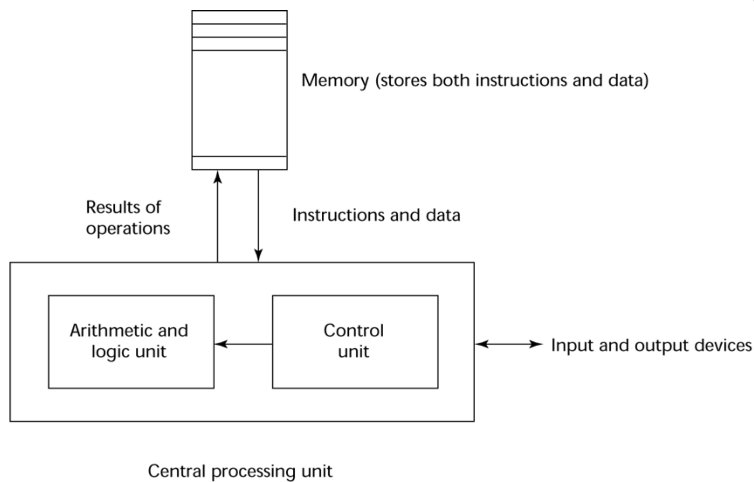
# MACCHINE ASTRATTE, LINGUAGGI, INTERPRETAZIONE, COMPILAZIONE

1

# von Neumann Architecture




1-2

# von Neumann Architecture

- Fetch-execute-cycle

```
initialize the program counter
repeat forever
  fetch the instruction pointed by the counter
  increment the counter
  decode the instruction
  execute the instruction
end repeat
```

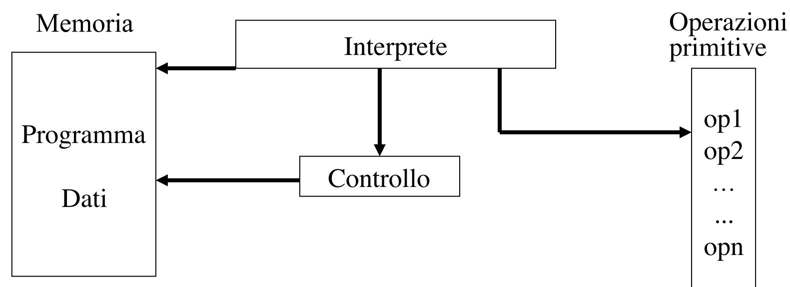


# Implementazione PL

- Compilazione:
  - Programmi sono tradotti nel linguaggio macchina
- Interpretazione:
  - Programmi sono interpretati da un altro programma detto interprete
- Mista:
  - Compilazione + Interpretazione

# MACCHINE ASTRATTE

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

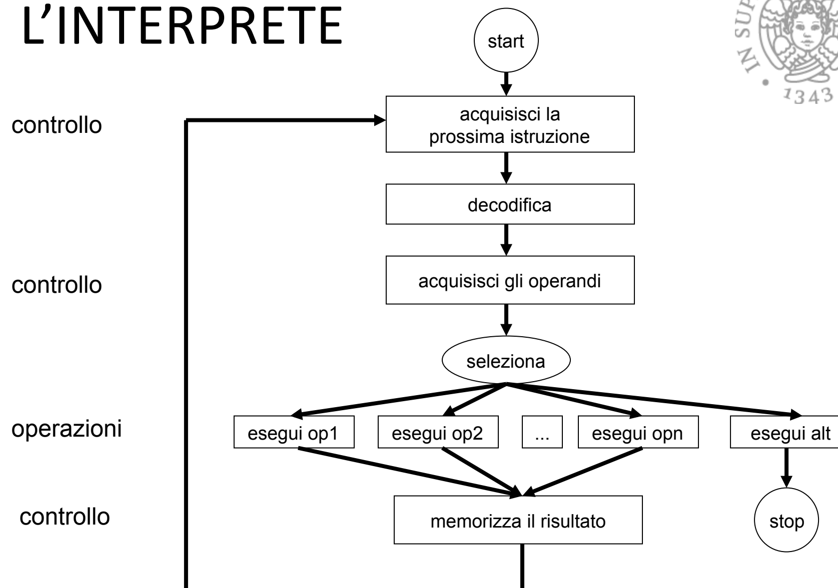



5

# IL COMPONENTE DI CONTROLLO

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

6




# IL 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**
- i programmi sono particolari dati su cui opera l'interprete
- ai 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

8

# MACCHINE ASTRATTE: IMPLEMENTAZIONE

- **M** macchina astratta
- i componenti di **M** sono realizzati mediante strutture dati ed 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

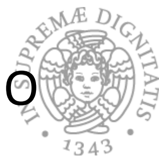
9

# DAL LINGUAGGIO ALLA MACCHINA ASTRATTA

- **M** macchina astratta — $L_M$ linguaggio macchina di **M**
- **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 ed $M_O$ è 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**?)

10

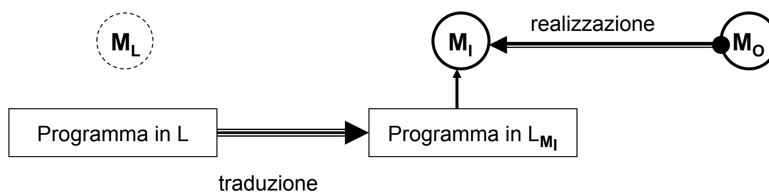
# IMPLEMENTARE UN LINGUAGGIO



- **L** linguaggio ad alto livello
- $\mathbf{M_L}$ macchina astratta di **L**
- $\mathbf{M_O}$ macchina ospite
- **interprete** (puro)
  - $\mathbf{M_L}$ è realizzata su $\mathbf{M_O}$ 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 $\mathbf{M_O}$
  - i programmi tradotti sono eseguiti direttamente su $\mathbf{M_O}$
    - ✓ $\mathbf{M_L}$ non viene realizzata
  - il problema è quello della dimensione del codice prodotto
- due casi limite che nella realtà non esistono quasi mai

11

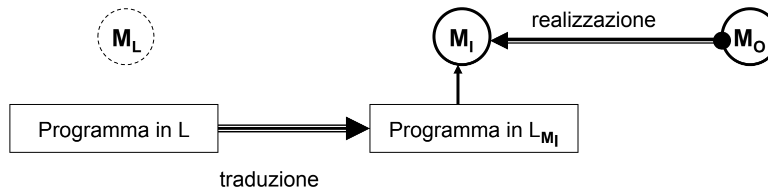
# LA MACCHINA INTERMEDIA




- **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 $\mathbf{M_O}$

12

# INTERPRETAZIONE E TRADUZIONE PURA




- $M_L = M_I$ interpretazione pura
- $M_O = M_I$ traduzione pura
  - 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

13

# IL COMPILATORE

- quando l'interprete della macchina intermedia $M_I$ coincide con quello della macchina ospite $M_O$
- che differenza c'è tra $M_I$ e $M_O$?
  - il **supporto a tempo di esecuzione (rts)**
    - ✓ collezione di strutture dati e sottoprogrammi 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 $L_{M_I}$ è il linguaggio macchina di $M_O$ esteso con chiamate al supporto a tempo di esecuzione

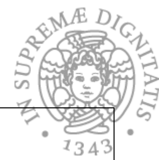
14

# A CHE SERVE IL SUPPORTO A TEMPO DI ESECUZIONE?

- un esempio da un linguaggio antico (FORTRAN): in linea di principio, è possibile tradurre completamente un programma FORTRAN in un linguaggio macchina puro, senza chiamate al rts, ma ...
  - la traduzione di alcune primitive FORTRAN (per esempio, relative all'ingresso uscita) produrrebbe centinaia di istruzioni in linguaggio macchina
    - ✓se le inserissimo nel codice compilato, la sua dimensione crescerebbe a dismisura
    - ✓in alternativa, possiamo inserire nel codice una chiamata ad una routine (indipendente dal particolare programma)
    - ✓tale routine deve essere caricata su $\mathbf{M_O}$ ed entra a far parte del rts
- nei veri linguaggi ad alto livello, questa situazione si presenta per quasi tutti i costrutti del linguaggio
  - meccanismi di controllo
  - non solo routines ma anche strutture dati

15

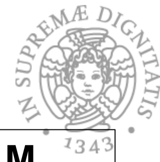
# IL COMPILATORE C




- il supporto a tempo di esecuzione contiene
  - varie strutture dati
    - ✓la pila dei records di attivazione
      - –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

16

# IMPLEMENTAZIONI MISTE




- quando l'interprete della macchina intermedia $\mathbf{M_I}$ non coincide con quello della macchina ospite $\mathbf{M_O}$
- esiste un ciclo di interpretazione del linguaggio intermedio $L_{\mathbf{M_I}}$ realizzato su $\mathbf{M_O}$
  - per ottenere un codice tradotto più compatto
  - per facilitare la portabilità su diverse macchine ospiti
  - si deve riimplementare l'interprete del linguaggio intermedio
  - non è necessario riimplementare il traduttore

17

# 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 routines utilizzate dall'interprete del linguaggio intermedio

18

# L'IMPLEMENTAZIONE DI JAVA

- è un'implementazione mista
  - traduzione dei programmi da Java a byte-code, linguaggio macchina di una macchina intermedia chiamata Java Virtual Machine
  - i programmi byte-code 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 byte-code che l'interprete di Java
    - ✓byte-code è più "vicino" al tipico linguaggio macchina

19

# TRE FAMIGLIE DI IMPLEMENTAZIONI

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

20

## IMPLEMENTAZIONI MISTE E INTERPRETI PURI

- 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"
  - ….

21

## ANALISI STATICA

- dipende dalla semantica del linguaggio
- certi linguaggi (LISP) non permettono praticamente nessun tipo di analisi statica
  - a causa della regola di scoping dinamico nella gestione dell'ambiente non locale
- altri 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

22

## ANALISI STATICA IN JAVA

- Java è fortemente tipato
  - il type checking può essere in gran parte effettuato dal traduttore e sparire quindi dal byte-code generato
- le relazioni di subtyping 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 supertype di quello vero
- di conseguenza, alcune questioni legate ai tipi possono solo essere risolte a tempo di esecuzione
  - scelta del più specifico fra diversi metodi overloaded
  - casting (tentativo di forzare il tipo apparente ad un suo possibile sottotipo)
  - dispatching dei metodi (scelta del metodo secondo il tipo vero)
- controlli e simulazioni a tempo di esecuzione

23

## SEMANTICA FORMALE E SUPPORTO A RUN TIME

- come già anticipato, questo corso si interessa di linguaggi, concentrandosi su due aspetti
  - semantica formale
    - ✓ sempre in forma eseguibile, implementazione ad altissimo livello
  - implementazioni o macchine astratte
    - ✓ interpreti e supporto a tempo di esecuzione
- 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
- perché le macchine astratte?
    - ✓ il progettista deve tener conto delle caratteristche possibili dell'implementazione
    - ✓ l'implementatore la realizza
    - ✓ il programmatore la deve conoscere per utilizzare al meglio il linguaggio

24

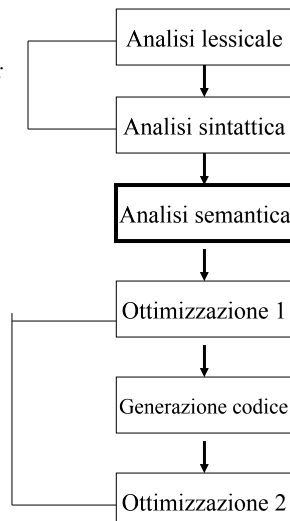
# E IL COMPILATORE?

- 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

25

# STRUTTURA DI UN COMPILATOR

*non ci interessano: aspetti sintattici*




Supporto a run time

26

# JIT Compiler

- Programmi sono tradotti in un linguaggio intermedio
- I sottoprogrammi del linguaggio intermedio vengono compilati nel linguaggio macchina al momento della chiamata
- Il codice complilato viene memorizzato per essere utilizzato successivamente
- JIT Compilers: Java
- .NET utilizza JIT compiler

27

# UNA PICCOLA VARIAZIONE SUL TEMA

28

# Segnature

- Una segnatura many-sorted e’ una coppia (S; O)
- S e’ un insieme non vuoto di elementi (sort)
- O e’ un insieme di elementi (operazioni)
  - f : s1 … sn -> s;
  - S1, …, sn, s sono sort (elementi di S).
- Costanti sono operazioni della forma c : -> s (semplicemente c:s)
- Per ogni sort deve essere presente una costante.

29

# Algebra

- Un’algebra many-sorted A per la segnatura (S, O) e’ una struttura tale che
- Per ogni sort s in S esiste un insieme (non vuoto) As.
- Per ogni constante c : s esiste un elemento $c_A$ in As.
- Per ogni operazione f : s1 … sn -> s esiste una funzione $f_A$ : $A_{s1}$ … $A_{sn}$ -> $A_s$

30

# Esempio

- **Segnatura**
- Sorts nat; bool
- Constants
  - 0: nat;
  - T: bool;
  - F: bool
- Operations
  - add: nat nat -> nat
  - leq: nat nat -> bool

- **Algebra**
- **$A_{nat}$ = N, $A_{bool}$ = B,**
- Costanti 0, T, F,
- Operazioni
  - $add_A(n, m) = n + m$,
  - Leq(n, m) =
    if n <=m then T else F

31

# Termini

- Segnatura: (S, O) ,
- X = ($X_s$) una famiglia di insiemi di variabili per ogni sort
  - Ogni variabile di sort s e' un termine di sort s
  - Ogni costante di sort s e' un termine di sort s
  - If f : s1 … sn -> s e' una operazione e t1, …, tn sono termini di sort s1; … sn, allora f(t1 … tn) e' un termine di sort s.
- Un termine e' chiuso se non contiene variabili

32

# Termini

- x
- 0
- add(0; y)
- add(add(0; x); y)
- add(add(0; 0); add(x; x))
- add(0; add(0; add(0; 0)))
- leq(0, add(0))

33

# Algebra dei termini

Un assegnamento di variabili e' una funzione che associa a ogni variabile di sort s un valore di quel sort

$$\alpha(x) \in A_s.$$

(i) $x^{A,\alpha} := \alpha(x)$.

(ii) $c^{A,\alpha} := c^A$.

(iii) $f(t_1, \ldots, t_n)^{A,\alpha} := f^A(t_1^{A,\alpha}, \ldots, t_n^{A,\alpha})$.

34

# Esempio

$\alpha: \{x, y\} \to \mathbf{N}$, $\alpha(x) := 3$ and $\alpha(y) := 5$

$$x^{A,\alpha} = \alpha(x) = 3$$

$$0^{A,\alpha} = 0^A = 0$$

$$\mathrm{add}(0, y)^{A,\alpha} =$$

$$\mathrm{add}(\mathrm{add}(0, x), y)^{A,\alpha} =$$

$$\mathrm{add}(\mathrm{add}(0, 0), \mathrm{add}(x, x))^{A,\alpha} =$$

35

Let $\Sigma = (S, \Omega)$ a signature and $X$ a set of variables for $\Sigma$. We define a $\Sigma$-algebra $\mathrm{T}(\Sigma, X)$, called **term algebra**, as follows.

| | |
|---|---|
| **Algebra** | $\mathrm{T}(\Sigma, X)$ |
| **Carriers** | $\mathrm{T}(\Sigma, X)_s \quad (s \in S)$ |
| **Constants** | $c^{\mathrm{T}(\Sigma,X)} := c$ |
| **Operations** | $f^{\mathrm{T}(\Sigma,X)}(t_1, \ldots, t_n) := f(t_1, \ldots, t_n)$ |

36

## Everything is an algebraic type

```
data Bool = False | True
data Season = Winter | Spring | Summer | Fall
data Shape = Circle Float | Rectangle Float Float
data Exp = Lit Int | Add Exp Exp | Mul Exp Exp
data List a = Nil | Cons a (List a)
data Nat = Zero | Succ Nat
data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)
data Maybe a = Nothing | Just a
data Pair a b = Pair a b
data Sum a b = Left a | Right b
```

37

## Boolean

```
data Bool = False | True

not :: Bool -> Bool
not False  =  True
not True   =  False

(&&) :: Bool -> Bool -> Bool
False && q   =  False
True  && q   =  q

(||) :: Bool -> Bool -> Bool
False || q   =  q
True  || q   =  True
```

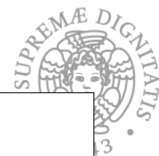
38

## Boolean — eq and show

```
eqBool :: Bool -> Bool -> Bool
eqBool False False  =  True
eqBool False True   =  False
eqBool True  False  =  False
eqBool True  True   =  True

showBool :: Bool -> String
showBool False  =  "False"
showBool True   =  "True"
```

39



## Lists

With declarations

```
data  List a  =  Nil
              |  Cons a (List a)

append :: List a -> List a -> List a
append Nil ys           =  ys
append (Cons x xs) ys   =  Cons x (append xs ys)
```

With built-in notation

```
(++) :: [a] -> [a] -> [a]
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)
```

40

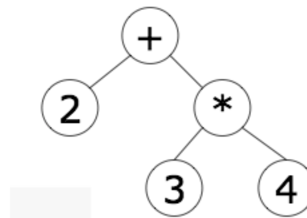
```
# type persona =
  Studente of string
  | Docente of string * int;;
type persona = Studente of string | Docente of string * int
```

```
# let getCode persona =
  match persona with
   Docente(nome, codice) -> codice
  | Studente nome -> failwith "nessun codice";;
val getCode : persona -> int = <fun>
```

41

## Sintassi astratta

- Una espressione algebraica (2 + (3 * 4)) puo' essere rappresentata come un albero
- Come possiamo rappresentarla con i tipi di dato algebrici di Ocaml?
- Una espressione algebraica




42