PROGRAMMAZIONE 2
15. Macchine astratte, linguaggi, interpretazione e compilazione

Linguaggi e astrazione

- I ***linguaggi di programmazione*** sono il più potente strumento di ***astrazione*** messo a disposizione dei programmatori
  - I linguaggi si sono evoluti trasformando in costrutti linguistici (e realizzandoli una volta per tutte nell'implementazione)
- **settori di applicazione** (basi di dati, web applications, intelligenza artificiale, simulazione, etc.)
- Di fondamentale importanza l'introduzione di ***meccanismi di astrazione***, che permettono di estendere un linguaggio ***programmando*** nuove operazioni, tipi di dato, etc.

Linguaggi di Programmazione

- Studiare i principi che stanno alla base dei linguaggi di programmazione
- Essenziale per comprendere il progetto, la realizzazione e anche l'applicazione pratica dei linguaggi
- Non ci interessa rispondere alla domanda "Java è meglio di C#"?

Tanti aspetti importanti…

- *Paradigmi linguistici:*
  - ***Imperativo, Funzionale, Orientato agli Oggetti***
- *Implementazione*: strutture a tempo di esecuzione
  - Quali sono le strutture del run-time?
  - Come vengono gestite?
  - Quali sono le relazioni tra paradigmi linguistici e strutture del run-time?
- *Il nostro approccio*: la descrizione dell'implememementazione del linguaggio è guidata dalla semantica formale!
  - Struttura del run-time simulata in **`Ocaml`**
- Ci sono numerosi libri sul tema che sono utili per il corso… ma metteremo a disposizione delle note

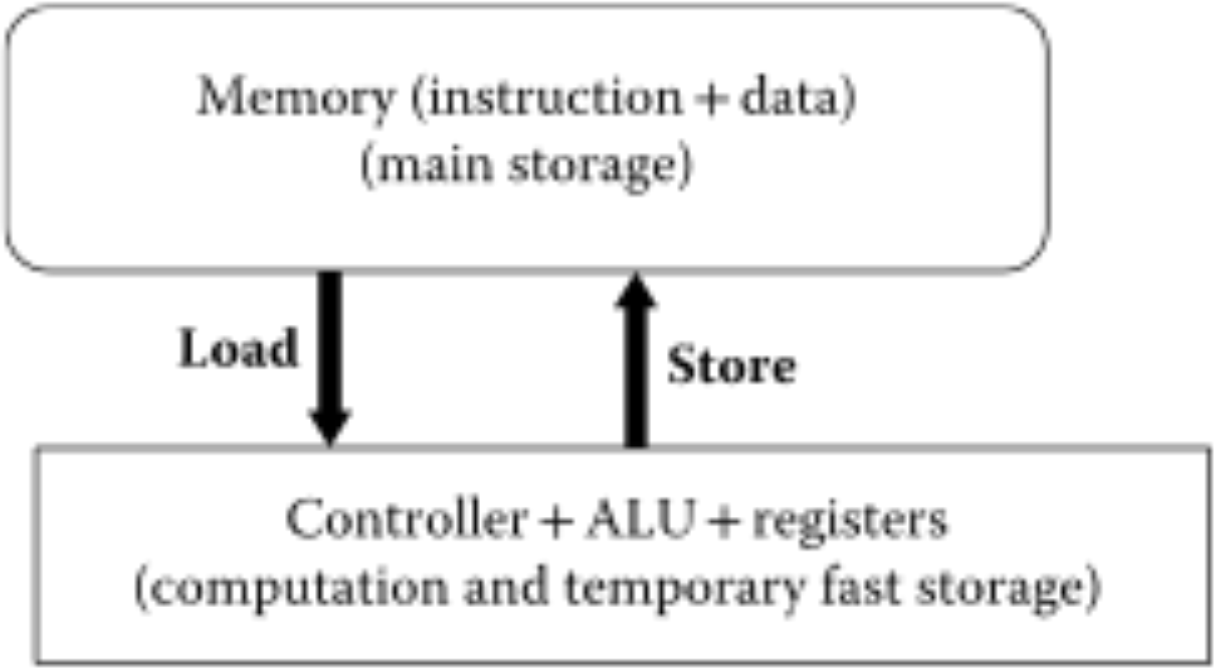
Materiale didattico (testo di riferimento)

M. Gabbrielli, S. Martini
Linguaggi di programmazione
(McGraw-Hill 2006)

Von Neumann: The first draft report…

- Il modello di **Von Neumann** è alla base della struttura dei computer attuali
- Due componenti principali
  - **Memoria**, dove sono memorizzati i programmi e i dati
  - **Unità centrale di elaborazione,** che ha il compito di eseguire i programmi immagazzinati in memoria prelevando le istruzioni (e i dati relativi), interpretandole ed eseguendole una dopo l'altra

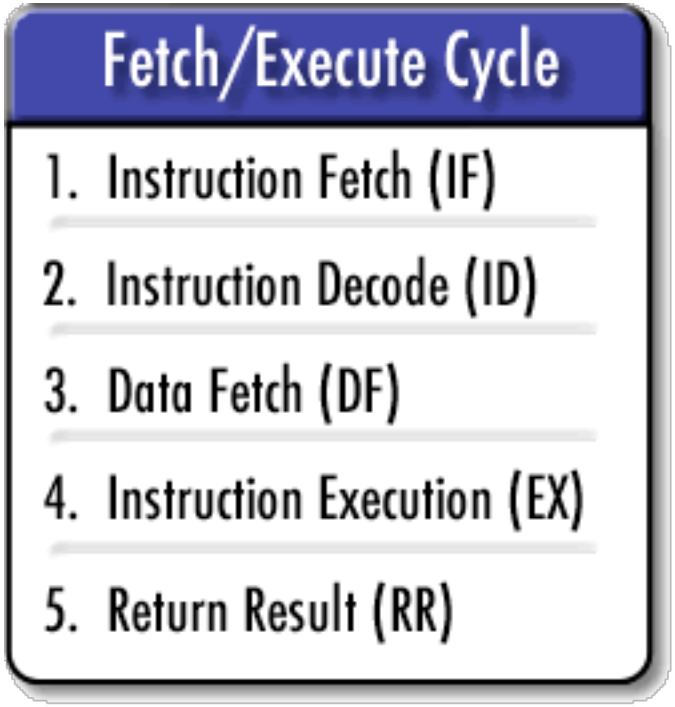
La macchina di Von Neumann



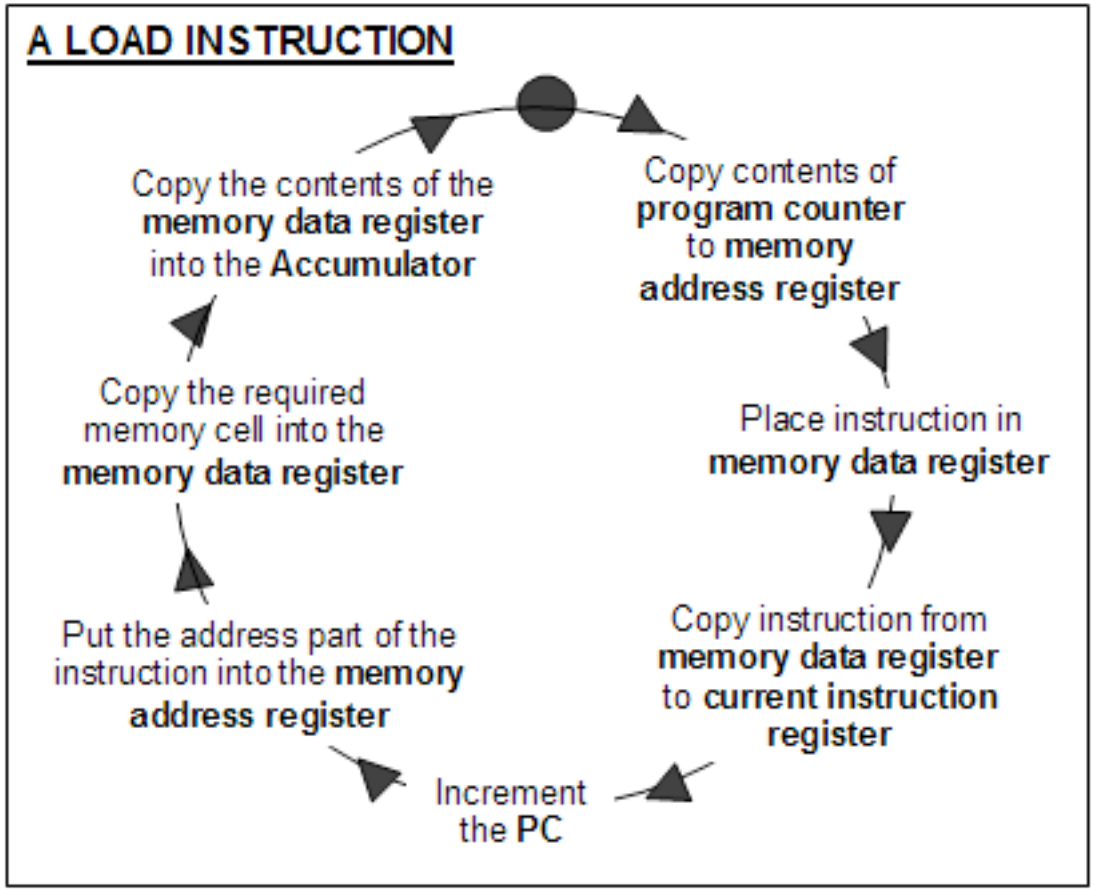
Ciclo Fetch-Execute

- **Fetch**: l' istruzione da eseguire viene prelevata dalla memoria e trasferita all'interno della CPU
- **Decode**: l' istruzione viene interpretata e vengono avviate le azioni interne necessarie per la sua esecuzione
- **Data Fetch**: sono prelevati dalla memoria i dati sui quali eseguire l'operazione prevista dalla istruzione
- **Execute**: è portata a termine l'esecuzione dell'operazione prevista dall'istruzione
- **Store**: è memorizzato il risultato dell'operazione prevista dall'istruzione

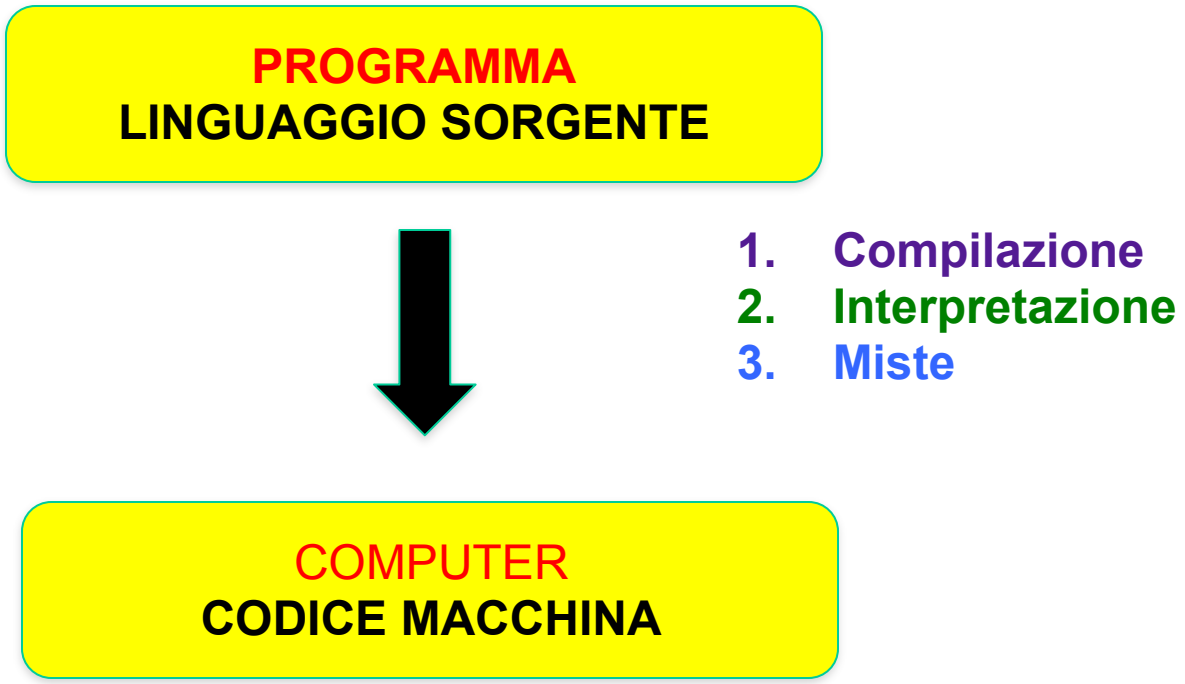
Ciclo Fetch-Execute



An example


A LOAD INSTRUCTION
Copy contents of program counter to memory address register
Place instruction in memory data register
Copy instruction from memory data register to current instruction register
Increment the PC
Put the address part of the instruction into the memory address register
Copy the required memory cell into the memory data register
Copy the contents of the memory data register into the Accumulator

Implementare i Linguaggi di Programmazione



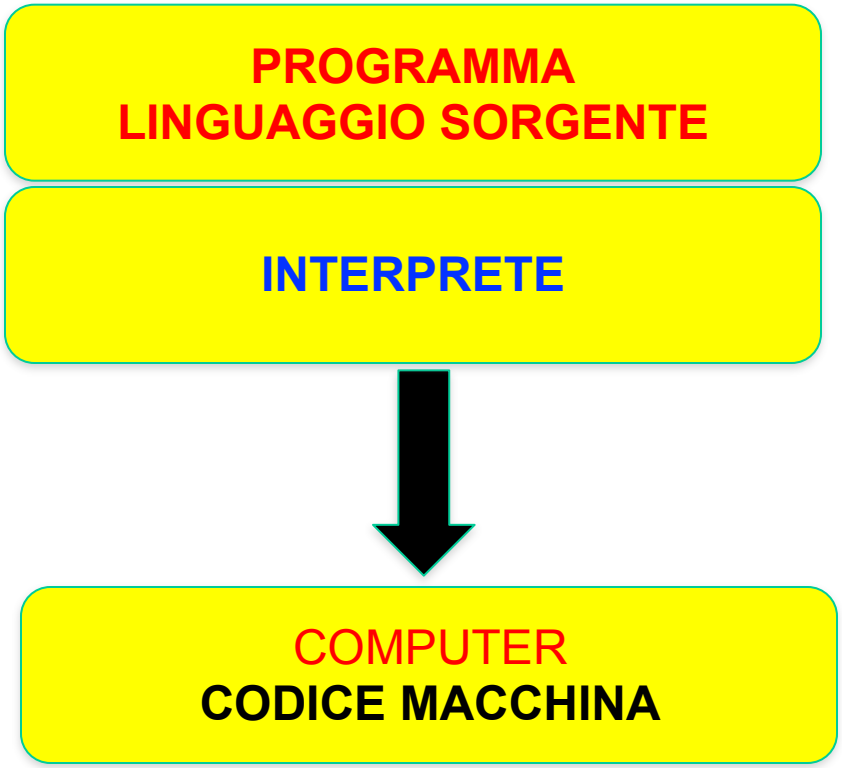
Implementare Linguaggi di Programmazione

PROGRAMMA
LINGUAGGIO SORGENTE

Compilazione: il codice sorgente viene tradotto in quello macchina

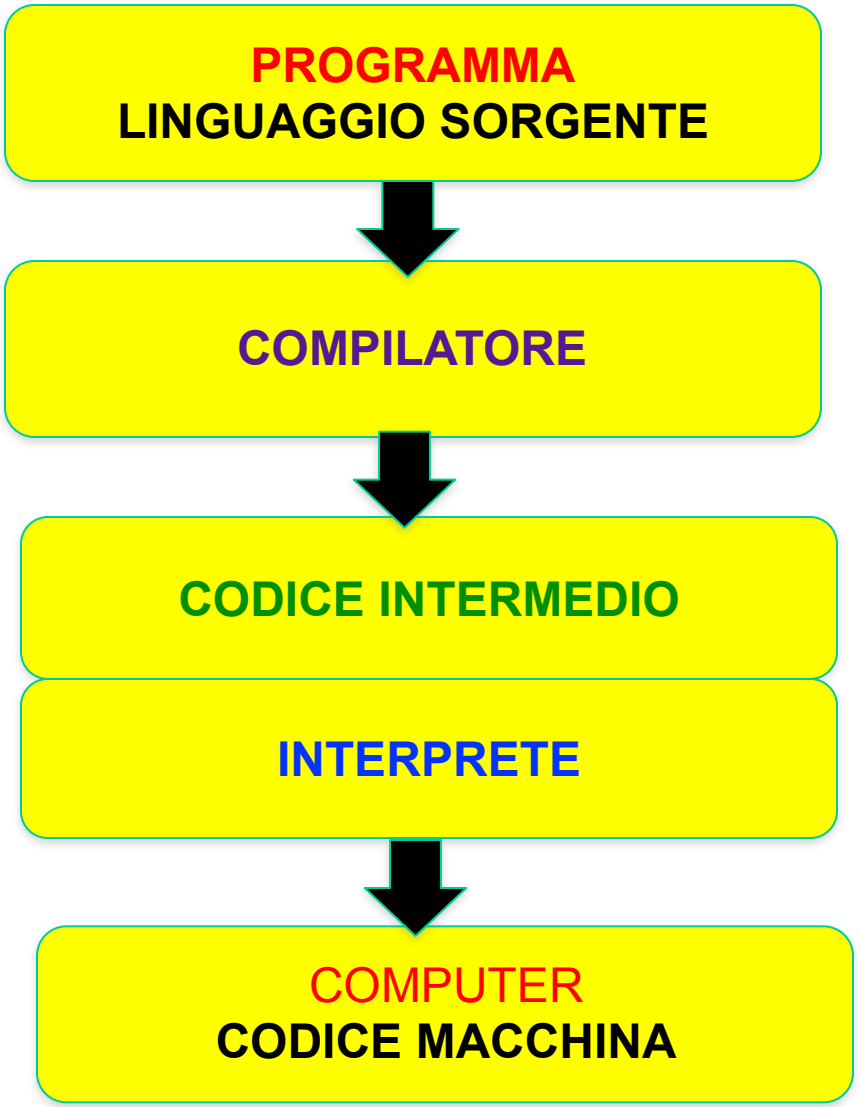
COMPUTER
CODICE MACCHINA

Implementare Linguaggi di Programmazione



Interpretazione:
macchina virtuale implementata su quella fisica che esegue le istruzioni del linguaggio sorgente

Implementare Linguaggi di Programmazione

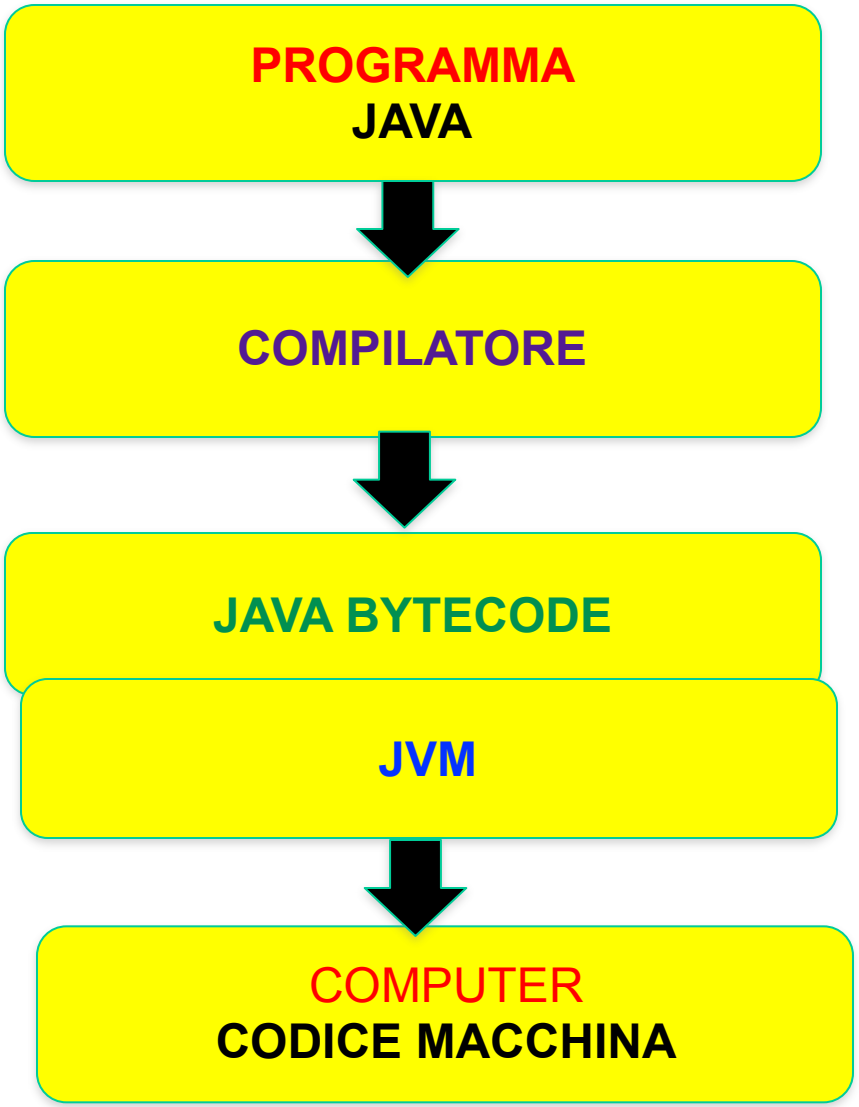


Compilazione

+

Interpretazione

Implementare Linguaggi di Programmazione



Compilazione

+

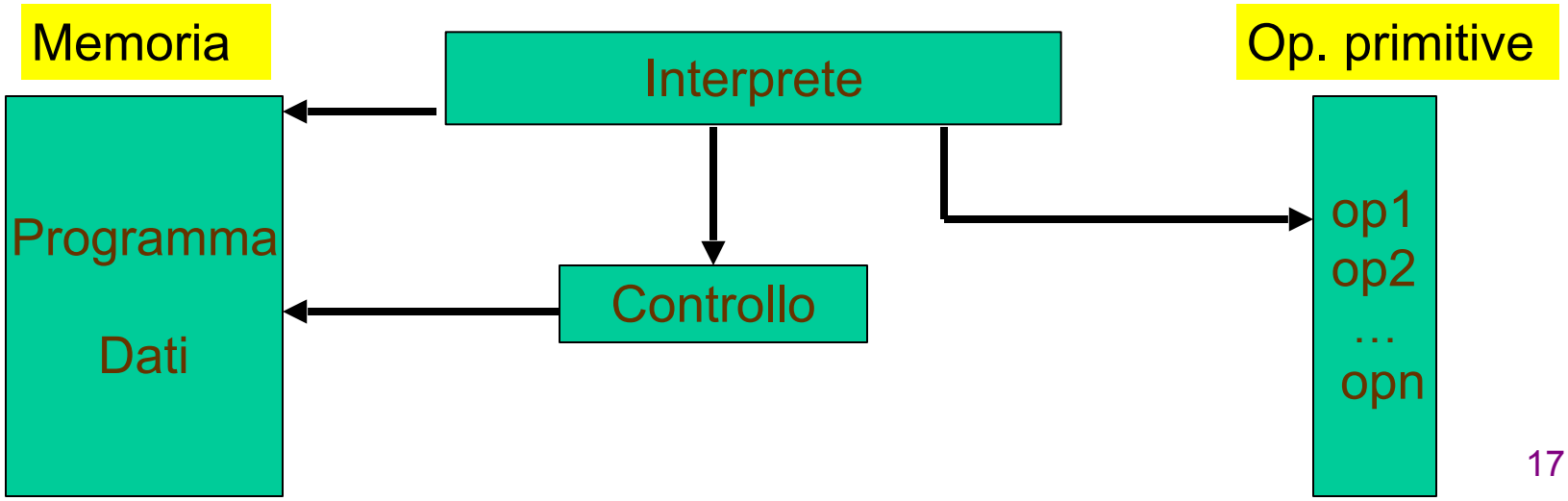
Interpretazione

Macchina astratta

- Macchina astratta: un **sistema virtuale** che rappresenta il comportamento di una macchina fisica individuando precisamente l'insieme delle risorse necessarie per l'esecuzione di programmi

Macchina astratta

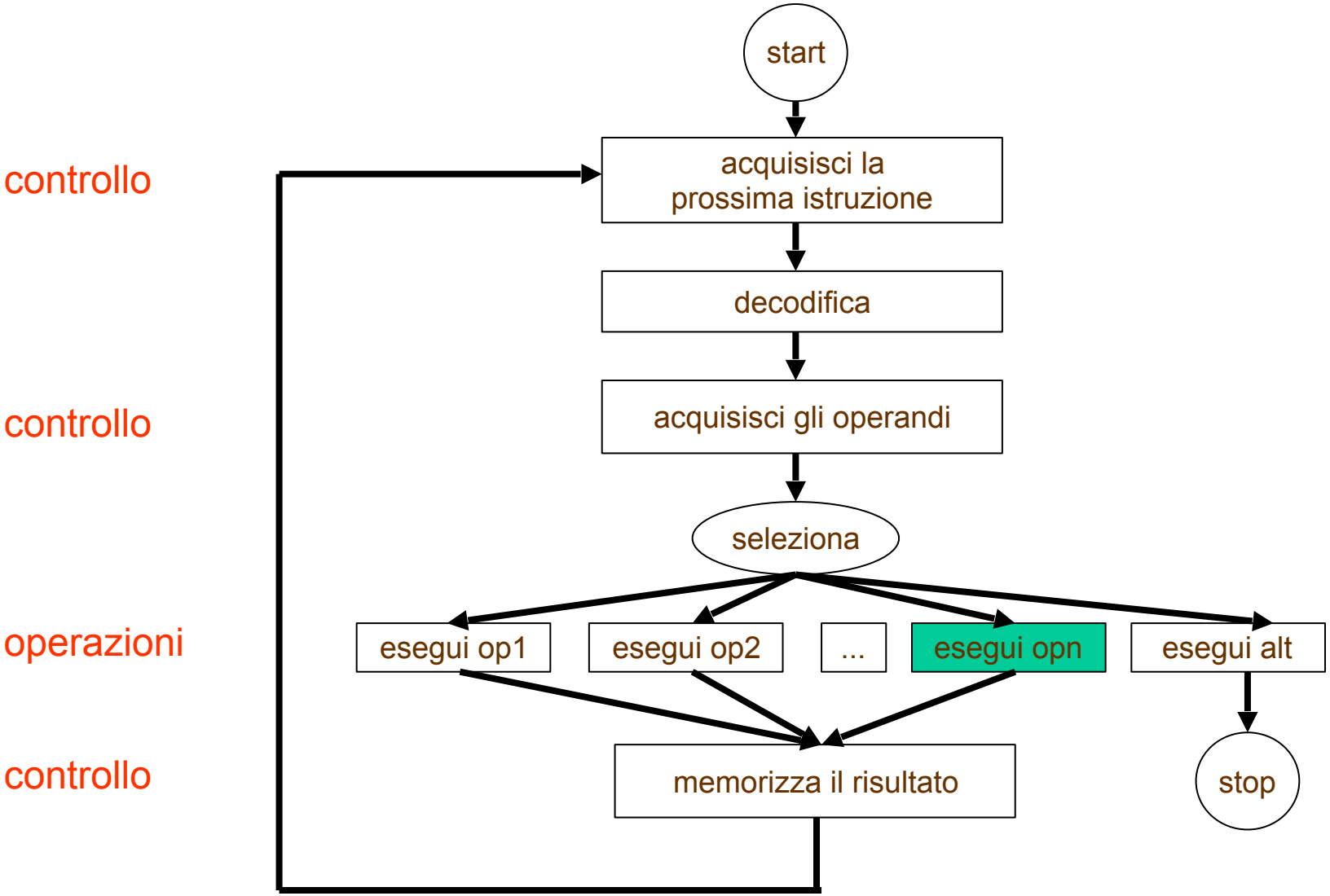
- Una collezione di strutture dati e algoritmi in grado di **memorizzare** ed **eseguire** programmi
- Componenti della macchina astratta
  - interprete
  - memoria (dati e programmi)
  - controllo
  - operazioni “primitive”



Componente di controllo

- Una collezione di strutture dati e algoritmi per
  - acquisire la prossima istruzione
  - acquisire gli operandi e memorizzare i risultati delle operazioni
  - gestire le chiamate e i ritorni dai sottoprogrammi
  - mantenere le associazioni fra nomi e valori denotati
  - gestire dinamicamente la memoria
  - …

L'interprete



Linguaggio di una macchina astratta

- **M** macchina astratta
- $\mathbf{L_M}$ linguaggio macchina di **M**
  - è il linguaggio che ha come stringhe legali tutti i programmi interpretabili dall'interprete di **M**
- I programmi sono particolari dati su cui opera l'interprete
- Alle componenti di **M** corrispondono componenti di L_M
- Tipi di dato primitivi
  - costrutti di controllo
    - ✓ per controllare l'ordine di esecuzione
    - ✓ per controllare acquisizione e trasferimento dati

Implementare macchine astratte

- **M** macchina astratta
- I componenti di **M** sono realizzati mediante strutture dati e algoritmi implementati nel linguaggio macchina di una **macchina ospite M_O**, già esistente (implementata)
- È importante la realizzazione dell'interprete di **M**
  - può coincidere con l'interprete di **M_O**
    - ✓ **M** è realizzata come **estensione** di **M_O**
    - ✓ altri componenti della macchina possono essere diversi
  - può essere diverso dall'interprete di **M_O**
  - M è realizzata su M_O in modo interpretativo
    - ✓ altri componenti della macchina possono essere uguali

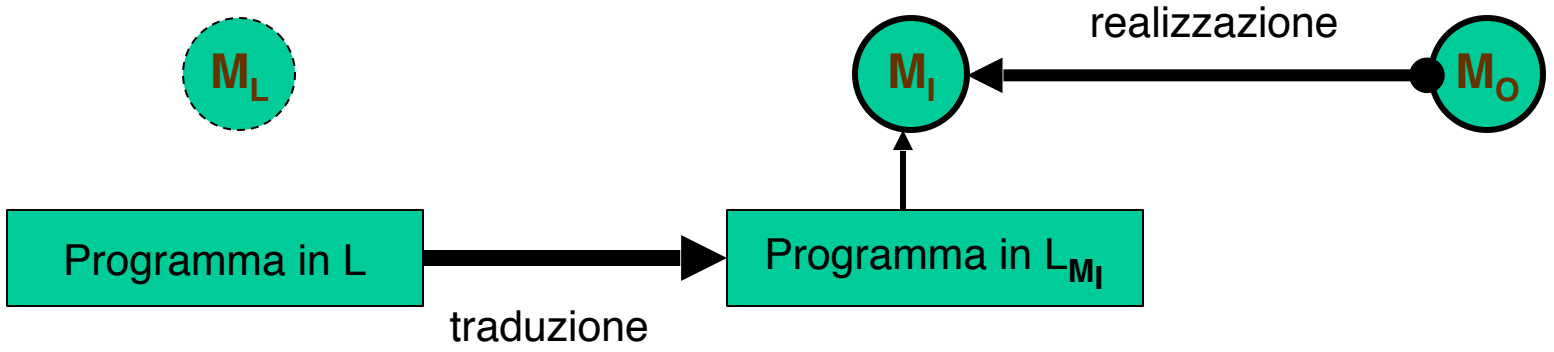
Dal linguaggio alla macchina astratta

- L **linguaggio** M_L **macchina** astratta di L
- **Implementazione di L =**
  realizzazione di M_L su una macchina ospite M_O
- Se **L** è un linguaggio ad alto livello e M_O una macchina "fisica"
  - l'interprete di M_L è necessariamente diverso dall'interprete di M_O
    - ✓ M_L è realizzata su M_O in modo interpretativo
    - ✓ l'implementazione di **L** si chiama **interprete**
    - ✓ esiste una soluzione alternativa basata su tecniche di traduzione (**compilatore**?)

Implementare un linguaggio

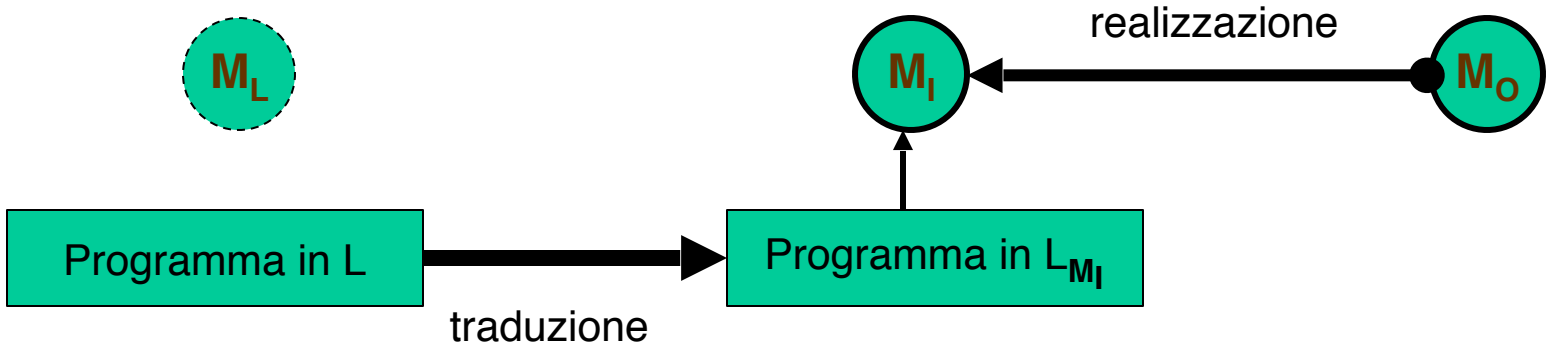
- L linguaggio ad alto livello
- M_L macchina astratta di L
- M_O **macchina ospite**
- **interprete** (puro)
  - M_L è realizzata su M_O in modo interpretativo
  - scarsa efficienza, soprattutto per colpa dell'interprete (ciclo di decodifica)
- **compilatore** (puro)
  - i programmi di **L** sono tradotti in programmi funzionalmente equivalenti nel linguaggio macchina di M_O
  - i programmi tradotti sono eseguiti direttamente su M_O
  - M_L non viene realizzata
  - il problema è quello della dimensione del codice prodotto
- Casi limite che nella realtà non esistono quasi mai

La macchina intermedia



- L linguaggio ad alto livello
- M_L macchina astratta di L
- **M_I macchina intermedia**
- **L_{M_I} linguaggio intermedio**
- M_O **macchina ospite**
  - traduzione dei programmi da L al linguaggio intermedio **L_{M_I}**
  - realizzazione della macchina intermedia **M_I** su M_O

La macchina intermedia



- M_L = M_I **interprete** (puro)
- M_O = M_I **compilatore** (puro)
  - possibile solo se la differenza fra M_O e M_L è molto limitata
    - L linguaggio assembler di M_O
    - in tutti gli altri casi, c'è sempre una macchina intermedia che estende eventualmente la macchina ospite in alcuni componenti

Il compilatore

- Quando l'interprete della macchina intermedia M_I coincide con quello della macchina ospite M_O
- Che differenza esiste tra M_I e M_O ?
  - il **supporto a tempo di esecuzione (rts)**
    - collezione di strutture dati e sotto-programmi che devono essere caricati su M_O (estensione) per permettere l'esecuzione del codice prodotto dal traduttore (compilatore)
  - M_I = M_O **+ rts**
- Il linguaggio L_{MI} è il linguaggio macchina di M_O esteso con chiamate al supporto a tempo di esecuzione

Il compilatore C

- Il supporto a tempo di esecuzione contiene
  - varie strutture dati
    - lo stack
      - ambiente, memoria, sottoprogrammi, …
    - la memoria a heap
      - puntatori, ...
  - i sotto-programmi che realizzano le operazioni necessarie su tali strutture dati
- Il codice prodotto è scritto in linguaggio macchina esteso con chiamate al rts

Implementazioni miste

- Quando l'interprete della macchina intermedia $\mathbf{M_I}$ è diverso da quello della macchina ospite M_O
- Esiste un ciclo di interpretazione del linguaggio intermedio $\mathbf{L_{M_I}}$ realizzato su M_O
- Si ottiene un codice tradotto più compatto
  - per facilitare la portabilità su più macchine ospiti
  - si deve reimplementare l'interprete del linguaggio intermedio
  - non è necessario reimplementare il traduttore

Compilatore o implementazione mista?

- Nel compilatore non c'è di mezzo un livello di interpretazione del linguaggio intermedio
  - sorgente di inefficienza
    - ✓ la decodifica di una istruzione nel linguaggio intermedio (e la sua trasformazione nelle azioni semantiche corrispondenti) viene effettuata ogni volta che si incontra l'istruzione
- Se il linguaggio intermedio è progettato bene, il codice prodotto da una implementazione mista ha dimensioni inferiori a quelle del codice prodotto da un compilatore
- Un'implementazione mista è più portabile di un compilatore
- Il supporto a tempo di esecuzione di un compilatore si ritrova quasi uguale nelle strutture dati e routine utilizzate dall'interprete del linguaggio intermedio

L'implementazione di Java

- È una implementazione mista
  - traduzione dei programmi da Java a **`bytecode`**, linguaggio macchina di una macchina intermedia: **`Java Virtual Machine`**
  - i programmi **`bytecode`** sono interpretati
  - l'interprete della **`Java Virtual Machine`** opera su strutture dati (stack, heap) simili a quelle del rts del compilatore C
    - la differenza fondamentale è la presenza di una gestione automatica del recupero della memoria a heap (garbage collector)
  - su una tipica macchina ospite, è più semplice realizzare l'interprete di **`bytecode`** che l'interprete di tutto il linguaggio
    - il **`bytecode`** è più "vicino" al tipico linguaggio macchina

Tre famiglie di implementazioni

- **Interprete (puro)**
  - M_L = M_I interprete di **L** realizzato su M_O
  - alcune implementazioni (vecchie!) di linguaggi logici e funzionali (LISP, PROLOG)
- **Compilatore(puro)**
  - macchina intermedia $\mathbf{M_I}$ realizzata per estensione sulla macchina ospite M_O (rts, nessun interprete) (C, C++, PASCAL)
- **Implementazione mista**
  - traduzione dei programmi da **L** a $\mathbf{L_{M_I}}$
  - il linguaggio intermedio viene interpretato su M_O
    - Java
    - i "compilatori" per linguaggi funzionali e logici (LISP, PROLOG, ML)
    - alcune (vecchie!) implementazioni di Pascal (Pcode)

Implementazioni miste: vantaggi

- La traduzione genera codice in un linguaggio più facile da interpretare su una tipica macchina ospite
- Ma soprattutto può effettuare una volta per tutte (a tempo di traduzione, staticamente) analisi, verifiche e ottimizzazioni che migliorano
  - l'affidabilità dei programmi
  - l'efficienza dell'esecuzione
- Varie proprietà interessate
  - inferenza e controllo dei tipi
  - controllo sull'uso dei nomi e loro risoluzione "statica"
  - …

Analisi statica

- Dipende dalla semantica del linguaggio
- Certi linguaggi (LISP) non permettono in pratica alcun tipo di analisi statica
  - a causa della regola di scoping dinamico nella gestione dell'ambiente non locale
- Linguaggi funzionali più moderni (ML) permettono di inferire e verificare molte proprietà (tipi, nomi, …) durante la traduzione, permettendo di
  - localizzare errori
  - eliminare controlli a tempo di esecuzione
    - type checking dinamico nelle operazioni
  - semplificare certe operazioni a tempo di esecuzione
    - come trovare il valore denotato da un nome

Analisi statica in Java

- Java è fortemente tipato
  - il type checking può essere in gran parte effettuato dal traduttore e sparire quindi dal bytecode generato
- Le relazioni di sub-typing permettono che una entità abbia un tipo vero (actual type) diverso da quello apparente (apparent type)
  - tipo apparente noto a tempo di traduzione
  - tipo vero noto solo a tempo di esecuzione
  - è garantito che il tipo apparente sia un super-type di quello vero
- Di conseguenza, alcune questioni legate ai tipi possono essere risolte solo a tempo di esecuzione
  - scelta del più specifico fra diversi metodi overloaded
  - casting (tentativo di forzare il tipo apparente a un possibile sotto-tipo)
  - dispatching dei metodi (scelta del metodo secondo il tipo vero)
- Controlli e simulazioni a tempo di esecuzione

Semantica formale e rts

- Due aspetti essenziali nella nostra visione (intendendo quella del corso) dei linguaggi di programmazione
  - **semantica formale**
    - ✓ eseguibile, implementazione ad altissimo livello
  - **implementazioni o macchine astratte**
    - ✓ interpreti e supporto a tempo di esecuzione

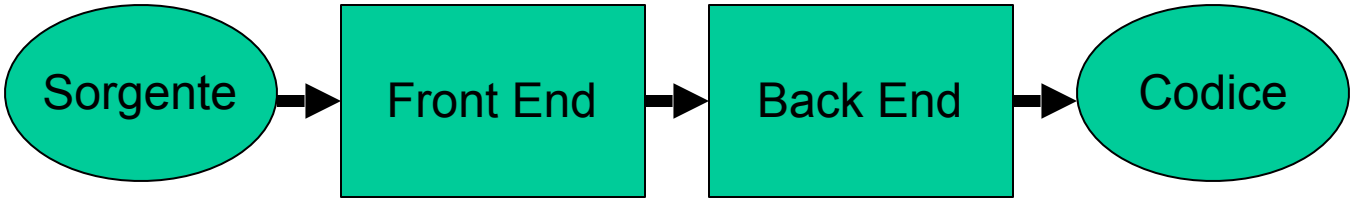
Perchè?

- Perché la **semantica formale**?
  - definizione precisa del linguaggio indipendente dall'implementazione
    - il progettista la definisce
    - l'implementatore la utilizza come specifica
    - il programmatore la utilizza per ragionare sul significato dei propri programmi
- Perchè **macchine astratte?**
    - il progettista deve tener conto delle caratteristiche possibili dell'implementazione
    - l'implementatore la realizza
    - il programmatore la deve conoscere per utilizzare al meglio il linguaggio

E il compilatore?

- La maggior parte dei corsi e dei libri sui linguaggi si occupano di compilatori
- Perché noi no?
  - il punto di vista dei compilatori verrà mostrato in un corso fondamentale della laurea magistrale
  - delle cose tradizionalmente trattate con il punto di vista del compilatore, poche sono quelle che realmente ci interessano
- Guardiamo la struttura di un tipico compilatore

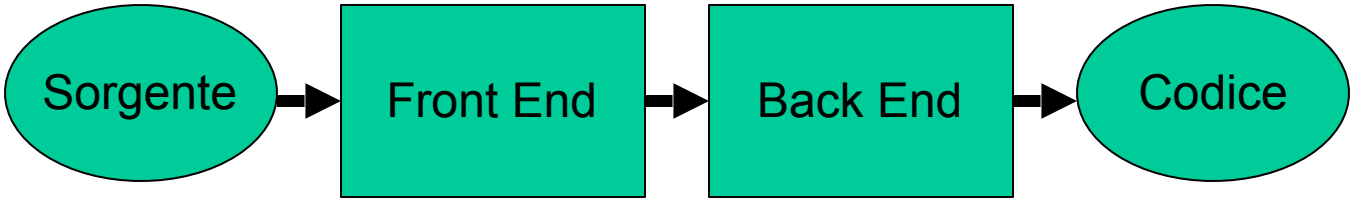
Compilatore



Front end: fasi di analisi
Legge il programma sorgente e determina la sua struttura sia sintattica che semantica

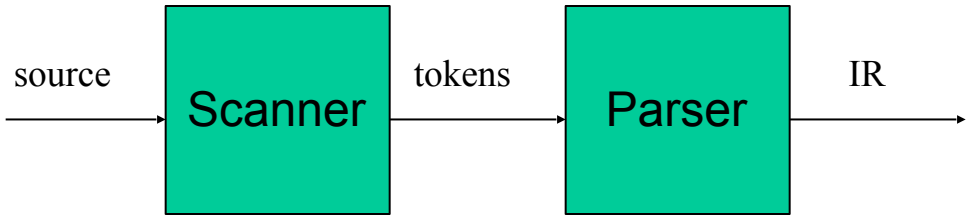
Back end: sintesi
Genera il codice nel linguaggio macchina, programma equivalente al programma sorgente

Compilatore



Aspetti critici
Riconoscere i programmi legali (sintatticamente corretti)
Gestire la struttura dei tipi
Genererare codice compatibile con il SO della macchina ospite

Front End



- Due fasi principali
  - Analisi lessicale (scanner): trasforma il programma sorgente nel lessico (token)
  - Analisi lessicale (parser): legge i token e genera il codice intermedio (IR)
- La teoria aiuta
  - la teoria dei linguaggi formali: automi, grammatiche
  - strumenti automatici per generare scanner e parser

Parser: output (IR)

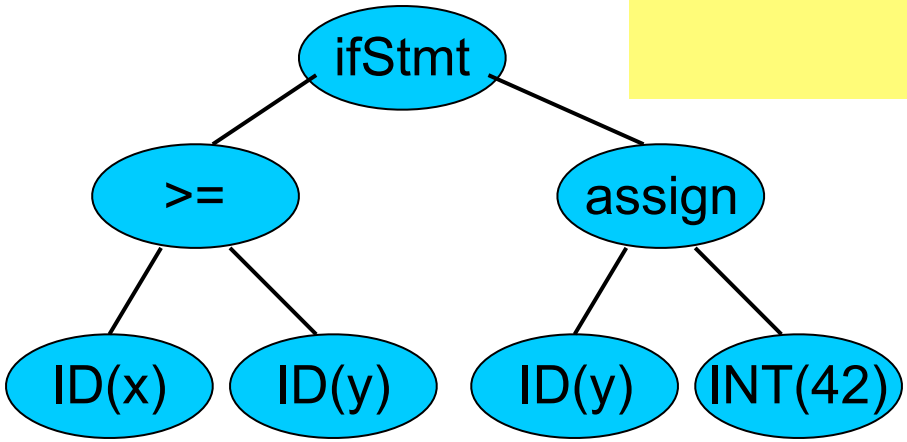
- Formati differenti
- Formato tipico riconosciuto: albero di sintassi astratta (abstract syntax tree)
  - la struttura sintattica essenziale del programma senza gli aspetti di zucchero sintattico
  - ne parleremo anche nel seguito

Parser: AST

- Abstract Syntax Tree (AST)

- Input
- // codice stupido

```
if (x >= y) y = 42;
```



IF | LPAREN | ID(x) | GEQ | ID(y)

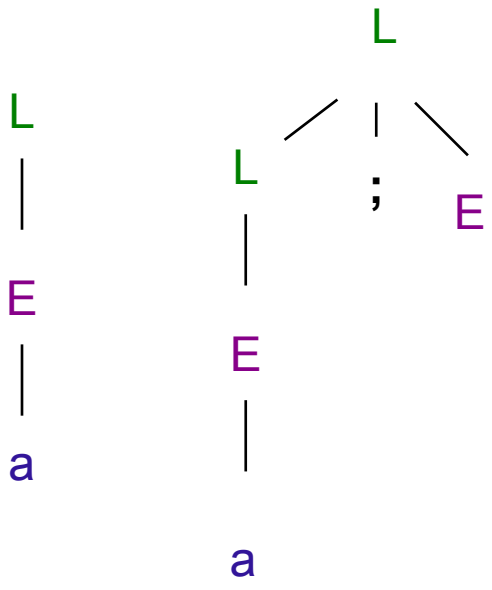
RPAREN | ID(y) | BECOMES | INT(42) | SCOLON

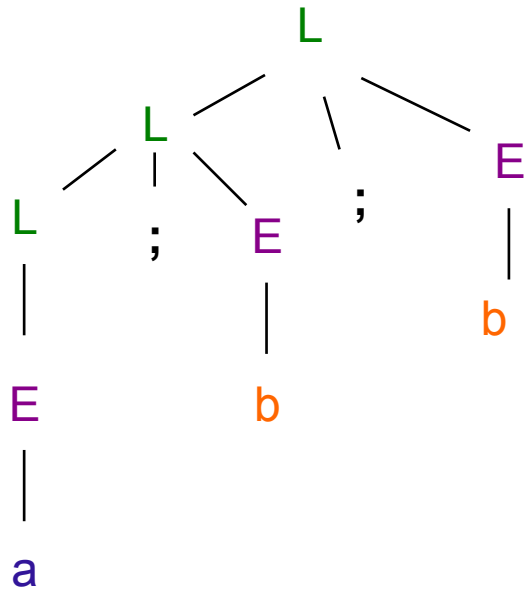
Gli alberi di sintassi astratta (AST)

- Gli alberi di sintassi astratta sono particolarmente rilevanti perché mostrano la struttura semantica significativa dei programmi
- Noi nel seguito consideremo sempre la sintassi astratta!!
  - Senza considerare gli aspetti di dettaglio quali precedenza operatori, ambiguità, etc.

AST: esempi

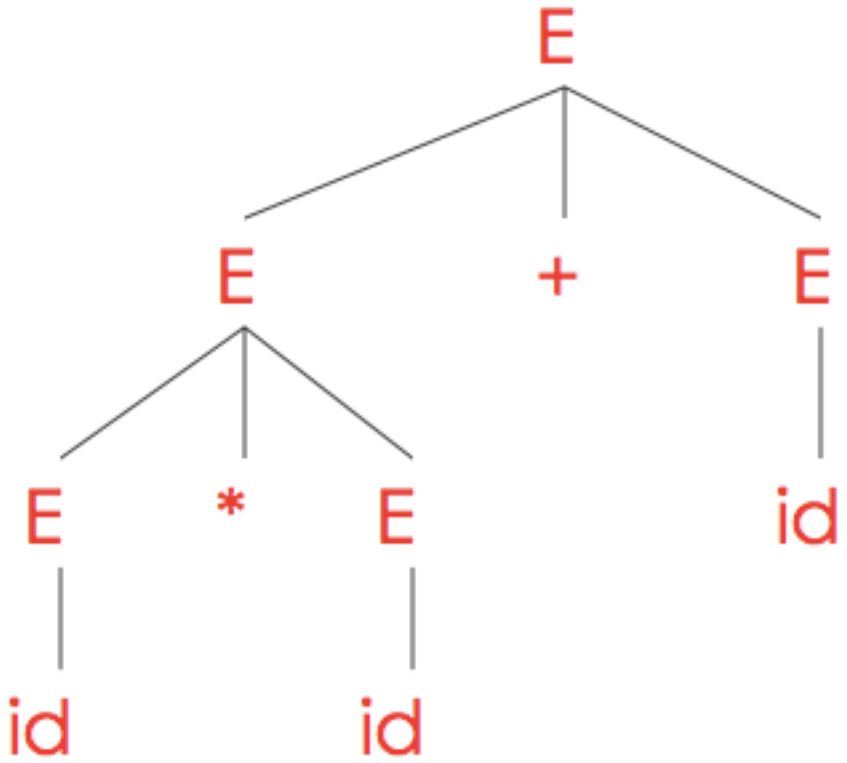
G: L -> L ; E | E
E -> a | b





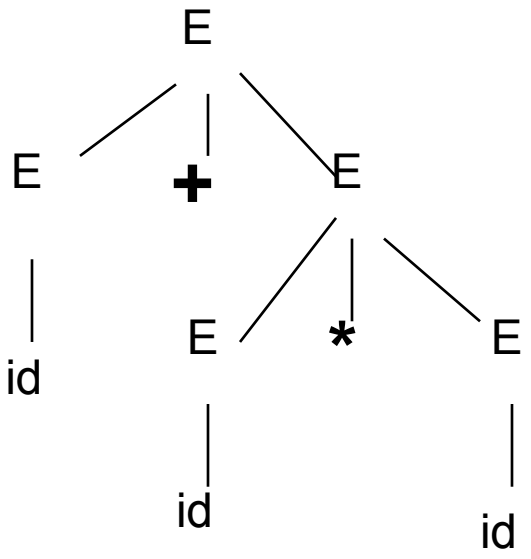
Derivazioni e AST

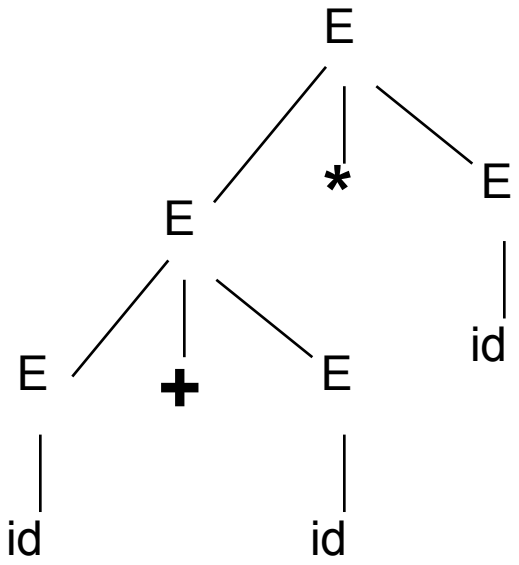
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E*E+E \\
\rightarrow\ & id*E+E \\
\rightarrow\ & id*id+E \\
\rightarrow\ & id*id+id
\end{aligned}
$$



Ambiguità

- Programma corretto con AST diversi
- Esempio
  - **E -> E+E | E*E | id**





Come si risolve?

- Esistono più metodi
- Ad esempio, codificare nelle regole della grammatica la precedenza degli operatori

$$E \rightarrow E' + E \mid E'$$

$$E' \rightarrow id * E' \mid id \mid (E) * E' \mid (E)$$

Morale

- La teoria (grammatiche e linguaggi formali) aiuta a strutturare le grammatiche in modo tale da evitare problemi come quello dell'ambiguità
  - … e tanti altri ancora
- Tutte queste problematiche le vedrete nella magistrale…

Sintassi astratta

- La sintassi astratta di un linguaggio è espressa facilmente coi tipi di dato algebrici di **Ocaml**
  - ogni categoria sintattica diventa un tipo di dato algebrico di **Ocaml**

BNF

```
BoolExp =
  | True
  | False
  | NOT BoolExp
  | BoolExp AND BoolExp
```

Algebraic Data Type

```
Type BoolExp =
   | True
   | False
   | Not of BoolExp
   | And of BoolExp * BoolExp
```

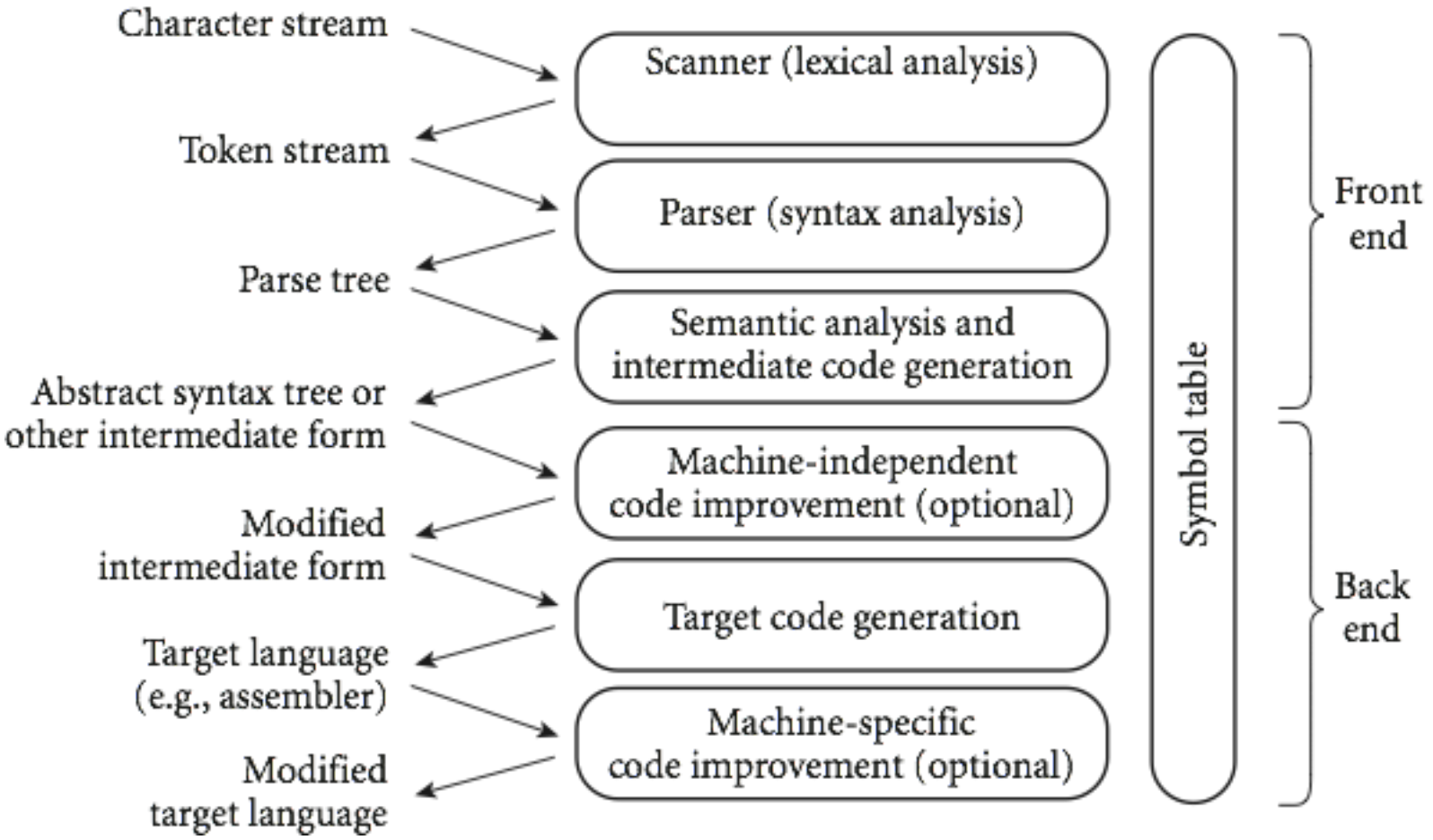
Esempio

Nome	Produzione grammaticale
EAdd	Exp ::= Exp "+" Exp1 ;
ESub	Exp ::= Exp "-" Exp1 ;
EMul	Exp1 ::= Exp1 "*" Exp2 ;
EDiv	Exp1 ::= Exp1 "/" Exp2 ;
EInt	Exp2 ::= Integer ;

```
type exp =
          EAdd of exp * exp
        | ESub of exp * exp
        | EMul of exp * exp
        | EDiv of exp * exp
        | EInt of int
```

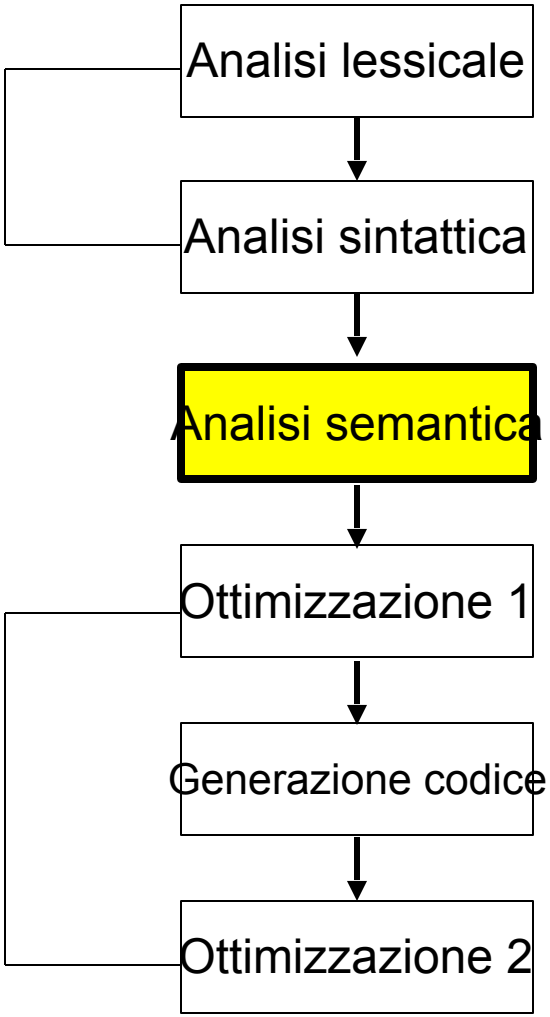
Mettiamo insieme le cose

Struttura di un compilatore



Cosa ci interessa?

non ci interessano: aspetti sintattici



non ci interessano: aspetti di generazione codice

Supporto a runtime

Solo la parte in giallo!!