

# Linguaggi e compilatori

## Corso di Laurea in Informatica

Mauro Leoncini

A.A. 2023/2024

## 1 Nozioni introduttive

- Informazioni generali
- Compilatori ed interpreti

## 2 Struttura del compilatore

- La struttura attraverso un esempio

# Linguaggi e compilatori

## 1 Nozioni introduttive

- Informazioni generali
- Compilatori ed interpreti

## 2 Struttura del compilatore

- La struttura attraverso un esempio

# Informazioni di contesto

- Tutte le news importanti e/o urgenti saranno “postate” su Moodle
- Gli iscritti su Moodle riceveranno il link al repository (gdrive) contenente il materiale didattico preparato dai docenti, che dunque NON starà su moodle
- MS Teams NON sarà utilizzato
- Ricevimento in presenza: mercoledì ore 12:00. A richiesta su appuntamento (a distanza)
- CFU assegnati: 12 (circa 300 ore di lavoro), ripartiti in due moduli da 6 CFU
- Esame: un progetto, che verrà suddiviso in diversi assegnamenti intermedi, e un orale sugli argomenti affrontati nelle due parti in cui è organizzato il corso.

# Testo per la parte teorica

- Per la prima parte del corso il testo di riferimento (non tutto necessario ma neppure sufficiente) è:

Aho A.V.; Lam M.S.; Sethi R.; Ullman J.D.

*Compilatori: Principi, tecniche e strumenti*

Pearson Ediz. MyLab. Con aggiornamento online

<https://he.pearson.it/catalogo/1079>

- Il libro ancora oggi è noto come il *Dragon Book*, per via delle immagini presenti sulla copertina della prima versione



# Riferimenti per la parte di laboratorio

- In laboratorio svilupperemo il front-end di un semplice linguaggio per l'infrastruttura *LLVM*.
- Per iniziare il riferimento è <https://llvm.org/>
- Si tenga presente che in molte distribuzioni (Debian inclusa) LLVM è installabile mediante il relativo packet manager
- Durante il semestre seguiremo “abbastanza dettagliatamente” il tutorial alla pagina:  
<https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html>

# Obiettivi formativi

Conoscenze su:

- struttura e funzionamento di un compilatore moderno
- tecniche e algoritmi utilizzati nel processo di compilazione

Competenze operative su:

- strumenti automatici di ausilio alla costruzioni di compilatori
- progetto di un compilatore completo per un linguaggio semplificato
- soluzioni programmatiche che incidono sulle prestazioni del programma finale

# Prerequisiti

- Algoritmi e strutture dati, in particolare ricorsione, alberi e relativi algoritmi di visita
- Programmazione in C++ e padronanza del paradigma ad oggetti
- Nota: alcuni elementi del C++ verranno presentati/ripresi in questo stesso insegnamento
- Architettura dei calcolatori. Per questa prima parte è sufficiente un po' di dimestichezza con i linguaggi "tipo assembly"



# Linguaggi e compilatori

## 1 Nozioni introduttive

- Informazioni generali
- Compilatori ed interpreti

## 2 Struttura del compilatore

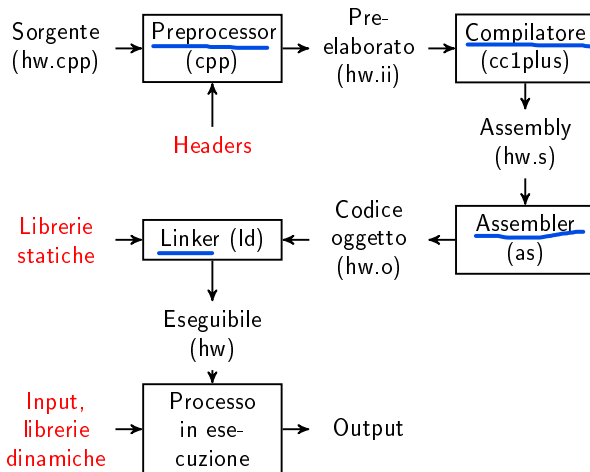
- La struttura attraverso un esempio

# Che cos'è un compilatore: definizioni e richiami d'uso

- Un compilatore per un linguaggio di programmazione  $L$  è un componente di una (cosiddetta) *toolchain* di programmi il cui obiettivo è creare eseguibili a partire da programmi scritti in  $L$ .
- Altri componenti che possono far parte (o che sicuramente fanno parte) della toolchain sono
  - precompilatore
  - assembler
  - linker (statico e dinamico)
- Anche se (di regola) non vengono invocati esplicitamente, questi ultimi programmi sono componenti a se stanti utilizzabili in modo separato.

# La toolchain completa

- Schema dei passi che portano dal sorgente `hw.cpp` (il classico “Hello World”) al processo in esecuzione



# Compilazione con GCC

- GCC è un termine abbreviato comune per *GNU Compiler Collection*, una suite in grado di compilare non solo programmi in C/C++ ma anche sorgenti in *Fortran* e *Ada*.
- Originariamente, però, GCC significava *GNU C Compiler* e in effetti facendo riferimento a GCC a volte si intende proprio il compilatore C e/o C++.
- In relazione a questi ultimi linguaggi, e all'ambiente Linux, gli elementi della toolchain GCC sono riferiti nel modo seguente:
  - *cpp* è nome con cui si indica il preprocessore C/C++;
  - *cc1/cc1plus* è il nome del compilatore vero e proprio;
  - *as* è il nome del programma assemblatore;
  - *ld* è il nome del linker (linking statico e dinamico).
- Normalmente si usa un programma driver (es. *g++*) ma è istruttivo per una volta procedere all'invocazione esplicita dei programmi della toolchain

# Uso dei programmi della toolchain

- Preprocessing:  
`> cpp -o hw.i1 hw.cpp`
- Compilazione:  
`> cc1plus -o hw.s hw.i1 2>/dev/null`
- “Assemblaggio”:  
`> as -o hw.o hw.s`
- Linking:  
`> ld ...`

## Note

- 1 `cc1plus` quasi certamente non è nel search path. Nelle distribuzioni Debian si trova “attualmente” (settembre 2023) in:  
`/usr/libexec/gcc/x86_64-linux-gnu/XX/` (XX indica la release)
- 2 I parametri da passare al linker sono numerosi. Nella prossima slide vediamo come determinarli

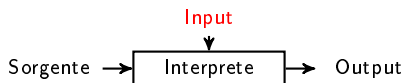
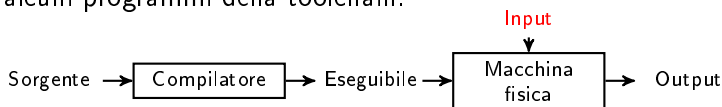
# I parametri del linker

- Per determinarli, usiamo il driver g++ con l'opzione -v  
> g++ -v -o hw hw.o
- Il driver invoca il linker usando il comando collect (o collect2), invocazione che viene evidenziata in fondo alla lunga sequenza di dati prodotti
- I parametri da inserire nella chiamata esplicita di ld sono proprio quelli utilizzati nella chiamata di collect
- È di norma possibile escludere i plugin e usare i parametri a partire da -build-id

```
> ld -build-id -eh-frame-hdr -m elf_x86_64 \  
> -hash-style=gnu -as-needed \  
> -dynamic-linker /lib64/ld-linux-x86-64.so.2 \  
> ...  
> /usr/lib/x86_64-linux-gnu/crti.o
```

# Compilatori vs interpreti

- Un diverso modo per “implementare” un linguaggio di programmazione è costituito dall’*interpretazione*.
- Un interprete fornisce l’impressione di eseguire direttamente il programma in linguaggio sorgente.
- I seguenti schemi permettono di avere una comprensione immediata delle differenze fra le due implementazioni di un linguaggio, compilata e interpretata.
- Rispetto a quanto visto, lo schema di compilazione astrae il contributo di alcuni programmi della toolchain.



# Realizzazioni alternative di un interprete

- Un *interprete puro* legge il testo sorgente di un programma, lo analizza e lo esegue mentre procede.
- Questo procedimento è inefficiente: l'interprete spende infatti molto tempo nell'analisi dell'input (testuale).
- Un tale interprete deve riconoscere e analizzare ogni espressione nel testo di partenza ogni volta che la incontra, in modo da eseguire ciò che essa prescrive.
- Questa alternativa “pura” è stata utilizzata per l'implementazione di pochissimi linguaggi (tra cui il più importante è il *Lisp* originale).
- In generale, un'implementazione interpretata include anche un traduttore che è essenzialmente identico al *front-end* di un compilatore.
- Il risultato prodotto dal traduttore può essere più o meno vicino alla macchina fisica ed è proprio tale linea di demarcazione che caratterizza i diversi “interpreti”.



# Modello PERL

- Il linguaggio *PERL* (*Practical Extraction and Report Language*) è il paradigma di un'implementazione in cui la parte iniziale di traduzione produce una rappresentazione *ad albero* del programma, noto come *Abstract Syntax Tree (AST)*.
- L'interpretazione del programma potrebbe essere eseguita essenzialmente mediante una visita in *ordine posticipato* dell'AST, utilizzando opportune strutture dati a supporto.
- Per ragioni di efficienza, l'interprete PERL esegue dapprima svariate ottimizzazioni, ad esempio per rendere più efficiente l'attraversamento dell'albero.
- Alcune porzioni di codice, che dovranno essere eseguite più volte, vengono anche tradotte in codice macchina.

# Modello Java

- La tipica implementazione di Java (ma anche di Python) prevede che il traduttore produca codice eseguibile da una *Virtual Machine (VM)*.
- Tale codice, chiamato *bytecode*, può dunque essere visto come il linguaggio macchina della VM.
- Nel caso di PERL la percezione dell'utente è analoga a quella che si ha nell'interpretazione pura: traduttore e interprete appaiono come un unico programma e l'esecuzione avviene in risposta ad singolo comando.
- Nel caso di Java, invece, la traduzione in linguaggio bytecode e l'interpretazione avvengono in momenti distinti.
- È noto infatti che l'implementazione di Java consiste di due moduli software: il *compilatore Java* e il cosiddetto *Java Runtime Environment (JRE)*

# Linguaggi e compilatori

## 1 Nozioni introduttive

- Informazioni generali
- Compilatori ed interpreti

## 2 Struttura del compilatore

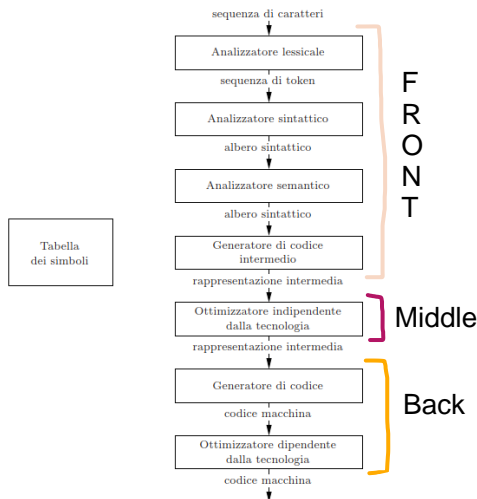
- La struttura attraverso un esempio

# I moduli del compilatore

- Da ora in avanti il termine *compilatore* si riferirà solo al modulo che effettua la traduzione da sorgente ad assembly (quindi non la toolchain completa).
- Un compilatore è strutturato in tre *moduli*
  - 1 front-end
  - 2 middle-end
  - 3 back-end
- Il *front-end* è la parte del compilatore *specializzata nel linguaggio*.
- Il front-end opera sul programma sorgente e produce una sua *rappresentazione intermedia*, indipendente dal linguaggio e dalla macchina.
- Il *middle-end* esegue un'ottimizzazione del codice intermedio machine-independent.
- Infine il *back-end* produce il codice per l'architettura target (eseguendo altre specifiche ottimizzazioni).

# Le fasi della compilazione

- La seguente figura è tratta dal *Dragon Book*
- Il front/middle/back-end è composto di 4/1/2 fasi



# Esempio di traduzione di un'istruzione

- Dal Dragon Book: fasi applicate ad una istruzione

- 1
- 2
- 3

position	...
initial	...
rate	...

Tabella dei simboli

```
position = initial + rate * 60
```

Analizzatore lessicale

```
<(id,1)> <=> <(id,2)> <+> <(id,3)> <*> <(60)>
```

Analizzatore sintattico

```

      =
     / \
  (id,1) +
       / \
    (id,2) *
         / \
      (id,3) 60
  
```

Analizzatore semantico

```

      =
     / \
  (id,1) +
       / \
    (id,2) *
         / \
      (id,3) inttofloat
                |
                60
  
```

Generatore di codice intermedio

```

t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
  
```

Ottimizzatore di codice

```

t1 = id3 * 60.0
id1 = id2 + t1
  
```

Generatore di codice

```

LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
  
```

# Struttura del front-end

- I moduli che compongono il front-end di un compilatore e le loro interazioni sono riassunte nel seguente schema
- La tabella dei simboli è un dizionario (tipicamente implementato come tabella hash) che memorizza i simboli via via incontrati nell'analisi del sorgente e le loro caratteristiche

