

- [Interprete \(per il linguaggio \\$L\\$\)](#)
- [Compilatore \(per il linguaggio \\$L\\$ verso il linguaggio \\$M\\$\)](#)
  - [Compilatori ottimizzati](#)
- [Interpretazione o compilazione](#)
  - [Esempi per gli approcci misti](#)
- [Perché si studiano i compilatori?](#)

---

## Interprete (per il linguaggio \$L\$)

Un programma che prende in input un programma eseguibile (espresso nel linguaggio \$L\$) e lo **esegue**, producendo l'output corrispondente, è detto *interprete*.

Un programma è eseguibile quando esiste un interprete in grado di eseguirlo.

## Compilatore (per il linguaggio \$L\$ verso il linguaggio \$M\$)

Un programma che prende in input un programma eseguibile (espresso nel linguaggio \$L\$) e lo **traduce**, producendo in output un programma equivalente (espresso nel linguaggio \$M\$) è detto *compilatore* (traduttore). Per eseguire il compilato serve un **interprete** per il linguaggio \$M\$.

Esiste anche un compilatore da un linguaggio a se stesso. Questo viene utilizzato per vari motivi, tra cui: semplificazione della leggibilità del codice, offuscamento, firma del codice proprietario.

## Compilatori ottimizzati

Il compilatore traduce il programma in modo da ottenere un **miglioramento** di qualche metrica (tempo di esecuzione, memoria usata, consumo energetico, ecc.). Nonostante questo, l'ottimizzazione vera e propria (in senso matematico) è in pratica impossibile da ottenersi, per cui ci si accontenta di **tecniche euristiche**, che funzionano bene nei casi comuni ma non forniscono garanzie di ottimalità.

---

## Interpretazione o compilazione

La compilazione è un'attività *off-line*, per vari motivi:

- Identificare alcuni errori di programmazione prima dell'esecuzione del programma.
- Migliorare l'efficienza (e.g., spostando alcuni calcoli a tempo di compilazione o evitando la ripetizione di calcoli identici).

- Rendere utilizzabili alcuni costrutti dei linguaggi ad alto livello (inaccettabilmente costosi per l'approccio interpretato).

| Approccio                                 | Linguaggi                           | Note   |
|---|-------------------------------------|--|
| Linguaggi <i>tipicamente</i> compilati    | FORTRAN, Pascal, C, C++, OCaml, ... | possono comunque essere interpretati (e.g., <code>cling</code> ) |
| Linguaggi <i>tipicamente</i> interpretati | PHP, R, Matlab, ...                 | possono comunque essere compilati                                |
| Approccio misto                           | Java, Python, SQL, ...              | Varie combinazioni di compilazione e interpretazione             |

È **necessario** stabilire dei compromessi sulla scelta tra interpretazione e compilazione:

- Bilanciamento tra attività *off-line* e *on-line*.
- Il tempo di compilazione deve essere accettabile.
- L'occupazione in spazio del programma compilato deve essere accettabile.

Un esempio lo si vede nella *programmazione generica* con i generics in Java e i template in C++:

- Nel primo caso, il compilatore `javac` applica la cosiddetta *type erasure*, sostituendo (nei casi più comuni) tutti i parametri di tipo con `Object` ; ciò assicura che a tempo di esecuzione non verrà creato nuovo codice in corrispondenza dell'istanziamento di oggetti generici [[rif. Oracle](#)].
- Nel secondo caso, il compilatore si occupa di "leggere" il template in questione e generare il codice *ad-hoc* per il tipo concreto richiesto.

È chiaro dunque che la compilazione di un template C++ introduce un overhead maggiore rispetto alla controparte in Java. Si rifletta però sul fatto che il conoscere i tipi utilizzati permette al compilatore di poter introdurre in alcuni casi delle ottimizzazioni; ne è un esempio intuitivo il caso di un "vettore" parametrico rispetto al tipo dell'oggetto contenuto: per Java, il tipo generico è una reference ad una porzione di memoria e non è assicurato che posizioni successive del vettore si riferiscano a segmenti contigui della stessa; al contrario, conoscendo la dimensione dell'oggetto contenuto, il compilatore C++ è in grado di disporre gli oggetti su porzioni successive di memoria.

## Esempi per gli approcci misti

Java è un tipo di approccio misto perché:

- Una fase di compilazione da sorgente Java verso bytecode Java.
- L'interpretazione del bytecode Java da parte della JVM (Java Virtual Machine).
- Compilazione JIT (Just-In-Time, a tempo di esecuzione) di alcune porzioni (e.g., cicli onerosi ripetuti più volte durante l'esecuzione) di bytecode verso il linguaggio macchina.

SQL è un tipo di approccio misto perché:

- Le query SQL sono interpretate.
- L'interprete include fasi di ottimizzazione.

- Offre la possibilità di compilare in forma ottimizzata porzioni di SQL (prepared statements, stored procedures, ecc.).

---

# Perché si studiano i compilatori?

---

I compilatori si studiano per vari motivi:

## 1. Applicazioni pratiche a concetti teorici imparati.

- Analisi lessicale, espressioni regolari e automi a stati finiti.
- Analisi sintattica, grammatiche libere da contesto e automi a pila.
- Analisi e ottimizzazione IR, teoria dell'approssimazione, calcoli di punto fisso, equivalenza tra programmi.
- Progettazione dei linguaggi di programmazione

## 2. Applicazioni di algoritmi e strutture dati sofisticati.

- Tabelle hash, alberi, grafi.
- Algoritmi di visita (di alberi e grafi).
- Algoritmi greedy, dynamic programming, tecniche euristiche di ricerca in spazi di soluzioni.
- Pattern matching, scheduling (processori), colorazione di grafi.

## 3. Interessanti problemi di system/software engineering.

- Il compilatore è parte importante del software di sistema: interconnessioni con architettura e sistema operativo.
- Gestione progetto complesso, organizzazione del codice.
- Ottimo test case per applicare i design pattern (e.g., visitor pattern).
- Compromessi tra efficienza e scalabilità.

## 4. Implementare interpreti/compilatori per DSL (Domain Specific Language).

- DSL: creazione di linguaggi ad alto livello per semplificare il processo di sviluppo.
- Linguaggi di alto livello progettati per una classe specifica di applicazioni.
- Linguaggi per la generazione automatica di documentazione tecnica per il software (Doxygen, Javadoc).
- Alcuni esempi di DSL riguardano: linguaggi di scripting per librerie grafiche, videogiochi, automazione industriale, robotica, domotica, data science, ecc..

Un possibile motivo del perchè si studiano i compilatori potrebbe fare riferimento all'implementazione di un compilatore per un **linguaggio mainstream** (C, C++, Java, Python, ecc.). Questa situazione si presenta molto raramente e, spesso, è ben oltre le capacità del programmatore medio; in alcuni casi però si riscontra la necessità di **estendere** un compilatore esistente per:

- Supportare nuovi costrutti del linguaggio.
- Supportare nuove architetture hardware (e.g., GPU).

Sono presenti dei progetti collaborativi come **Clang/LLVM** che sono aperti a contributi esterni.