1. Introduzione

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

Un **compilatore** per il linguaggio L verso il linguaggio M è 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)

Nota. Per eseguire il compilato serve un **interprete** per il linguaggio M

Nota. Un decompilatore è un programma che esegue l'operazione inversa a quella del compilatore

Un **compilatore ottimizzante** è un compilatore che traduce il programma in modo da ottenere un miglioramento di qualche metrica (tempo di esecuzione, memoria usata, consumo energetico, . . .)

Attenzione. L'ottimizzazione vera e propria (in senso matematico, ovvero in senso assoluto) è in pratica impossibile da ottenersi, per cui si ci accontenta di tecniche euristiche, che funzionano bene nei casi comuni, ma che non forniscono garanzie di ottimalità.

Interpretazione vs Compilazione

La **compilazione**, che è un'attività svolta "offline" e non durante l'esecuzione del programma, consente di identificare errori di programmazione prima dell'esecuzione stessa. Inoltre, permette di migliorare l'efficienza del programma, ad esempio spostando alcuni calcoli al tempo di compilazione o evitando la ripetizione di calcoli identici. La compilazione rende anche utilizzabili alcuni costrutti dei linguaggi ad alto livello, che sarebbero troppo costosi in termini di risorse con un approccio interpretato.

D'altra parte, l'**interpretazione**, un'attività "online" che avviene durante l'esecuzione del codice, offre vantaggi diversi. Consente l'esecuzione immediata del codice sorgente senza richiedere una fase di compilazione e permette al programma di essere eseguito su piattaforme diverse senza la necessità di modifiche o ricompilazioni, poiché l'interprete esegue direttamente il codice sorgente.

Nota: La scelta tra compilazione e interpretazione dipende da vari fattori, inclusi gli obiettivi per cui un linguaggio è progettato. È importante ricordare che la compilazione o l'interpretazione non sono caratteristiche intrinseche di un linguaggio; un linguaggio può

essere tipicamente compilato o interpretato, ma non è obbligatoriamente legato a uno dei due approcci.

Infine, esistono anche approcci misti che combinano elementi di compilazione e interpretazione.

Esempio. Il codice sorgente Java viene compilato in bytecode Java, che viene interpretato dalla JVM (Java Virtual Machine). Inoltre, la JVM identifica le sezioni critiche del codice (ad esempio, valutando quante volte viene richiamato un metodo) e compila quelle porzioni di bytecode in linguaggio macchina. Questo processo è noto come compilazione JIT (Just-In-Time), ovvero a tempo di esecuzione.

Quando si progetta un linguaggio, è quindi fondamentale stabilire dei compromessi. È necessario trovare un equilibrio tra le attività svolte offline e quelle svolte online, garantendo che il tempo di compilazione rimanga accettabile. Inoltre, bisogna assicurarsi che l'occupazione di spazio del programma compilato sia adeguata, evitando che diventi eccessivamente ingombrante.

Esempio. I template in C++ generano nuovo codice ogni volta che viene specificato un nuovo tipo per il template, causando il fenomeno del **code bloat**.
