01-interpreti-compilatori

Interprete e Compilatore

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

Un **compilatore** 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).

> Per eseguire il compilato serve un interprete per il linguaggio M

Il compilatore traduce il programma in modo da ottenere un miglioramento di qualche metrica (tempo di esecuzione, memoria usata, consumo energetico, ...).

Interprete VS Compilatore

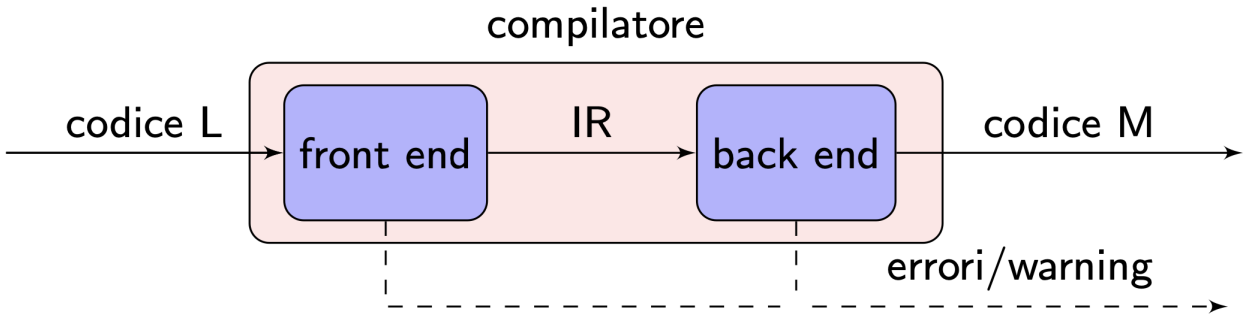
La compilazione è un'attività che viene compiuta off-line (non a run-time) questo perché il compilatore deve identificare alcuni errori di programmazione prima dell'esecuzione del programma, deve migliorare l'efficienza e deve rendere utilizzabili alcuni costrutti dei linguaggi ad alto livello.

Esistono approcci che sono tipicamente compilati (C, C++; Pascal, ...), approcci tipicamente interpretati (PHP, Matlab, ...) e approcci misti (Java, Python, SQL, ...).

Quindi è meglio interpretare o compilare? In realtà non c'è un migliore, dipende dalle necessità, l'importante è stabilire compromessi:

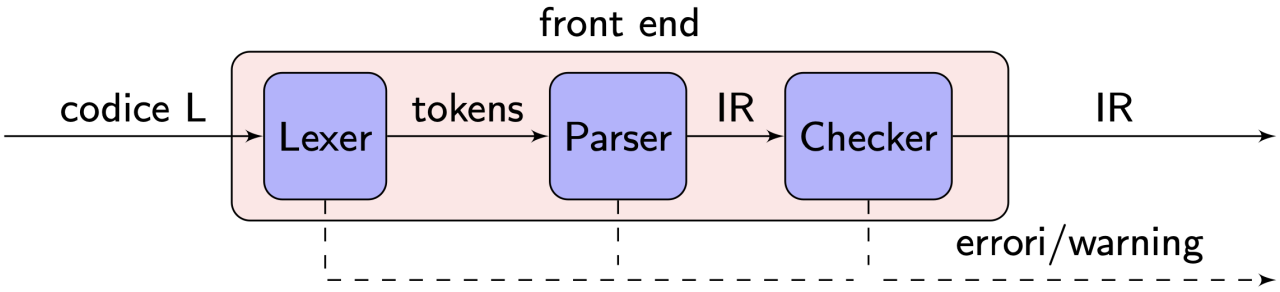
- Bilanciamento tra attività off-line ed on-line
- Il tempo di compilazione dev'essere accettabile
- L'occupazione in spazio del programma compilato deve essere accettabile

Struttura compilatore



Front-End

Il front-end è in grado di riconoscere programmi validi (e invalidi), segnalare errori e waring facilmente leggibili e produce codice IR (rappresentazione intermedia).
Esso si decompone nel seguente modo:



Come primo passo abbiamo il **Lexer**, che si occupa dell'analisi lessicale, questo ha il compito di suddividere il codice L in una sequenza di **token**, che darà in pasto al **Parser**.
Questo si occupa di eseguire l'analisi sintattica (correttezza della *struttura*) indipendente dal contesto, esso produce una rappresentazione intermedia (IR) e quindi il suo compito è riordinare la sequenza di token secondo una sintassi definita, spesso però l'analisi sintattica non è sufficiente per garantire la completa traduzione ed entra in aiuto il **Checker**.

Quest'ultimo si occupa dunque dell'analisi semantica (correttezza del *significato*) dipendente dal contesto.

Lexer

Prende in input una sequenza di caratteri e restituisce in output una sequenza di token, costruiti in questo modo: *token* = $\langle \text{parte del discorso}, \text{lessema} \rangle$, ovvero parte del discorso indica di che tipologia è la parola che ho letto, mentre il lessema è la parola letta.
Esempi: $\langle KWD, \text{ while} \rangle$ (ho letto la keyword while), $\langle IDENT, \text{ somma} \rangle$, $\langle INT, 42 \rangle$, $\langle STR, \text{"Hello"} \rangle$.

Come definisco in modo rigoroso quali sono i token validi? Lo faccio definendo un linguaggio adeguato come le espressioni regolari che permettono di identificare schemi specifici all'interno di linguaggi, quindi un linguaggio comprensibile all'essere umano.
Quando implementiamo un lexer dobbiamo includere un riconoscitore che si occupa dunque di identificare e classificare i token, esso è un DFSA, cioè un automa a stati finiti deterministico, questo viene spesso generato automaticamente partendo dalla specifica.

Esempio:

- DIGIT = [0-9]
- LETTER = [a-zA-Z] | [_]
- ID = LETTER (LETTER | DIGIT)*
  Abbiamo i digit che è un numero da 0 a 9, lettere che è una lettera dell'alfabeto minuscola o maiuscola oppure l'underscore e infine abbiamo gli ID che sono composti da una lettere seguita da una sequenza di lettere o numeri che possono apparire zero o più volte.

Parser

Prende in input una sequenza di token e restituisce in output una rappresentazione IR della struttura sintattica, creando un AST (Abstract Syntax Tree).
Viene definito un linguaggio adatto per implementarlo, utilizzando grammatiche libere dal contesto, significa che quando leggo un carattere non mi interessa cosa viene prima o dopo di esso perché appunto sto affrontando un analisi sintattica.
Il riconoscitore che dovremmo implementare è un PDA (automa a pila non deterministico).

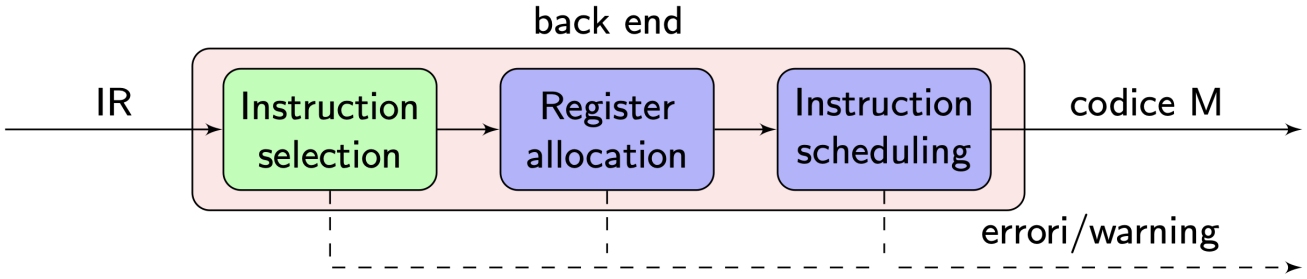
Checker

In input abbiamo un AST generato dal parser che viene definito grezzo in quanto il checker ha il compito di arricchirlo ed ultimarlo restituendolo in output.
Come posso definire in modo rigoroso quali programmi sono validi? Lo posso fare tramite il

linguaggio naturale servendomi dello **standard** del linguaggio, della documentazione del compilatore ecc., utilizzando anche semantiche formali (sistemi di regole).

Back-End

Si occupa di tradurre da IR a linguaggio M (macchina), decide quali valori mantenere nei registri, sceglie le istruzioni per implementare le operazioni e rispetta l'interfaccia di sistema. Esso viene decomposto nel modo seguente:



Instruction Selection

Si occupa di traduzione della rappresentazione intermedia (IR) del codice sorgente in istruzioni specifiche per l'architettura di destinazione (la macchina su cui eseguiamo il codice).

Register allocation

Si occupa di assegnare variabili e temporanei a registri del processore in modo efficiente, massimizzando l'uso di queste risorse e minimizzando l'accesso alla memoria. Le tecniche di register allocation sono cruciali per migliorare le prestazioni del codice generato, riducendo il tempo di esecuzione e l'overhead associato all'accesso alla memoria (LOAD e STORE).

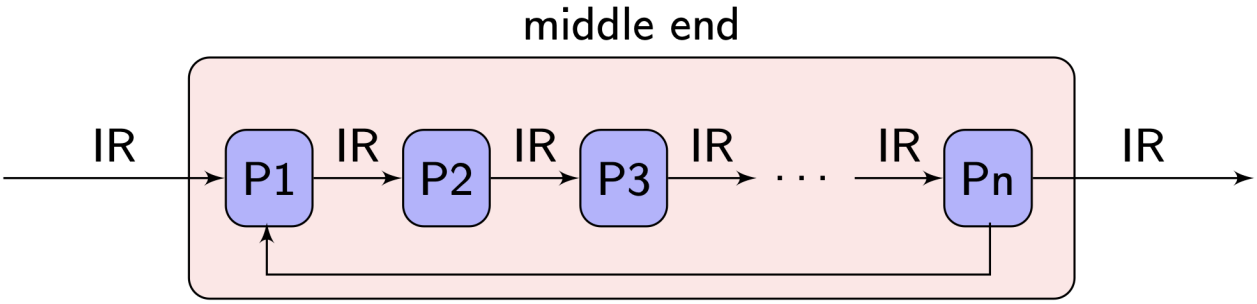
Instruction Scheduling

Si occupa di ottimizzare l'ordine delle istruzioni per migliorare l'efficienza dell'esecuzione. Attraverso tecniche di scheduling statico e dinamico, il compilatore cerca di massimizzare l'uso delle risorse del processore, ridurre i tempi di attesa e minimizzare il numero totale di cicli di clock necessari per l'esecuzione del programma.

Middle-End

Abbiamo anche una fase intermedia tra front-end e back-end, chiamata appunto middle-end. Il suo compito è analizzare il codice IR e trasformarlo, con l'obiettivo di migliorarlo preservando la semantica del programma.

Il middle-end viene decomposto in più passi, ognuno di essi implementa un'**analisi** o **trasformazione** dell'IR e un passo può dipendere o invalidare altri passi.



Passi di analisi

- identificazione di valori costanti
- identificazione di codice o valori inutili
- analisi di aliasing

Passi di trasformazione

- propagazione di valori costanti
- rimozione di codice inutile
- inlining di chiamate a funzione
- loop unrolling ("srotolo" i loop per poi ottimizzarli)