Macchine astratte, linguaggi, interpretazione e compilazione

Macchina di Von Neumann

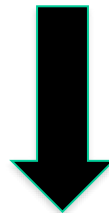
- Il modello di **Von Neumann** è alla base della struttura dei computer attuali
- Due componenti principali
  - **Memoria**, dove sono memorizzati i programmi e i dati
  - **Unità centrale di elaborazione**, che ha il compito di eseguire i programmi immagazzinati in memoria prelevando le istruzioni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

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

Implementazione Linguaggi di Programmazione

PROGRAMMATORE
LINGUAGGIO SORGENTE

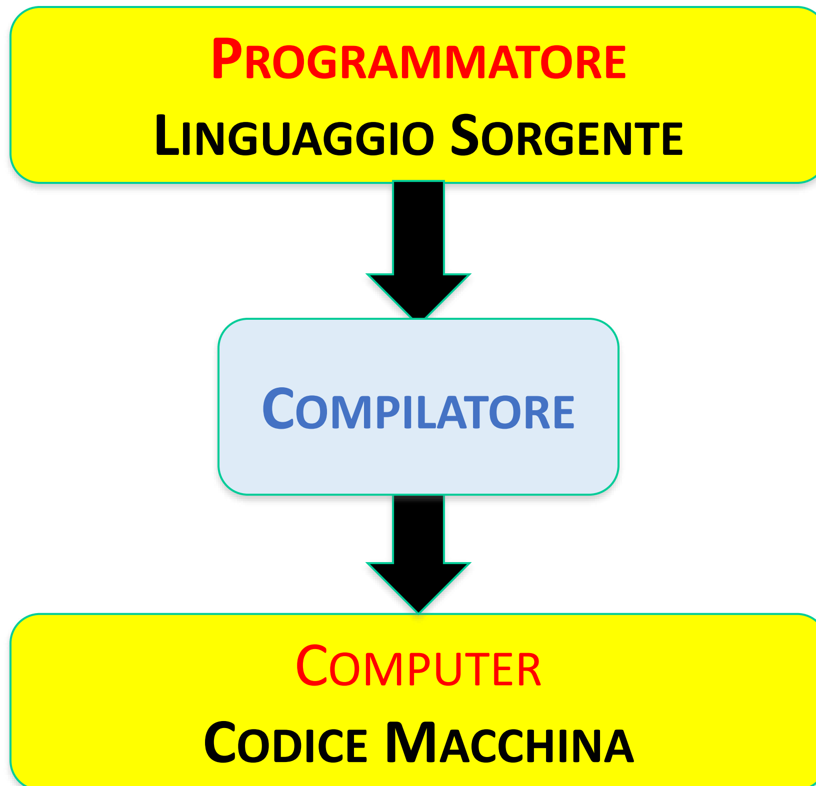


COMPUTER
CODICE MACCHINA

Tre approcci alternativi:

1. **Compilazione**
2. **Interpretazione**
3. **Misto**

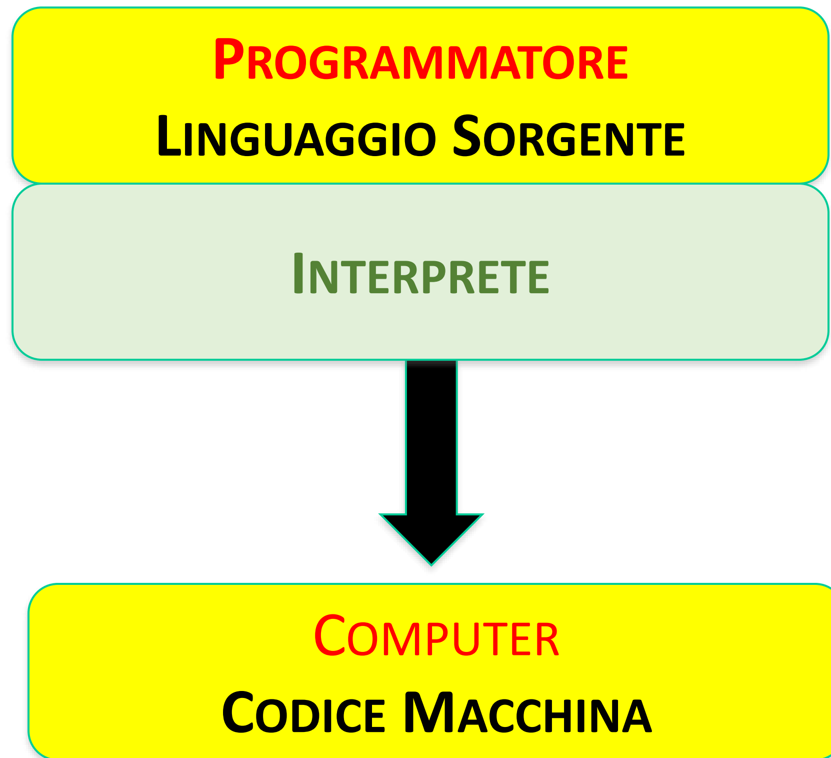
Implementazione Linguaggi di Programmazione



Compilazione:
Si traduce il codice sorgente in codice macchina

Esempi: C, C++

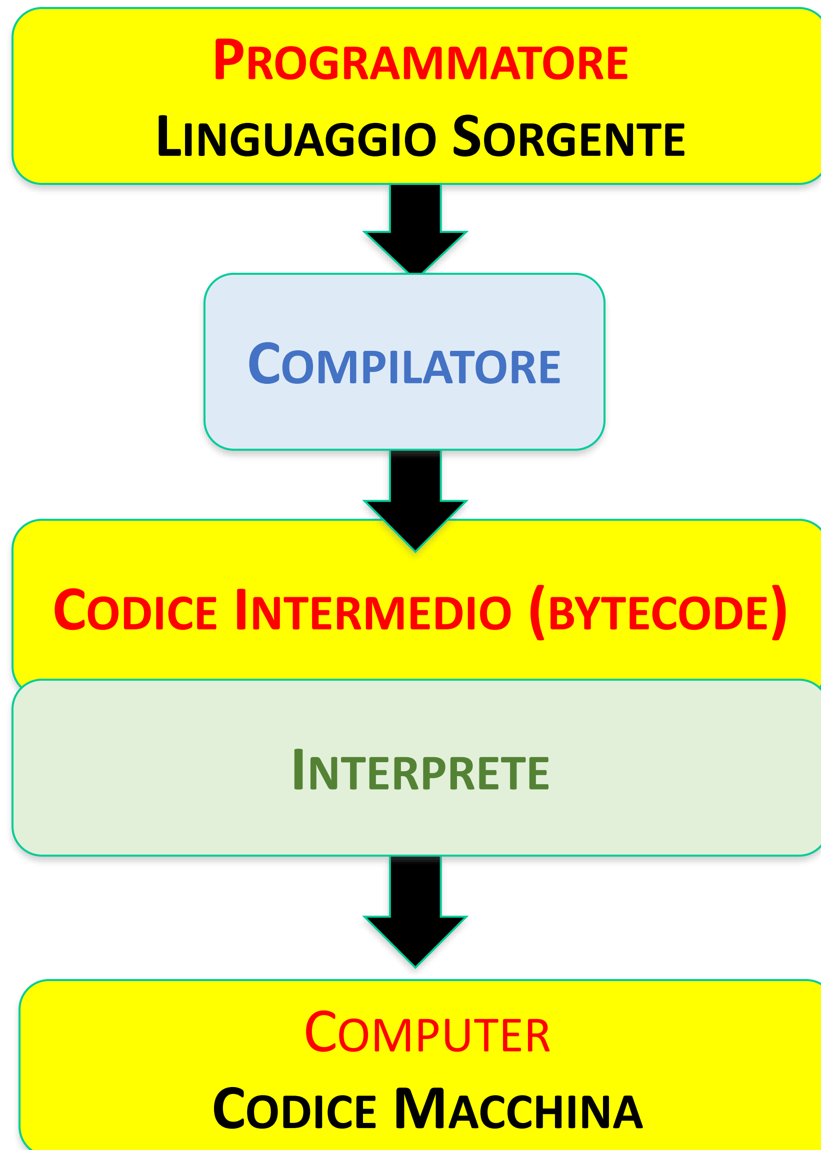
Implementazione Linguaggi di Programmazione



Interprete:
Implementazione software che esegue le istruzioni del linguaggio sorgente
-- analogo del ciclo fetch execute, ma a livello linguaggio sorgente
-- l'interprete è tipicamente compilato in codice macchina

Esempio: JavaScript

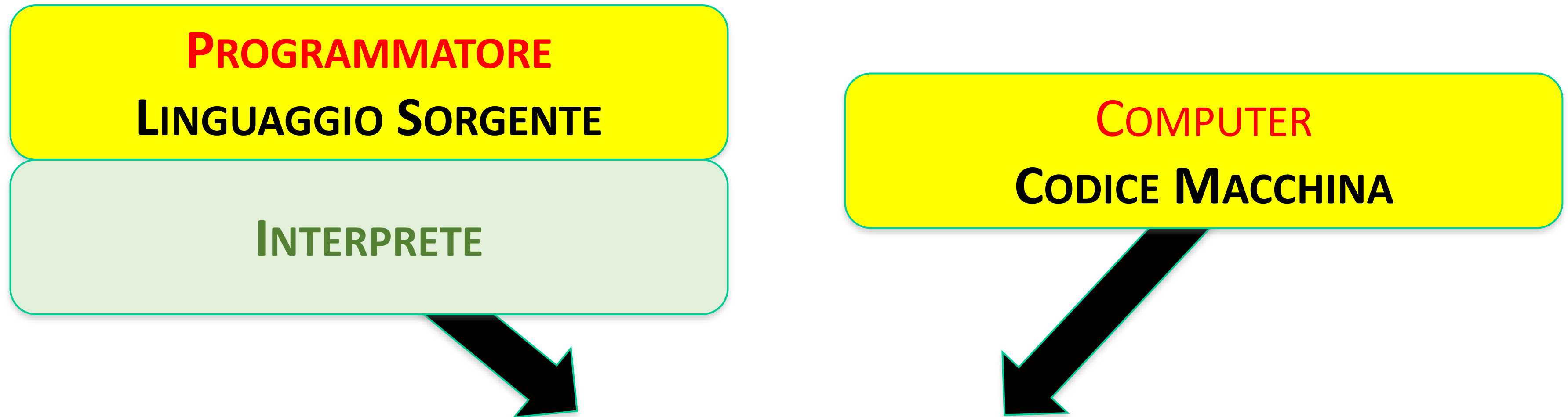
Implementazione Linguaggi di Programmazione



Approccio misto:
Compilazione
+
Interpretazione

Esempi: Java, C#, Python, TypeScript (il cui linguaggio intermedio è JavaScript)

Macchina Astratta



Macchine Astratte

L'interprete e il computer sono entrambi esecutori di programmi espressi in un dato linguaggio. Possiamo generalizzarli usando il concetto di macchina astratta.

Macchina: consente l'esecuzione step-by-step dei programmi

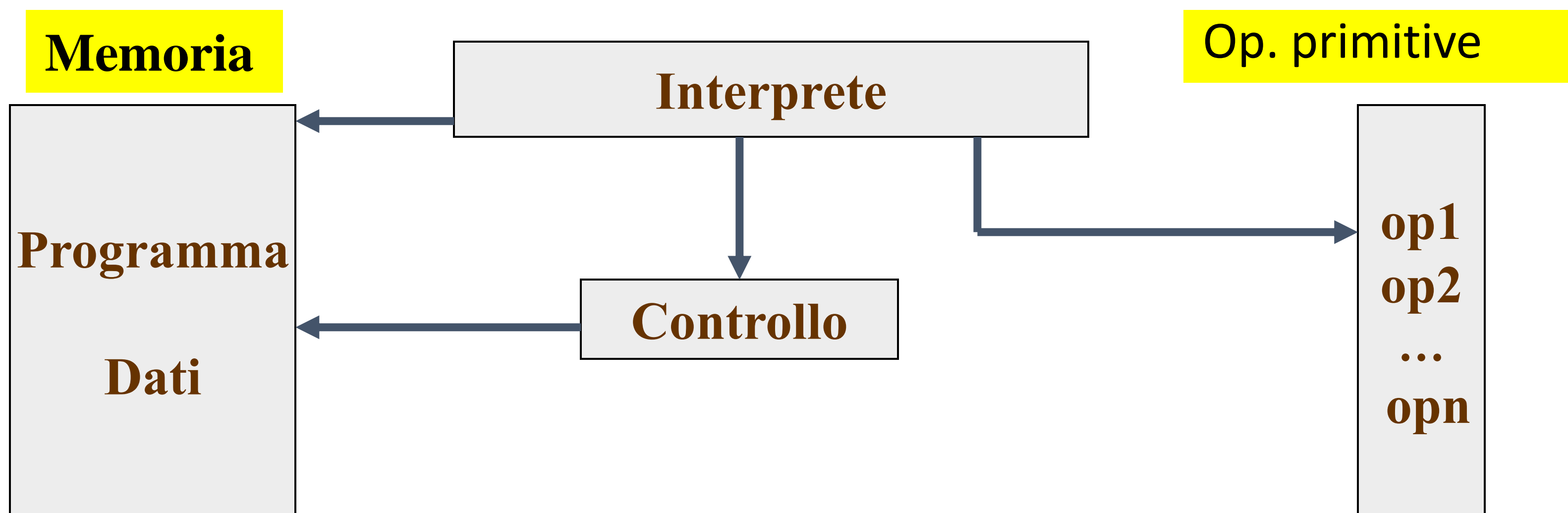
Astratta: omette i molti dettagli delle macchine reali.

Macchine Astratte

- **Macchina Astratta**: un **sistema virtuale** che rappresenta il comportamento di una macchina fisica **individuando**
  - l'insieme delle risorse necessarie per l'esecuzione di programmi
  - Un insieme di istruzioni specificatamente progettato per operare con queste risorse

Macchine astratta: visione fondazionale

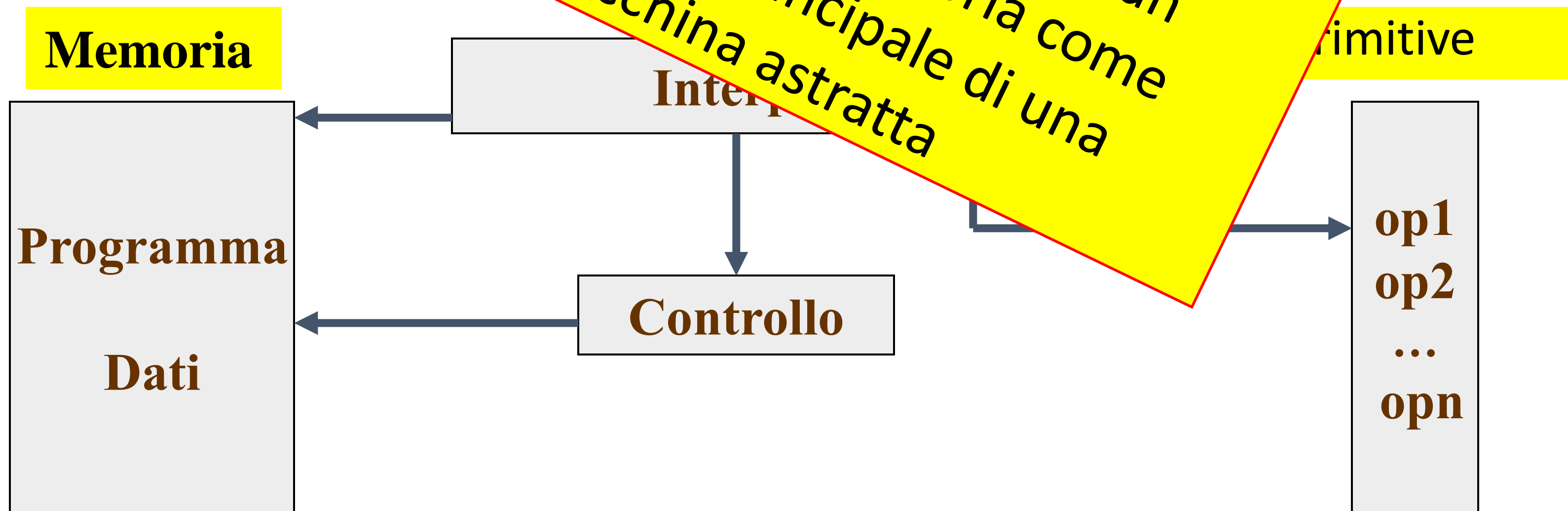
- Una collezione di **strutture dati e algoritmi** in grado di **memorizzare** ed **eseguire** programmi
- Componenti della macchina astratta
  - interprete
  - memoria (dati e programmi)
  - controllo
  - operazioni "primitive"



Macchine astratta: visione fondazionale

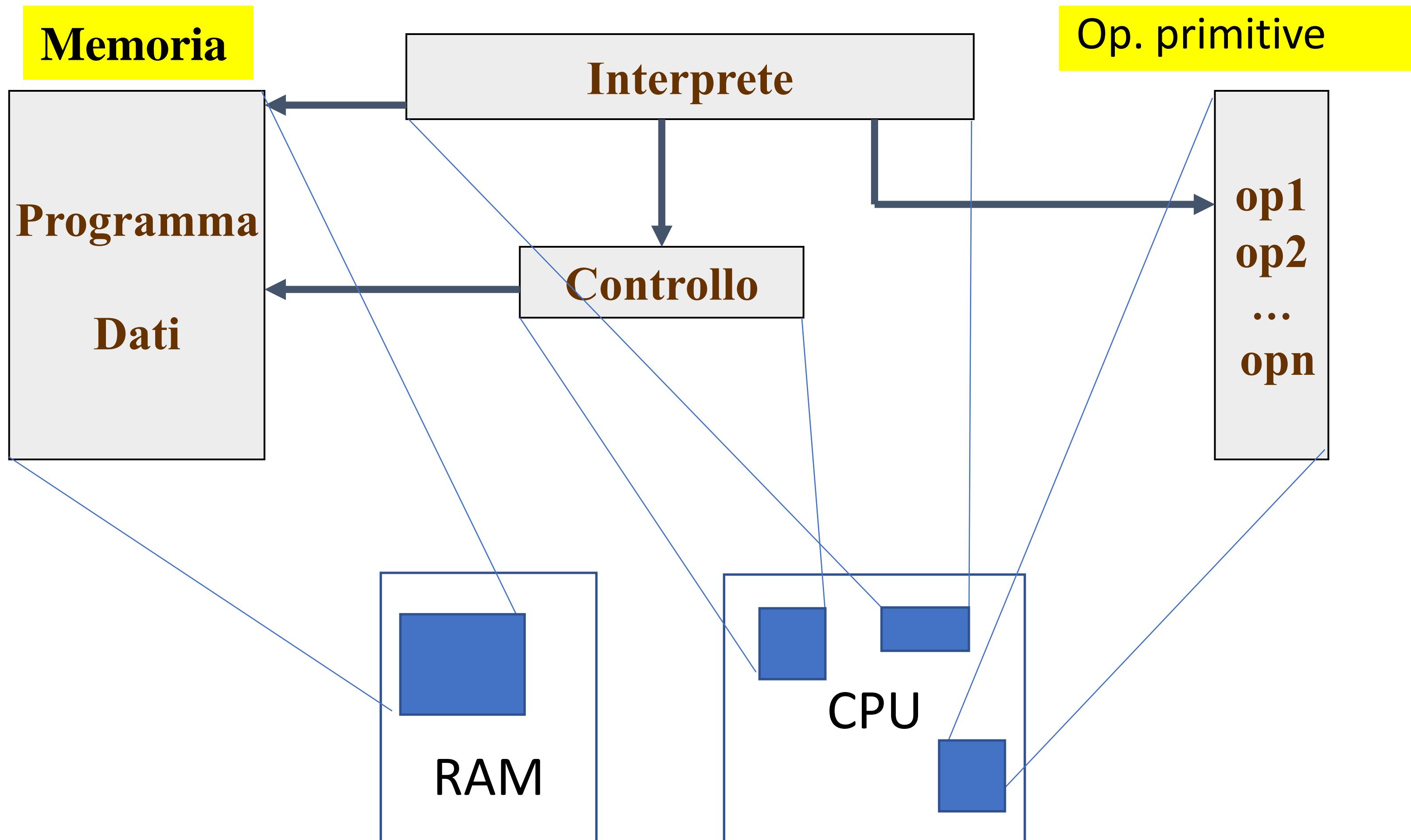
- Una collezione di ...tture dati e algoritmi in grado di **memorizzare** ed **eseguire** progra...
- Componenti de...
  - interprete
  - memoria (...
  - controllo...
  - operazioni...

ATTENZIONE: useremo il termine "interprete" sia per l'implementazione software di un esecutore di un linguaggio, sia nella teoria come componente principale di una macchina astratta



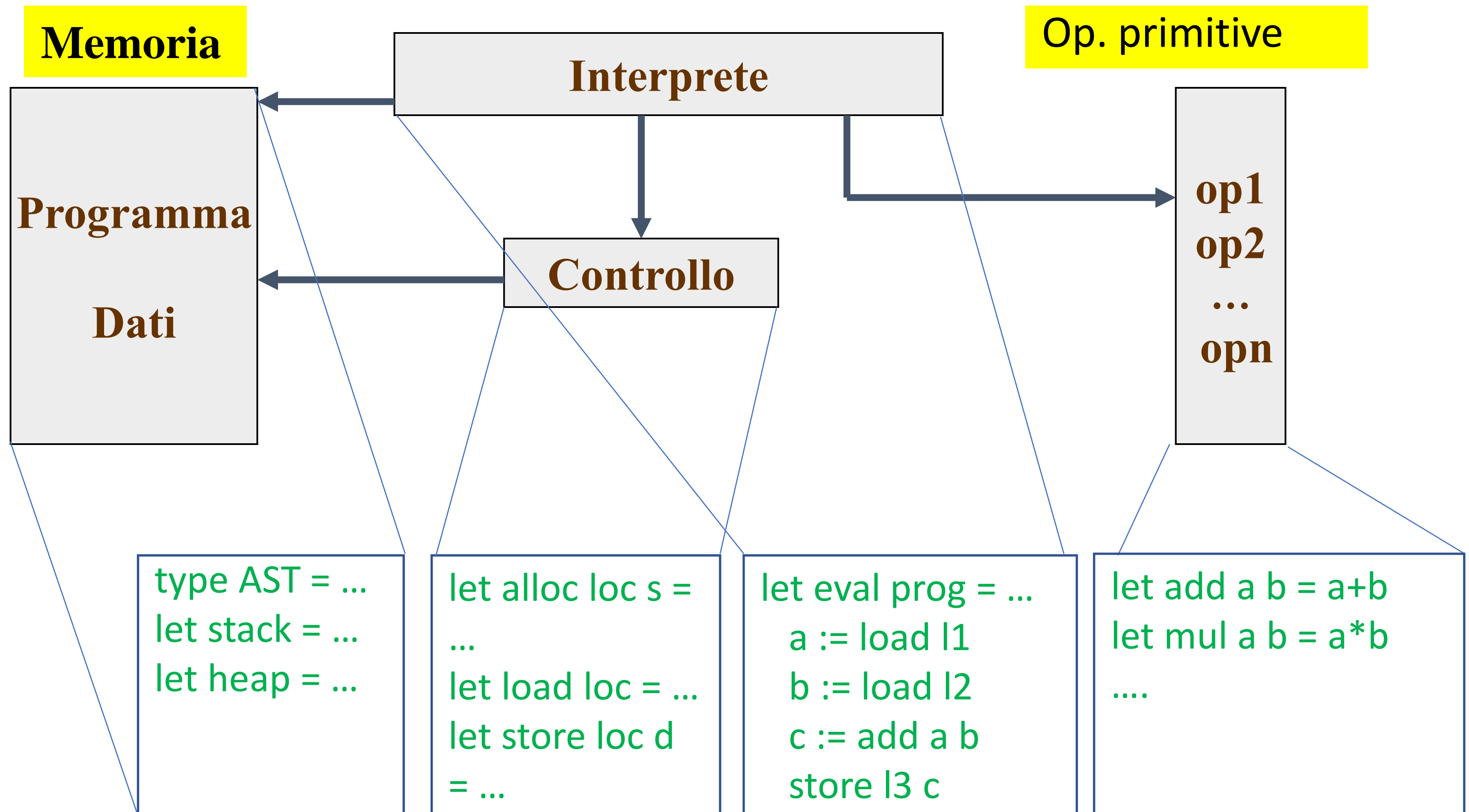
Macchine astratta: implementazioni

In Hardware (CPU + memoria) – macchina fisica



Macchine astratta: implementazioni

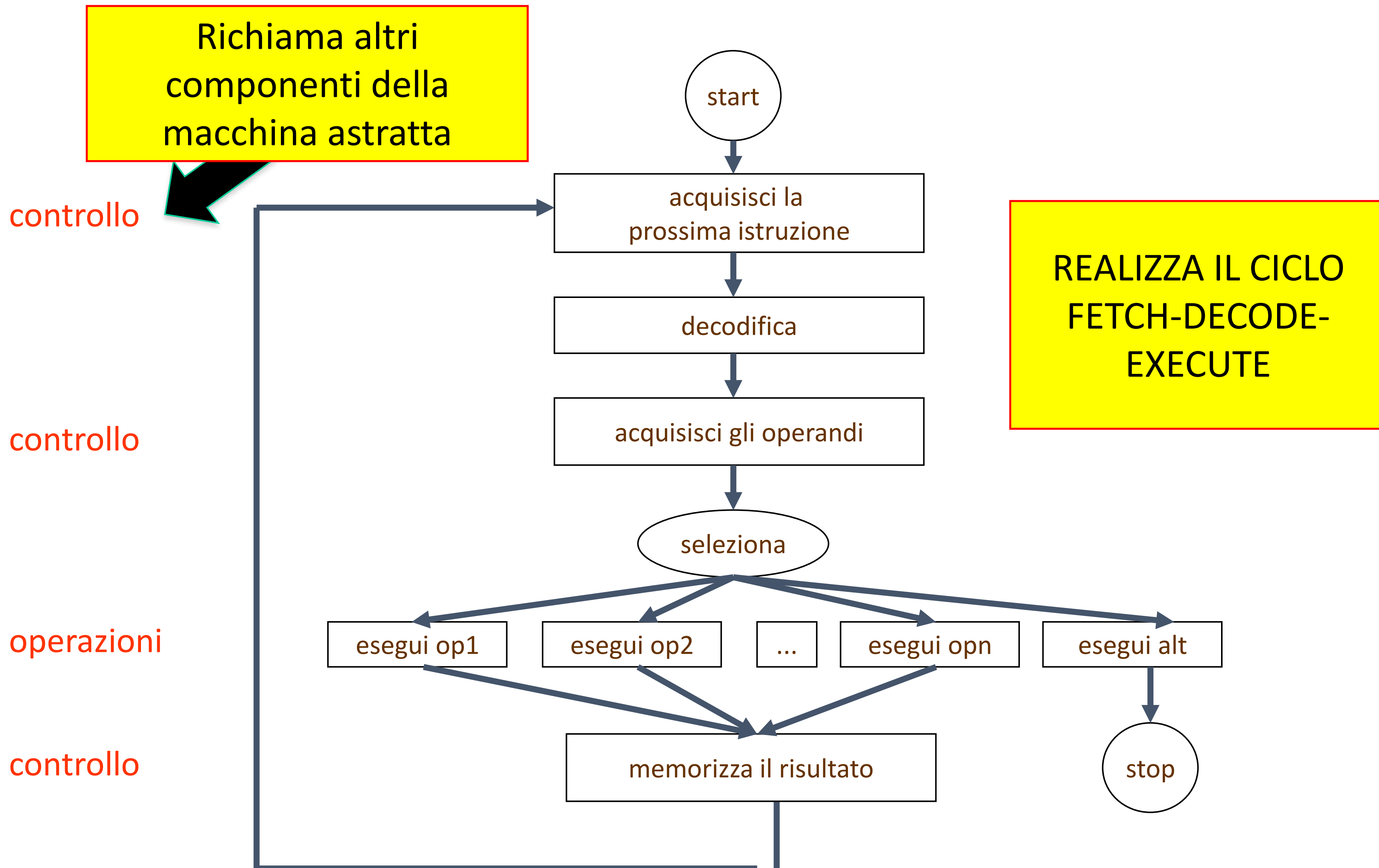
In Software – "interprete" scritto in un certo linguaggio



Interprete (componente di macchina astratta)

- In una implementazione software, l'interprete è il **programma** che prende in ingresso **il programma da eseguire** (tipicamente **l'albero di sintassi astratta del programma)** e lo **esegue ispezionandone la struttura** (l'albero di sintassi) per vedere cosa deve essere fatto.
- Di solito, gli interpreti consentono al programmatore di fornire il programma in formato testuale e ne fanno il **parsing** per generare l'albero di sintassi astratta
- L'esecuzione del programma svolta dall'interprete deve **rispettare la semantica** del linguaggio di programmazione

L’interprete



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
  - gestire i thread
  - mantenere le associazioni fra nomi e valori denotati
  - gestire dinamicamente la memoria
  - …

Operazioni primitive

- Una collezione di strutture dati e algoritmi per **svolgere ogni singola operazione** prevista dal linguaggio
- L'implementazione di un'operazione primitiva può richiedere di effettuare **controlli sui dati** prima di eseguire l'operazione stessa:
  - **controlli (dinamici) di tipo**: ad esempio, l'implementazione di add può dover innanzitutto controllare che i suoi operandi siano numerici
  - **controlli (dinamici) sull'accesso alla memoria**: ad esempio, la lettura di un elemento di una array può richiedere di controllare che non si stia accedendo fuori dai limiti dell'array stesso

Esempio: assegnamento su macchina a registri

$X = Y + 5$

//Controllo trasferimento dati

load Y, R0

// load the value in Y-location into register R0

load # 5, R1

// load constant 5 into register R1

//Esecuzione operazione primitiva

add R0, R1, R0

//add value of R0 and R1, and store the result in R0

//Controllo trasferimento dati

store R0, X

//store value-in(R0) into location of variable X

Macchine astratte e linguaggio macchina

- **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**
- Alle componenti di **M** corrispondono componenti di $\mathbf{L_M}$
  - tipi di dato primitivi
  - costrutti di controllo
    - ✓ per controllare l'ordine di esecuzione
    - ✓ per controllare acquisizione e trasferimento dati

Da linguaggio a macchina astratta

- **M** macchina astratta $\mathbf{L_M}$ linguaggio macchina di **M**
- **L** linguaggio $\mathbf{M_L}$ macchina astratta di **L**
- Implementazione di **L** =

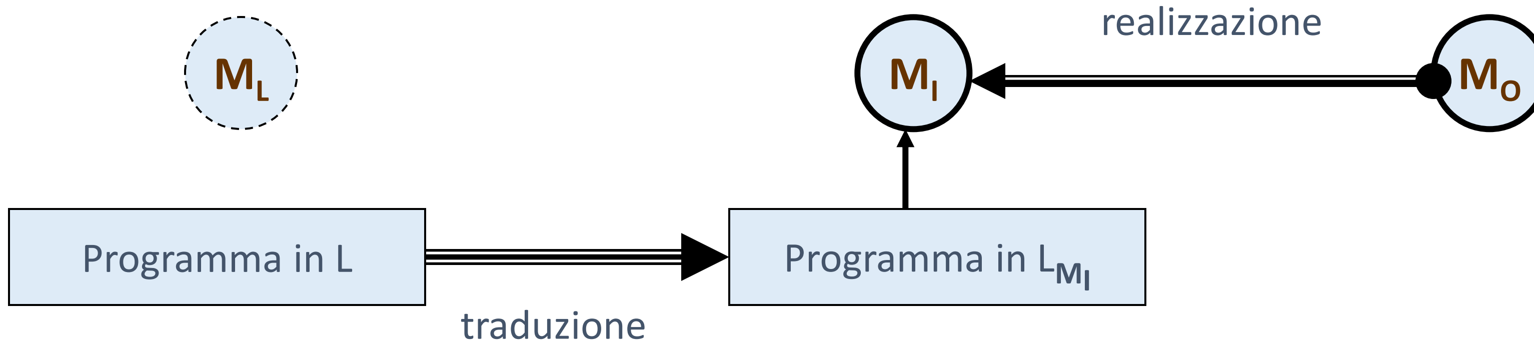
  realizzazione di $\mathbf{M_L}$ su una macchina ospite[1] $\mathbf{M_O}$

[1]Il termine "ospite" qui è usato come traduzione di "host", ossia "che ospita"

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

La macchina intermedia



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

Il Run-Time Support nei linguaggi compilati

- Molto comunemente, anche i linguaggi compilati (ad es. C) generano codice per una macchina intermedia leggermente diversa rispetto alla macchina ospite (più estesa)
- Che differenza esiste tra $\mathbf{M_I}$ e $\mathbf{M_O}$ in questi casi?

Supporto a tempo di esecuzione (run time support - RTS):

collezione di strutture dati e sottoprogrammi che devono essere caricati su $\mathbf{M_O}$ (estensione) per permettere l'esecuzione del codice prodotto dal compilatore

  - $\mathbf{M_I} = \mathbf{M_O} + RTS$

- Il linguaggio $L_{\mathbf{M_I}}$ è il linguaggio macchina di $\mathbf{M_O}$ esteso con chiamate al supporto a tempo di esecuzione

A cosa serve il RTS?

- In linea di principio è possibile tradurre totalmente un programma C in un linguaggio macchina puro, senza chiamate al RTS, ma... la traduzione di alcune primitive (per esempio, relative all'input/output) produrrebbe centinaia di istruzioni in linguaggio macchina che sono sempre le stesse.
  - se le inserissimo nel codice compilato, la sua dimensione crescerebbe a dismisura
  - in alternativa, possiamo inserire nel codice una chiamata a una routine (indipendente dal particolare programma)
  - tale routine deve essere caricata su $\mathbf{M_O}$ ed entra a far parte del rts
- Più il linguaggio è di alto livello, più questa situazione si presenta per quasi tutti i costrutti, per realizzare:
  - meccanismi di controllo dinamico dei tipi
  - gestione efficiente della memoria
  - :

Il run-time support del C

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

Implementazioni miste

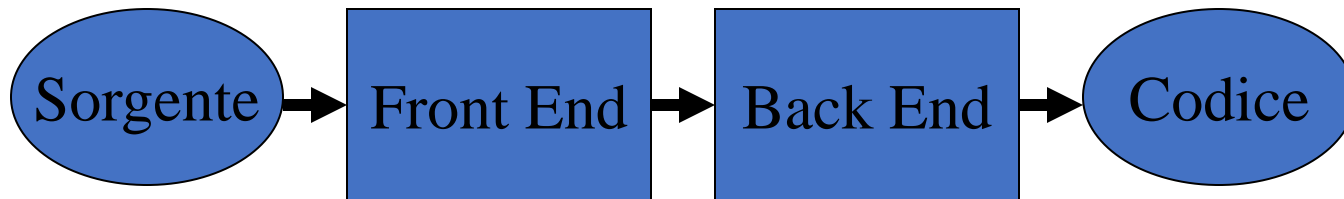
- Anche quando l’interprete della macchina intermedia $\mathbf{M_I}$ è diverso da quello della macchina ospite $\mathbf{M_o}$ è commune prevedere un RTS
  - implementato all'interno dell'interprete

L’implementazione di Java

- È una implementazione mista
  - traduzione dei programmi da Java a byte-code, linguaggio macchina di una macchina intermedia chiamata **Java Virtual Machine**
  - l’interprete della Java Virtual Machine opera su strutture dati (stack, heap) simili a quelle del RTS del compilatore C
  - una differenza fondamentale è la presenza di una gestione automatica del recupero della memoria a heap (**garbage collector**)

Come è fatto un compilatore?

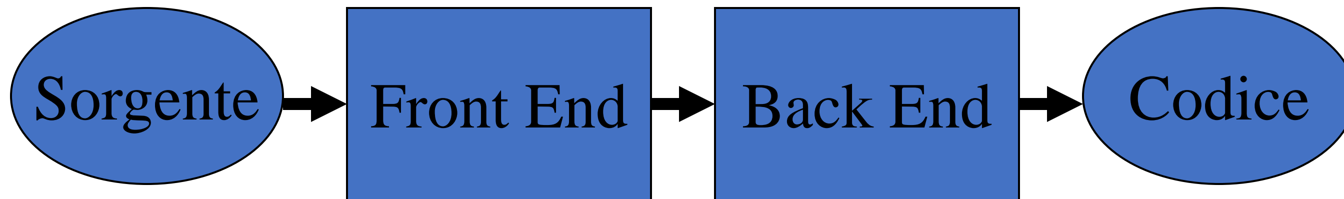
Compilatore



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

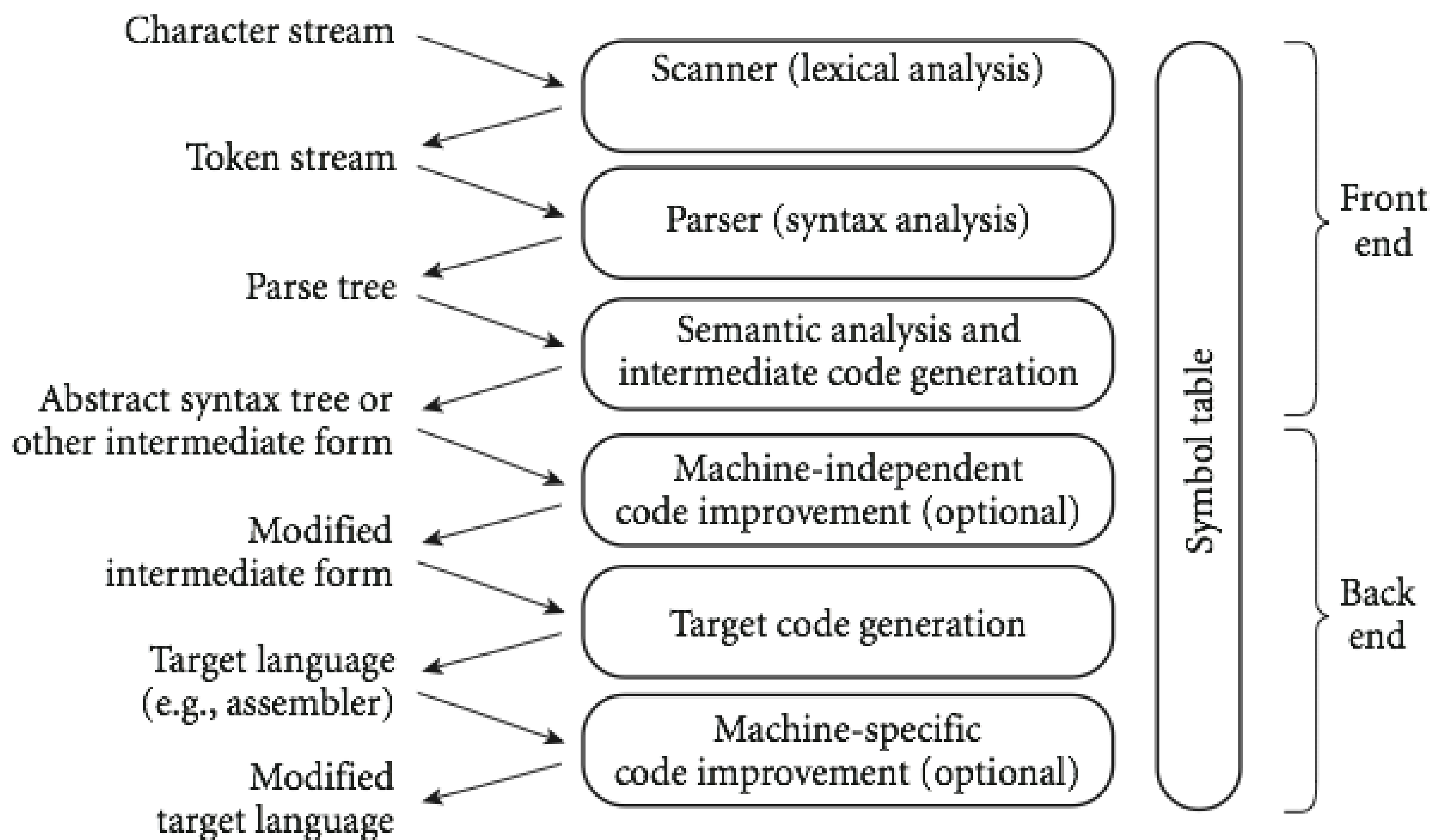


Aspetti critici

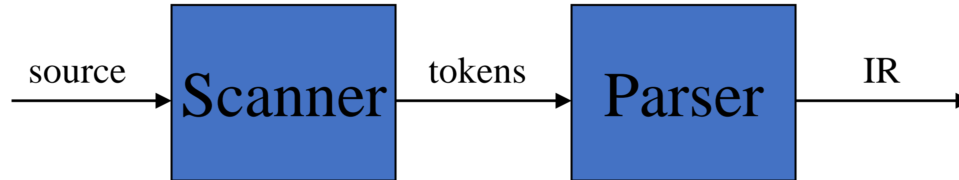
Riconoscere i programmi legali (sintatticamente corretti)

Genererare codice compatibile con il SO della macchina ospite

Struttura di un compilatore



Front End



- Le prime due fasi:
  - scanner: trasforma il programma sorgente nel lessico (token)
  - parser: legge i token e genera il codice intermedio (intermediate representation - IR)
- La teoria aiuta
  - la teoria dei linguaggi formali: automi, grammatiche
  - strumenti automatici per generare scanner e parser

Token

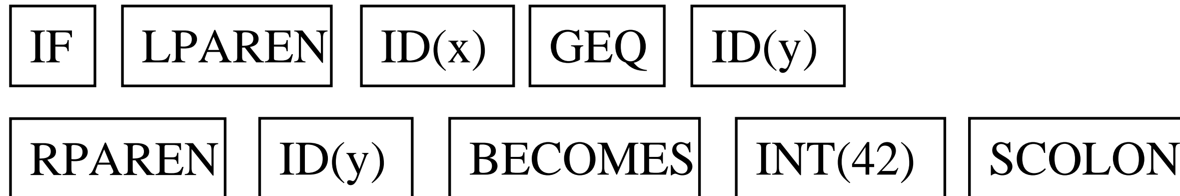
- Token: la costituente lessicale del linguaggio
  - operatori & punteggiatura: {}[]!+-=*;: …
  - parole chiave: if, while, return, class, …
  - identificatori: x, y, …
  - letterali: 3, 'a', 3.5 , "ciao", …

Scanner: un esempio

- Input (programma da compilare):

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

- Sequenza di token corrispondenti:

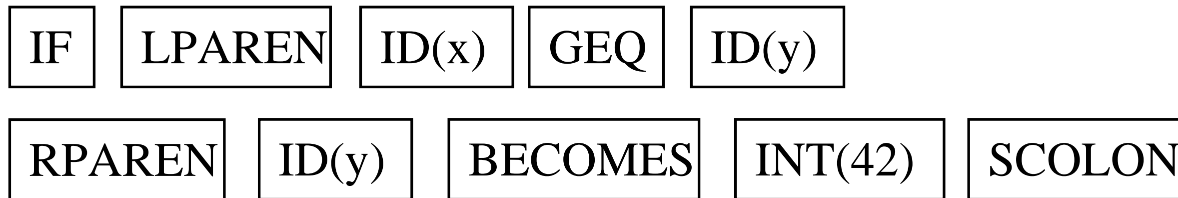


Parser: output (IR)

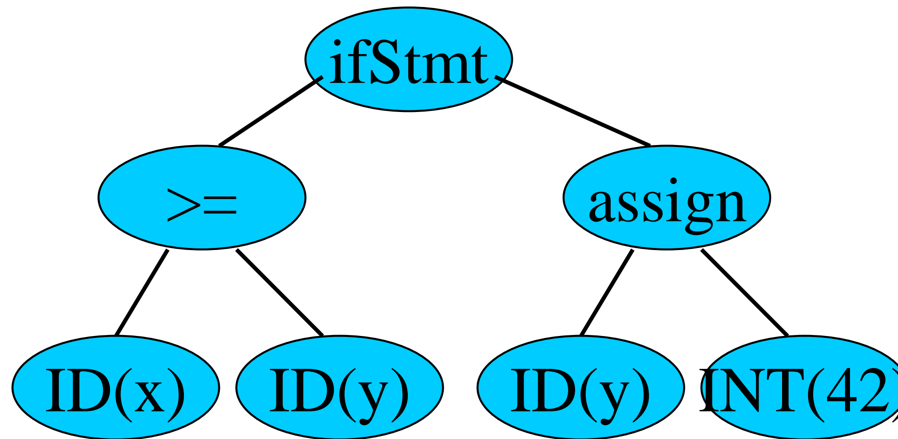
- Formato tipico della rappresentazione intermedia (IR): albero di sintassi astratta (abstract syntax tree)
  - è la struttura sintattica essenziale del programma senza gli aspetti di zucchero sintattico

Parser: AST

Dalla sequenza di token...



all'Abstract Syntax Tree (AST):



Analisi semantica (statica)

- Tipicamente dopo la fase di parsing:
  - type checking
  - varie analisi statiche
  - ottimizzazione del codice

Type checking (statico)

- I controlli di type checking svolti dal compilatore prevengono errori a tempo di esecuzione legati all'esecuzione delle operazioni

  esempio: 3 – "ciao"

- Il type checker controlla che tutte le operazioni siano fatte su operandi del tipo corretto

Nei linguaggi che non prevedono una fase di compilazione:

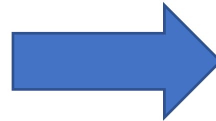
- questi controlli vengono fatti a tempo di esecuzione dal run-time support implementato nell'interprete
- Problema: come fa l'interprete a conoscere il tipo dei dati su cui sta lavorando?

Type checking statico vs dinamico

- Esempio: in C
- Type checking statico. Se il programma supera i controlli sui tipi, a tempo di esecuzione non ci saranno errori nelle operazioni

```
...
int x=3;
int y=4;
int z=x+y;
...
```

compilazione



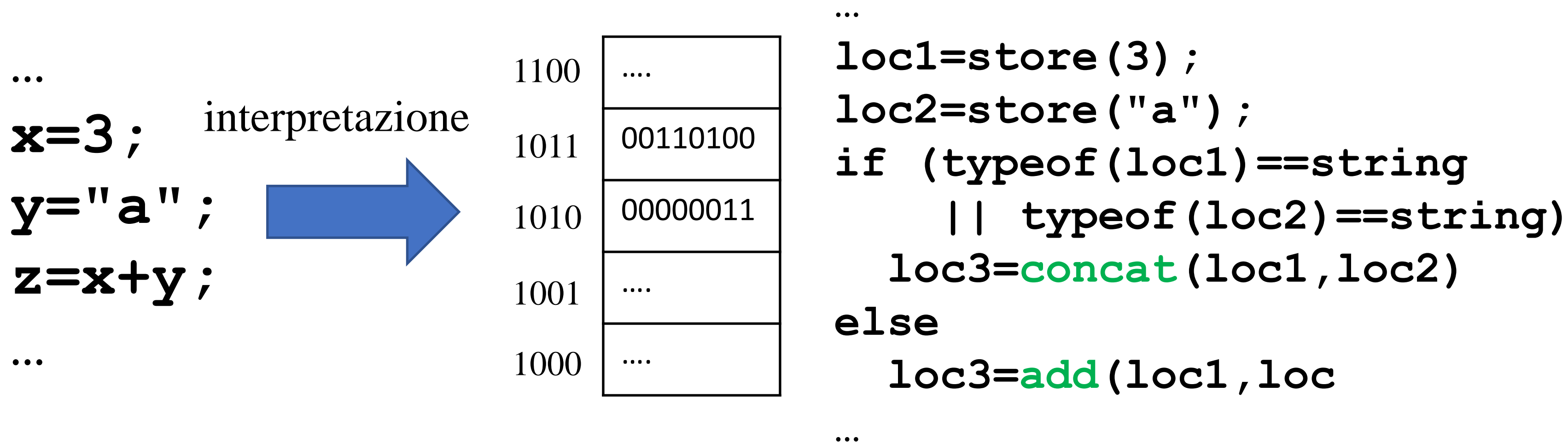
1100	....
1011	00000100
1010	00000011
1001	....
1000	....

```
...
load 1010
load 1011
sum
...
```

I dati nelle locazioni 1010 e 1011 sono per certo rappresentazioni di numeri interi (controllato a tempo di compilazione)

Type checking statico vs dinamico

- Esempio: in JavaScript
- Type checking dinamico. Se il programma supera i controlli sui tipi, a tempo di esecuzione non ci saranno errori nelle operazioni



l'interprete deve decidere se fare concatenazione o somma a seconda del tipo dei dati in memori

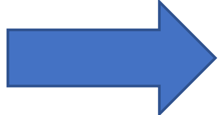
Type checking statico vs dinamico

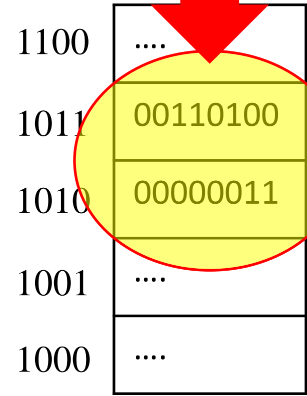
PROBLEMA: COME FA L'INTERPRETE A SAPERE DI CHE TIPO SONO I DATI? IN MEMORA C'E' SOLO LA LORO RAPPRESENTAZIONE BINARIA!
IN GENERALE, DATI DI TIPO DIVERSO POSSONO AVERE LA STESSA RAPPRESENTAZIONE (SEQUENZA DI BIT)...

- Esempi
- Type ch ... ui tipi, a tempo ...

```
...
x=3;
y="a";
z=x+y;
...
```

interpretazione






```
...
loc1=store(3);
loc2=store("a");
if (typeof(loc1)==string
     || typeof(loc2)==string)
  loc3=concat(loc1,loc2)
else
  loc3=add(loc1,loc
...
```

l'interprete deve decidere se fare concatenazione o somma a seconda del tipo dei dati in memori

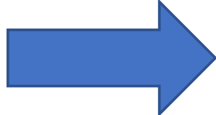
Type checking statico vs dinamico

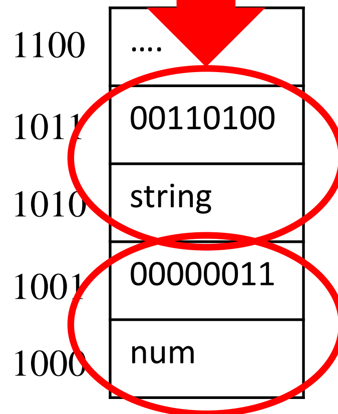
• Esen

• Type ... bi, a temp

SOLUZIONE: DESCRITTORI DEI DATI
OGNI DATO IN MEMORI HA ASSOCIATO UN DESCRITTORE CHE NE FORNISCE IL TIPO (ED EVENTUALMENTE ALTRE INFORMAZIONI UTILI A RUNTIME)

```
...
x=3;
y="a";
z=x+y;
...
```






```
...
loc1=store(3);
loc2=store("a");
if (typeof(loc1)==string
     || typeof(loc2)==string)
   loc3=concat(loc1,loc2)
else
   loc3=add(loc1,loc
...
```

l'interprete deve decidere se fare concatenazione o somma a seconda del tipo dei dati in memori

I descrittori di dato

Nei linguaggi con controllo di tipi dinamico, ogni dato è associato a un descrittore di dato che fornisce le informazioni necessarie per il controllo a tempo di esecuzione

Questo implica:

- Consumo di memoria
- Tempo necessario per effettuare i controlli

I linguaggi compilati, grazie ai controlli statici, sono più efficienti sotto questo punto di vista

Analisi statica: esempi
(controlli sull'inizializzazione delle variabili)

control flow analysis

```
if (b) { c = 5; } else { c = 6; }
```
initialises c

```
if (b) { c = 5; } else { d = 6; }
```
does not

data flow analysis

```
d = 5;  c = d;
```
initialises c

```
c = d;  d = 5;
```
does not

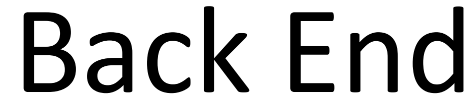
metodi di control-flow e data-flow analysis vengono utilizzati dal compilatore, ad esempio, per controllare che tutte le variabili vengano inizializzate

Analisi statica: esempi (identificazione del codice inutile/inutilizzato)

```
...
if (...) {
      int x=3;
      f(x);
      x=0;
}
...
```

ESEMPIO: ANALIZZANDO STATICAMENTE (SENZA ESEGUIRE) IL CODICE, IL COMPILATORE SI PUO' RENDERE CONTO CHE QUESTO ASSEGNAMENTO PUO' ESSERE RIMOSSO OTTENENDO QUALCOSA DI SEMANTICAMENTE EQUIVALENTE (PERCHE' x E' LOCALE)

Back End



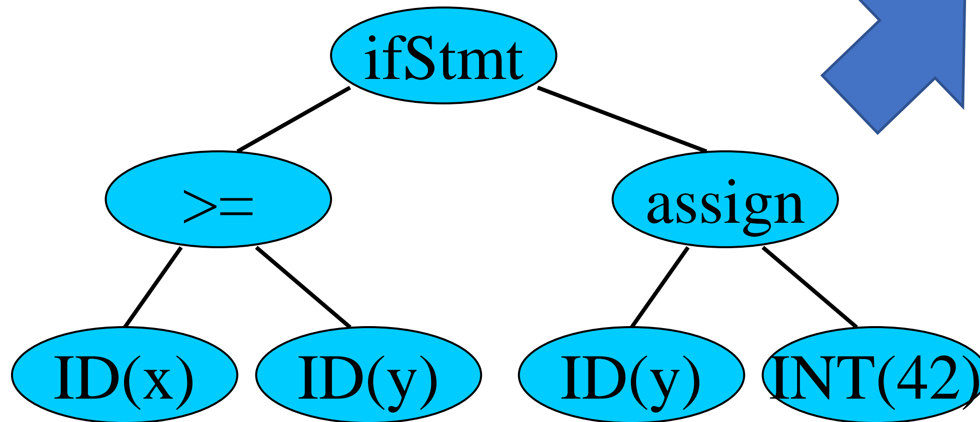
- Cosa fa?
  - traduce il codice nel linguaggio della macchina ospite o della macchina intermedia
    - **Necessaria una conoscenza profonda della macchina ospite o della macchina intermedia (modelo semantico)**

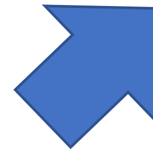
Il risultato complessivo

- Input

  if (x >= y)

  y = 42;





- Output

```
  mov   eax,[ebp+16]
  cmp   eax,[ebp-8]
  jl    L17
  mov   [ebp-8],42
L17:
```

Altri aspetti: Just In Time compiler

- *Idea*: compilare il codice intermedio nel codice nativo durante l'esecuzione
- *Vantaggi*
  - programma continua a essere portatile
  - esecuzioni "ottimizzate" (code inlining)
- *Svantaggi*
  - RTS molto complesso (ottimizza long-running activation)
  - costo della compilazioni JIT