi primitivi di calcolo descritti mediante una frase ben formata (programma) in un linguaggio di programmazione**. In altre parole, un algoritmo scompone un calcolo complesso in passi elementari di computazione. Esso è dunque un concetto astratto che trova la sua forma concreta in un programma che è la sequenza finita di istruzioni.

Data la definizione, è necessario precisare che:

- **Un programma non è necessariamente un algoritmo**, poiché una frase grammaticalmente corretta potrebbe non avere significato;
- **Lo stesso algoritmo può avere concretizzazioni diverse**, ovvero lo stesso algoritmo può essere implementato da infiniti programmi.

### 1.2.2 Dati

I programmi sono la concretizzazione degli algoritmi, i quali manipolano i **dati**. Essi sono informazioni memorizzate sia concretamente in celle di memoria, sia astrattamente in elementi che il linguaggio di programmazione può manipolare, ovvero le **variabili**.

### 1.2.3 Sintassi e semantica

La trasformazione (scrittura) di un programma in un determinato linguaggio di programmazione rappresenta la **sintassi**, mentre l'effetto della sua esecuzione e la trasformazione dei dati eseguita, costituisce la **semantica**.

### 1.2.4 Linguaggio matematico e logico

Il **linguaggio matematico** è una notazione rigorosa per rappresentare funzioni, ma non sempre oggetti infiniti e computazioni, cioè passi di calcolo.

Il **linguaggio logico** sono regole e assiomi che rendono possibile specificare il processo di computazione, in modo implicito, e consente di rappresentare formalmente oggetti infiniti in modo finito, ma non computazioni infinite.

### 1.2.5 Linguaggio di programmazione

Un **linguaggio di programmazione** consente di specificare in modo accurato esattamente le primitive del processo di computazione, con la rigorosità e la potenza della logica.

### 1.2.6 Programma

Un **programma** è un insieme finito di istruzioni e costrutti del linguaggio di programmazione.

---

## 1.3 Aspetti di progettazione

### 1.3.1 Leggibilità

La **leggibilità** (*readability*) è la sintassi chiara, l'assenza di ambiguità, la facilità di lettura e la comprensione dei programmi.

I fattori che contribuiscono alla leggibilità sono:

1. La **semplicità di un linguaggio**, per esempio pochi ed essenziali costrutti base. Infatti, un linguaggio inizia ad essere complicato quando per poter fare la stessa cosa si possono seguire molti percorsi diversi. Un altro fattore di complicazione è l'overloading degli operatori, ovvero quando il simbolo di un operatore ha molteplici significati.

2. L'**ortogonalità** della progettazione di un linguaggio. Un elemento di un programma è ortogonale se è indipendente dal contesto di utilizzo all'interno di esso. Più un programma è ortogonale, meno eccezioni alla regola esistono.

3. **Presenza di strumenti per la definizione di tipi di dati e strutture dati**. Ad esempio l'uso di booleani al posto dei valori interi.

4. **Struttura della sintassi**, come parole chiave significative ad esempio.

### 1.3.2 Scrivibilità

La **scrivibilità** (*writability*) è la facilita di utilizzo di un linguaggio per creare programmi, la facilità di analisi e la verifica dei programmi. I fattori che influenza la leggibilità sono gli stessi della scrivibilità.

In breve i fattori che che contribuiscono alla scrivibilità sono:

1. **Semplicità e ortogonalità**. La presenza di pochi costrutti consente al programmatore di conoscerli in gran parte e di sfruttare al massimo il linguaggio. Stessa cosa per il numero di primitive.

2. **Supporto per l'astrazione**. La possibilità di utilizzare strutture o operazioni complesse in modi che permettono di ignorare i dettagli. Esistono due tipi di astrazione: processi e dati.

3. **Espressività**. Si riferisce a molte caratteristiche, per esempio mettere a disposizione un insieme di modi relativamente convenienti per specificare operazioni.

### 1.3.3 Affidabilità e costo

Per **affidabilità** (*reliability*) si intende la conformità alle sue specifiche. Mentre per **costo**, letteralmente il costo complessivo di utilizzo.

Un **programma** viene categorizzato come **affidabile** se soddisfa le seguenti condizioni:

1. ***Type checking***, ovvero il controllo degli errori di tipo. Viene eseguito spesso a tempo di compilazione poiché risulta costoso.
2. **Gestione delle eccezzioni**. Gestire gli errori run-time per consentire la continuazione dell'esecuzione e l'attuazione di eventuali misure correttive.
3. **Presenza di potenziali aliasing**. La presenza di due o più metodi di riferimento per la stessa locazione di memoria è un problema.

Mentre le **specifiche di costo** riguardano:

- L'addestramento di programmatore per usare il linguaggio
- Scrittura di programma
- Compilazione dei programmi
- Esecuzione dei programmi
- Sistema di implementazione del linguaggio, ovvero la disponibilità di compilatori liberi
- Poca affidabilità fanno lievitare i costi
- Mantenimento dei programmi

## 1.4 Classificazione dei linguaggi

I linguaggi possono essere classificati per: **metodo di computazione** e **per caratteristiche**.

### 1.4.1 Metodo di computazione

I linguaggi possono essere a:

- **Basso livello**. Questi linguaggi hanno caratteristiche strettamente dipendente all'architettura su cui si sta programmando. Per esempio:
  - Linguaggio binario che non fa distinzione tra dati e programmi;
  - Assembly, linguaggio strutturato molto basso, vicino al linguaggio macchina.
- **Alto livello**. Questi linguaggi consentono una programmazione strutturata in cui dati ed istruzioni hanno rappresentazioni diverse. Esistono tre tipi:
  - **Linguaggi imperativi** che descrivono come **concetto chiave** l'elemento fondamentale dell'architettura di Von Neumann, ovvero la **cella di memoria**.
    Il concetto di variabile rappresenta l'astrazione logica della cella.
    Il concetto di assegnamento rappresenta l'operazione primitiva di modifica della cella di memoria e dunque dello stato della macchina.
    Nei linguaggi imperativi, gli assegnamenti vengono controllati in modo sequenziale, condizionale e ripetuti.
  - **Linguaggi funzionali** sono molto vicini alla matematica. Essi descrivono i passi di calcolo come funzioni matematiche. Il *core* principale si concentra sulla composizione e applicazione di funzioni.
    Una variabile viene intesa come un'incognita matematica e sostituita come se fosse un *placeholder* all'interno del linguaggio. Infatti, essa non può cambiare nel tempo durante la computazione.
  - **Linguaggi logici** usano la logica, ovvero eseguono pattern matching. Come passo di calcolo primitivo utilizzano l'unificazione o la sostituzione.

### 1.4.2 Per caratteristiche

La classificazione per caratteristiche è una metodologia utilizzata principalmente all'inizio dell'era informatica per studiare le caratteristiche di base quali: strutture di base di controllo, strutture per i dati, efficienze nell'esecuzione.

Andando avanti con il tempo, la classificazione si è focalizzata su caratteristiche aggiuntive, ovvero le strutture di base rimangono le stesse, ma ad esse vengono aggiunte nuove caratteristiche. In questo modo viene migliorata la soluziona di specifici problemi.

## 1.5 Implementazione dei linguaggi

L'**implementazione di un linguaggio** ha un collegamento stretto con il funzionamento della macchina su cui deve essere eseguito. Infatti, l'implementazione riguarda le metodologie per rendere comprensibile alla macchina da programmare il linguaggio scelto. Per farlo, è necessario introdurre il funzionamento di una macchina basata sull'architettura di Von Neumann. Essa si basa sulla ripetizione di un ciclo che costituisce l'**interprete** del linguaggio che la macchina riconosce:

- Lettura dell'istruzione dalla memoria (*fetch*)
- Decodifica dell'istruzione (*decode*)
- Lettura di eventuali operandi
- Memorizzazione ed esecuzione del risultato (*exec*)


Figura 2: Ciclo di esecuzione delle istruzioni.

### 1.5.1 Macchina astratta

L'implementazione di un linguaggio **significa** considerare una macchina astratta poiché lavorando ad alto livello, le istruzioni vengono interpretate e dunque si ignorano momentaneamente il linguaggio binario e la macchina fisica.

Dato un linguaggio $L$ di programmazione, la **macchina astratta** $M_L$ per $L$ è un insieme di strutture dati ed algoritmi che permettono di memorizzare ed eseguire i programmi scritti in $L$.

La collezione di strutture dati ed algoritmi è necessario per:

- Acquisire la prossima istruzione
- Gestire le chiamate e i ritorni dai sottoprogrammi
- Acquisire gli operandi e memorizzare i risultati delle operazioni
- Mantenere le associazioni fra nomi e valori denotati
- Gestire dinamicamente la memoria

In altre parole, una **macchina astratta** è la combinazione di una memoria che immagazzina i programmi e di un interprete che esegue istruzioni dei programmi.

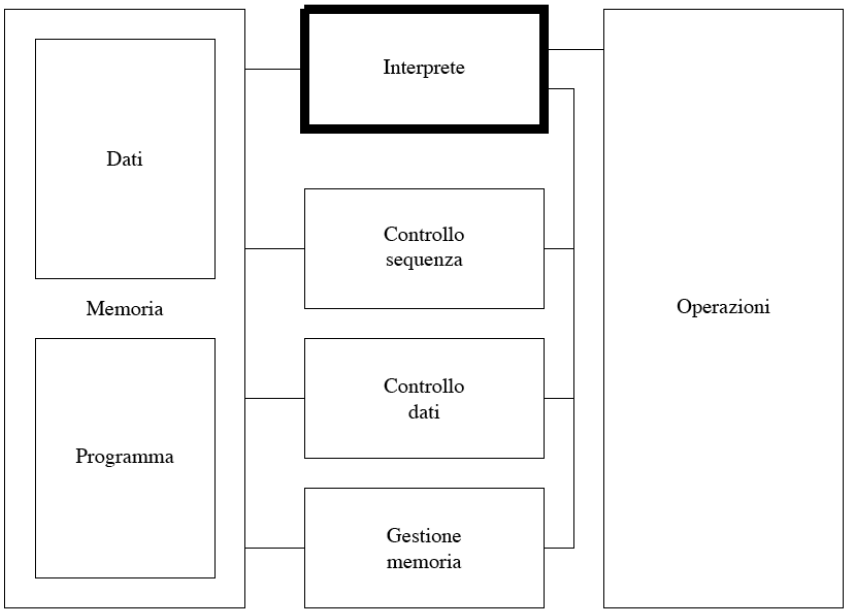



Figura 3: Rappresentazione di una macchina astratta.

Il **linguaggio** $L$ riconosciuto (interpretato) dalla macchina astratta $M_L$ viene chiamato **linguaggio macchina**. Formalmente, è l'insieme di tutte le stringhe interpretabili dalla macchina astratta $M$.

### 1.5.2 Realizzazione di una macchina astratta a vari livelli

Qualsiasi macchina astratta, per essere eseguita, deve prima o poi utilizzare qualche dispositivo hardware. Questo però non significa che tutte le macchine sono realizzate a livello hardware. Infatti, la realizzazione di una macchina astratta può avvenire tre categorie:

- **Realizzazione hardware (HW)**. Sempre possibile e concettualmente semplice. Il linguaggio macchina è il linguaggio fisico/binario e si realizza mediante dispositivi fisici. Data la sua lontananza dai linguaggi ad alto livello, la loro programmazione risulta complessa. Questo è uno dei tanti motivi per cui viene usata solo per sistemi dedicati.
- **Realizzazione firmware (FW)**. Le strutture dati e gli algoritmi vengono simulati nella macchina mediante microprogrammi. Il linguaggio macchina è a basso livello e consiste in microistruzioni che specificano le operazioni di trasferimento dati tra registri. Il vantaggio è dato dalla velocità e la flessibilità maggiore rispetto all'hardware.
- **Realizzazione software (SW)**. Le strutture dati e gli algoritmi vengono realizzati tramite un linguaggio implementato. In questo modo è possibile scrivere programmi che interpretano i costrutti del linguaggio macchina simulando le funzionalità della macchina. La velocità viene diminuita ma aumenta molto la flessibilità.

### 1.5.3 Realizzazione software e livelli di astrazione

Con la realizzazione software, vengono utilizzati linguaggi di programmazione ad alto livello poiché essi implementano una struttura suddivisa a livelli di astrazione. Ogni livello coopera in modo sequenziale ma allo stesso tempo è indipendente.

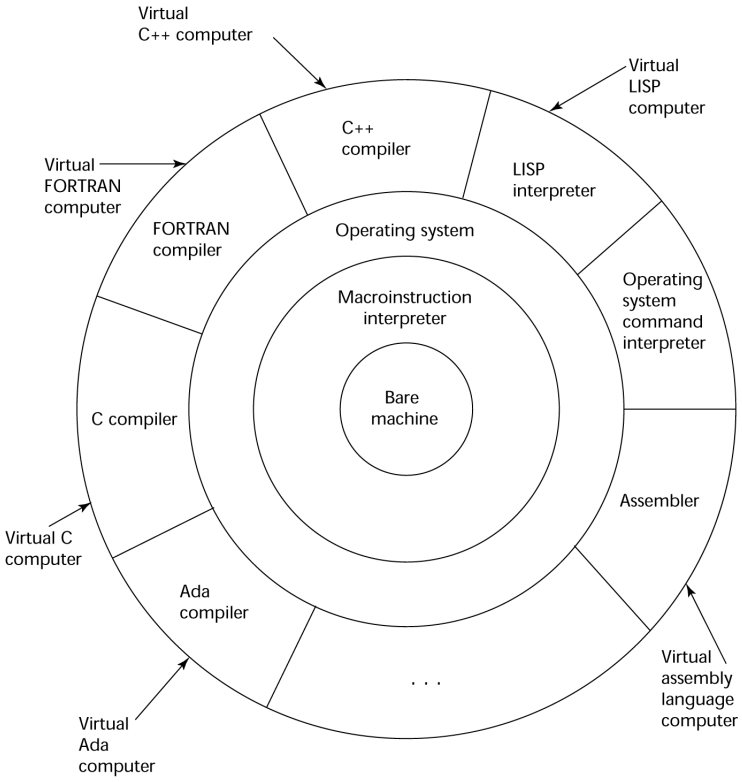
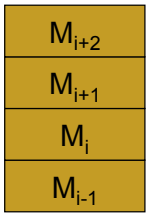



Figura 4: Livelli di astrazione utilizzati dai linguaggi ad alto livello.

Quindi, la macchina può essere vista come una stratificazione di livelli di astrazione:




$M_i$: • usa i servizi forniti da $M_{i-1}$ (il linguaggio $L_{Mi-i}$)
• per fornire servizi a $M_{i+1}$(interpretare $L_{Mi+i}$)
• nasconde (entro certi limiti) la macchina $M_{i-1}$

Stando al livello i può non essere noto
(e in genere non serve sapere...)
quale sia il livello 0 (hw)

## 1.6 Realizzazione di una macchina a livello software/firmware

Sia $L$ un linguaggio da implementare e sia $M_{L_0}$ una macchina astratta a disposizione che ha come linguaggio macchina $L_0$. La macchina astratta $M_{L_0}$ è il livello su cui si vuole implementare il linguaggio $L$ e che mettette a disposizione di $M_L$ le sue funzionalità.

Dunque, la realizzazione di una macchina astratta $M_L$ consiste nel realizzare una macchina che "traduce" il linguaggio $L$ (ad alto livello per esempio) in linguaggio macchina $L_0$, ovvero che interpreta tutte le istruzioni di $L$ come istruzioni di $L_0$. La traduzione avviene tramite due metodi a scelta:

- **Soluzione interpretativa**. Simulazione dei costrutti della macchina astratta $M_L$ da realizzare, mediante programmi scritti in $L_0$;
- **Soluzione compilativa**. Traduzione esplicita dei programmi di $L$ in corrispondenti programmi di $L_0$.

### 1.6.1 Soluzione interpretativa: interprete

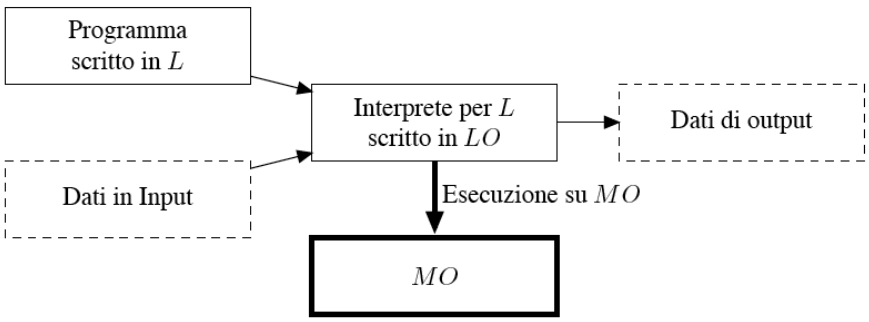
La soluzione interpretativa prevede l'utilizzo di un interprete per la realizzazione di una macchina astratta. Un **interprete** è un programma $\text{int}^{\text{L}_0,\text{L}}$ che esegue, sulla macchina astratta per $L_0$, programmi $P^L$, scritti nel linguaggio di programmazione $L$, su un input fissato appartenente all'insieme di dati (input e output). In breve, un interprete è una **macchina universale** che preso un programma e un suo input, lo esegue su quell'input usando solo funzionalità messe a disposizione dal livello (macchina astratta) sottostante.

***Notazioni***

- $Prog^L$ è l'insieme di programmi scritti nel linguaggio di programmazione $L$;
- $D$ è l'insieme di dati, ovvero input e output;
- $P^L$ è il programma scritto nel linguaggio di programmazione $L$;
- Relazioni ovvie: $P^L \in Prog^L$ e $in, out \in D$;
- $[\![P^L]\!] : D \longrightarrow D$ è la notazione utilizzata per indicare che l'esecuzione del programma scritto nel linguaggio di programmazione $L$ con input $in$ è uguale all'output $out$. Quindi $[\![P^L]\!](in) = out$.

Un **interprete formalmente** (**definizione**) è esprimibile nel seguente modo. Si consideri un interprete da $L$ a $L_0$: dato $P^L \in Prog^L$ e $in \in D$, un interprete $int^{L,L_0}$ per $L$ su $L_0$ è un programma tale che $[\![int^{L,L_0}]\!] : \left(Prog^L\right) \longrightarrow D$ e dunque $[\![int^{L,L_0}]\!]\left(P^L, in\right) = [\![P^L]\!](in)$.

Anche un programma può essere utilizzato come dato di input in un altro programma. Si osservi il seguente diagramma:




Si noti come un programma scritto nel linguaggio $L$, insieme ad eventuali altri input, viene interpretato da un programma creato appositamente per eseguire questo compito su $L$, ma scritto in $L_0$. L'**esecuzione comporta una decodifica e non una traduzione esplicita**. Infatti, l'interprete simula ogni istruzione di $L$ utilizzando un certo insieme di istruzioni di $L_0$. Questa è la base dei linguaggi di scripting.

### 1.6.2 Soluzione interpretativa: operazioni e struttura

Un interprete può eseguire una serie di **operazioni**:

- **Elaborazione dei dati primitivi**. I dati primitivi sono dati rappresentabili in modo diretto nella memoria, per esempio i numeri. Le elaborazioni di essi, sono implementate direttamente nella struttura della macchina;
- **Controllo di sequenza delle esecuzioni**. Non è altro che la gestione del flusso di esecuzione delle istruzioni, le quali non sempre sono sequenziali, tramite alcune strutture dati;
- **Controllo dei dati**. Recupero dei dati necessari per eseguire le istruzioni. I dati possono riguardare le modalità di indirizzamento della memoria e l'ordine con cui recuperare gli operandi;
- **Controllo della memoria**. È necessaria una gestione della memoria per allocare dati e programmi. Cambia a seconda del tipo di realizzazione della macchina astratta:
  - Realizzazione hardware (HW): la gestione è semplice poiché nella peggiore delle ipotesi, i dati potrebbero essere rimasti sempre nelle stesse locazioni.
  - Realizzazione software (SW): la gestione è complessa ed esistono costrutti di allocazione e deallocazione che richiedono alcune strutture dati (e.g. pile) e operazioni dinamiche.

Date le operazioni elencate, il **ciclo di esecuzione di un interprete** è il seguente:

```
1  begin
2    go := true;
3    while go do begin
4      FETCH(OPCODE, OPINFO) at PC
5      DECODE(OPCODE, OPINFO)
6      if OPCODE needs ARGS then FETCH (ARGS)
7      case OPCODE of
8        OP1: EXECUTE(OP1, ARGS)
9        ...
10       OPn: EXECUTE(OPn, ARGS)
11       HLT: go := false;
12     if OPCODE has result then STORE(RES);
13     PC := PC + SIZE(OPCODE);
14   end
15 end
```

- Righe 1-3: finché go ha il valore true, il codice viene eseguito;
- Riga 4: estrazione dell'istruzione riferita ad OPCODE (controllo sequenza 1 su 2);
- Righe 5: decodifica dell'istruzione estratta alla riga precedente;
- Riga 6: prelievo dalla memoria gli operandi richiesti da OPCODE e nelle modalità individuate (controllo dati 1 su 2);
- Righe 7-11: esecuzione delle operazioni;

- Riga 12: se l'operazione ha un risultato da salvare, allora viene salvato in memoria (controllo dati 2 su 2);
- Riga 13: viene incrementato il *program counter* (PC) per eseguire la prossima istruzione (controllo sequenza 2 su 2).

Il ciclo continua ad essere eseguito finché la variabile `go` ha valore `true`. Inoltre, le istruzioni riferite al program counter sono chiamate istruzioni di **controllo sequenza** (CS) perché manipolano e accedono al program counter. Mentre le operazioni di memorizzazione/estrazione sugli argomenti si chiamano **controllo dati** (CD).

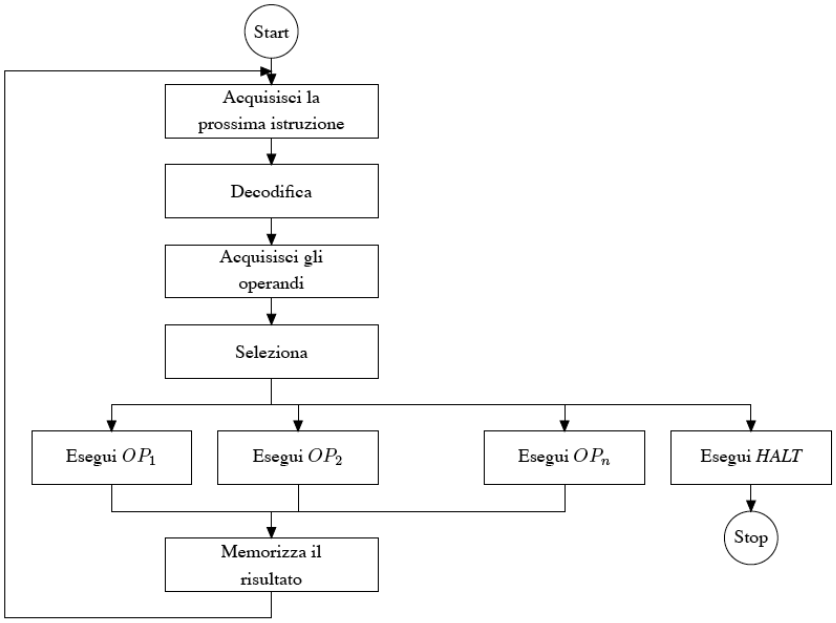



Figura 5: Diagramma a blocchi della struttura di un interprete.

### 1.6.3 Soluzione interpretativa: pro e contro

- **Pro:**
  - **Facilità di interazione *run-time***. Interpretazione al momento dell'esecuzione consente di interagire direttamente con l'esecuzione del programma (*debugging*);
  - Velocità nello sviluppo applicativo di un interprete, quindi **tempi ridotti per la sua creazione**;
  - Utilizzo della **memoria ridotto** rispetto ad un compilatore.
- **Contro:**
  - Tempi di decodifica sommati a quelli d'esecuzione ogni volta che un'istruzione viene eseguita, si traduce in un'**esecuzione lenta** e quindi una scarsa efficienza della macchina.

### 1.6.4 Soluzione compilativa: compilatore

Un **compilatore** è un programma $\text{comp}^{L_0,L}$ che **traduce**, preservando semantica e funzionalità, programmi scritti nel linguaggio di programmazione $L$ in programmi scritti in $L_0$, e quindi eseguibili direttamente sulla macchina astratta per $L_0$. Come l'interprete, anche il compilatore accetta un programma come input poiché viene considerato come dato.

***Notazioni***

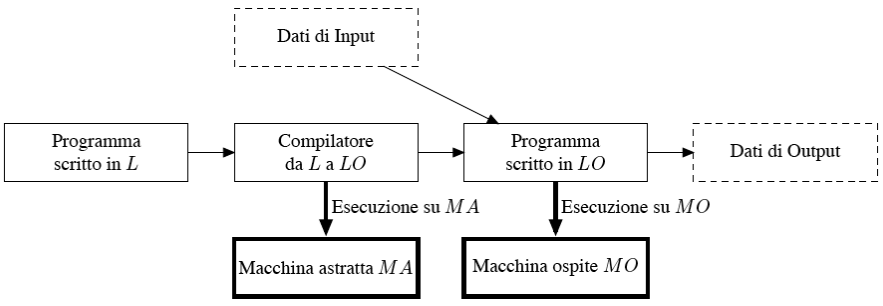
- $Prog^L$ è l'insieme di programmi scritti nel linguaggio di programmazione $L$;
- $D$ è l'insieme di dati, ovvero input e output;
- $P^L$ è il programma scritto nel linguaggio di programmazione $L$;
- Relazioni ovvie: $P^L \in Prog^L$ e $in, out \in D$;
- $[\![P^L]\!] : D \longrightarrow D$ rappresenta la semantica di $P^L$.

Un **compilatore formalmente** (**definizione**) è esprimibile nel seguente modo. Dato $P^L \in Prog^L$, un **compilatore** $\text{comp}^{L,L_0}$ da $L$ a $L_0$ è un programma tale che $[\![comp^{L,L_0}]\!] : Progr^L \longrightarrow Progr^{L_0}$ e:

$$[\![comp^{L,L_0}]\!]\left(P^L\right) = P^{L_0} \text{ tale che } \forall in \in D.[\![P^{L_0}]\!](in) = [\![P^L]\!](in)$$

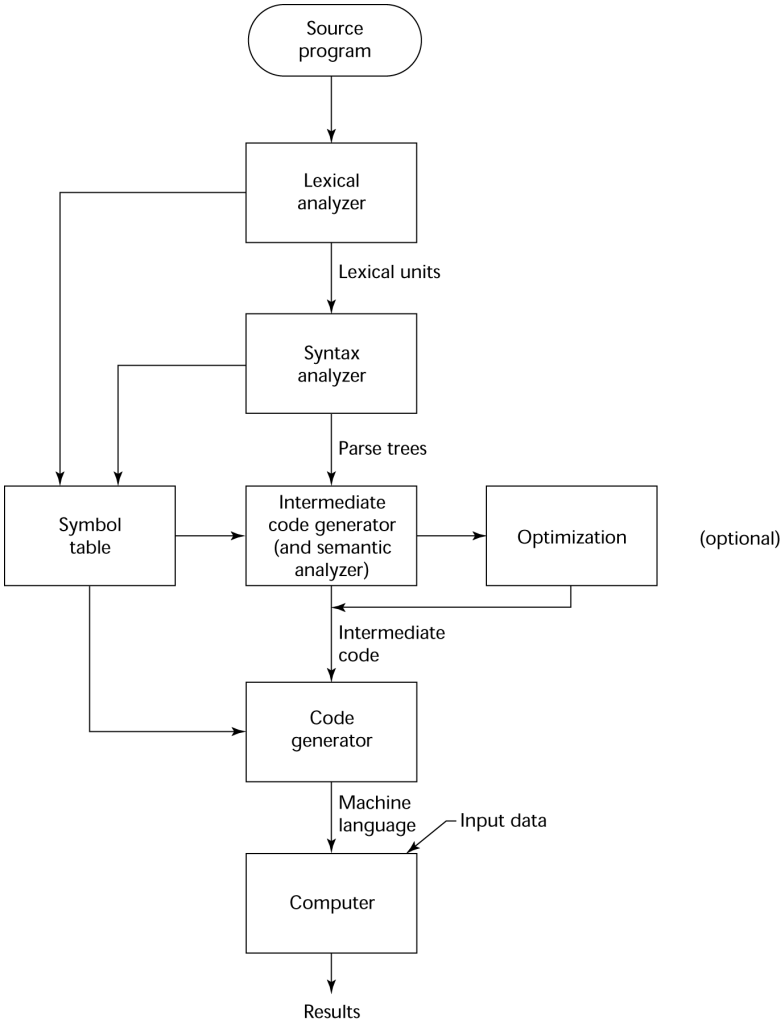
Ovvero che l'esecuzione della compilazione del linguaggio $L$ a $L_0$ con input il programma scritto in $L$, l'output sia uguale al programma scritto nel linguaggio $L_0$; tale che per ogni input appartenente all'insieme dei dati, l'esecuzione del programma scritto in $L_0$ con input $in$, sia uguale all'esecuzione del programma scritto in $L$ con input $in$.

Con un compilatore, la **traduzione** è **esplicita** poiché il codice in $L$ viene prodotto come output e non eseguito. Quindi, per eseguire il programma $P^L$ con input $in$, è necessario prima eseguire $comp^{L,L_0}$ con $P^L$ come input. L'esecuzione avverrà sulla macchina astratta $M_A$ del linguaggio in cui è scritto il compilatore. Il risultato dunque è un altro programma (compilato) $P^{L_0}$, scritto in $L_0$. Solo a questo punto è possibile eseguire $P^{L_0}$ su $M_{L_0}$ con input $in$. Un **esempio** di linguaggio compilato è il C.



### 1.6.5 Soluzione compilativa: struttura

La compilazione deve tradurre un programma da un linguaggio ad un altro preservandone la semantica: si deve avere la certezza che il programma compilato faccia esattamente quello che faceva il sorgente. L'**esecuzione** di un compilatore si articola in varie fasi:




- **Analisi lessicale** (*Lexical analyzer*), divide il programma in componenti sintattici primitivi chiamati **tokens** (identificatori, numeri, parole riservate). I *tokens* sono coloro che formano i linguaggi regolari.
  In altre parole, l'**analisi lessicale converte caratteri del programma sorgente in unità lessicali**;

- **Analisi sintattica** (*Syntax analyzer*), crea una rappresentazione ad albero della sintassi del programma. Ogni foglia è un *token* e le foglie lette da sinistra verso destra costituiscono frasi ben formate del linguaggio. Inoltre, l'albero costituisce la struttura logica del programma e dunque nel momento in cui non fosse possibile costruire l'albero, significherebbe che qualche frase è illegale. Questo genere di evento si traduce in un errore di compilazione. Le frasi di token formano linguaggi CF.
  In altre parole, l'**analisi sintattica trasforma unità lessicali in *parse tree* che rappresentano la struttura sintattica del programma**.
- **Tabella dei simboli** (*Symbol table*), memorizza le informazioni sui nomi presente nel programma, come gli identificatori, le chiamate di procedura, ecc.
- **Analisi semantica** (*Semantic analyzer*), consente di rilevare errori semantici, grazie all'analisi semantica, e di generare codice intermedio che ha la caratteristica di essere indipendente dall'architettura (compito del *Intermediate code generator*).
- **Ottimizzazione** (*Optimization*), opzionale, consente di ottimizzare il codice.
- **Generatore di codice** (*Code generator*), viene generato codice macchina che ha la caratteristica di essere dipendente dall'architettura.

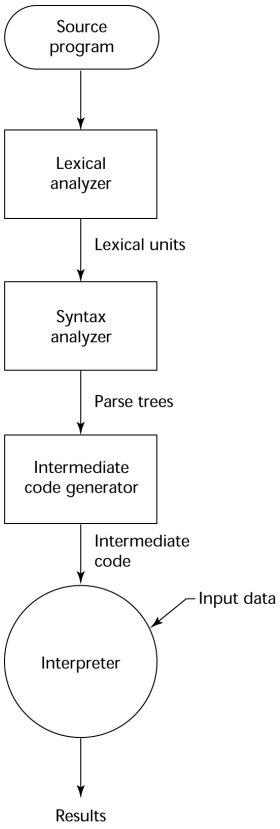
---

### 1.6.6 Soluzione compilativa: pro e contro

- **Pro:**
  - **Esecuzione molto efficiente**, il codice viene anche ottimizzato;
- **Contro:**
  - **Interazione *run-time* molto difficile**;
  - Un **errore** a *run-time* è **difficile da associare** all'esatto comando del codice sorgente (debugging complesso);

### 1.6.7 Soluzione reale: ibrido

Nella realtà esiste un compromesso tra compilatore e interprete. Ovvero, una **soluzione ibrida** dove il linguaggio ad alto livello viene compilato in un linguaggio a più basso livello che poi viene interpretato.




Il **procedimento** è il seguente.
Si consideri il linguaggio ad alto livello $L$ per il quale si deve realizzare la macchina astratta $M_L$.
Il linguaggio $L$ viene quindi tradotto in un linguaggio intermedio $L_{Mi}$ la cui macchina astratta $M_I$ consiste in un interprete del linguaggio $L_{Mi}$ sulla macchina ospite $M_O$.

La separazione non è netta poiché vengono interpretati i costrutti lontani da $M_O$, mentre viene compilato il resto. Il passaggio chiave è la traduzione (compilazione) da $L$ ad un linguaggio intermedio (quello interpretato). Questo accade spesso nella realtà, specialmente con le *system call* del sistema operativo. In parole povere, si cerca di trovare una connessione a metà strada.

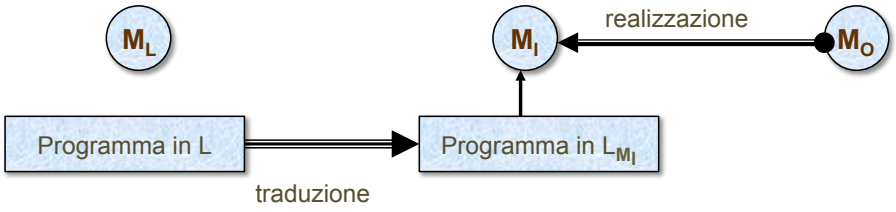



Figura 6: Soluzione ibrida con le *system call*.

---

## 1.7 Sintesi

Nell'evoluzione dei linguaggi di programmazione, esistono fondamentalmente tre situazioni possibili:

- **Interprete puro** (paragrafo 1.6.1), $M_L = M_I$ (interprete per $L$ realizzato sulla macchina ospite $M_O$). Per esempio i linguaggi logici e funzionali, e di scripting (JS, PHP, ...)
- **Compilatore** (paragrafo 1.6.4), macchina intermedia $M_I$ realizzata per estensione sulla macchina ospite $M_O$. Per esempio i linguaggi imperativi come C, C++, Pascal
- **Implementazione mista** (paragrafo 1.6.7), traduzione dei programmi da $L$ ad un linguaggio intermedio $L_{Mi}$. I programmi $L_{Mi}$ sono poi interpretati sulla macchina ospite $M_O$. Per esempio, Java con il suo linguaggio intermedio Java bytecode, Pascal con il suo linguaggio intermedio P-code.