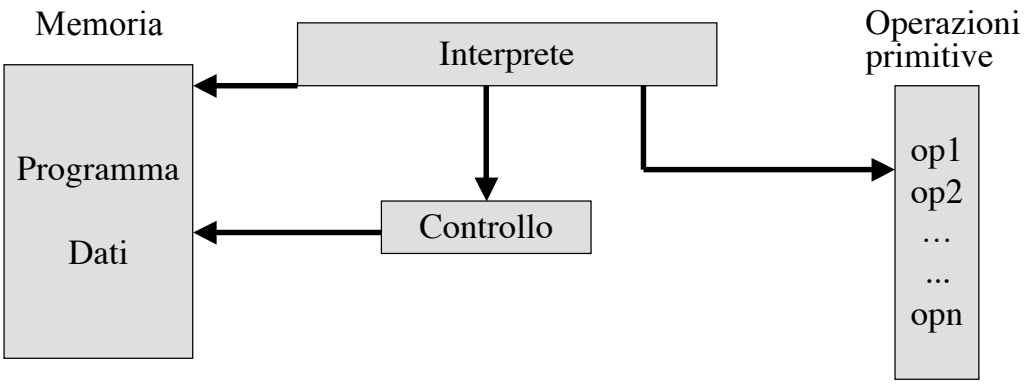
Macchine astratte, linguaggi, interpretazione, compilazione

Macchine astratte

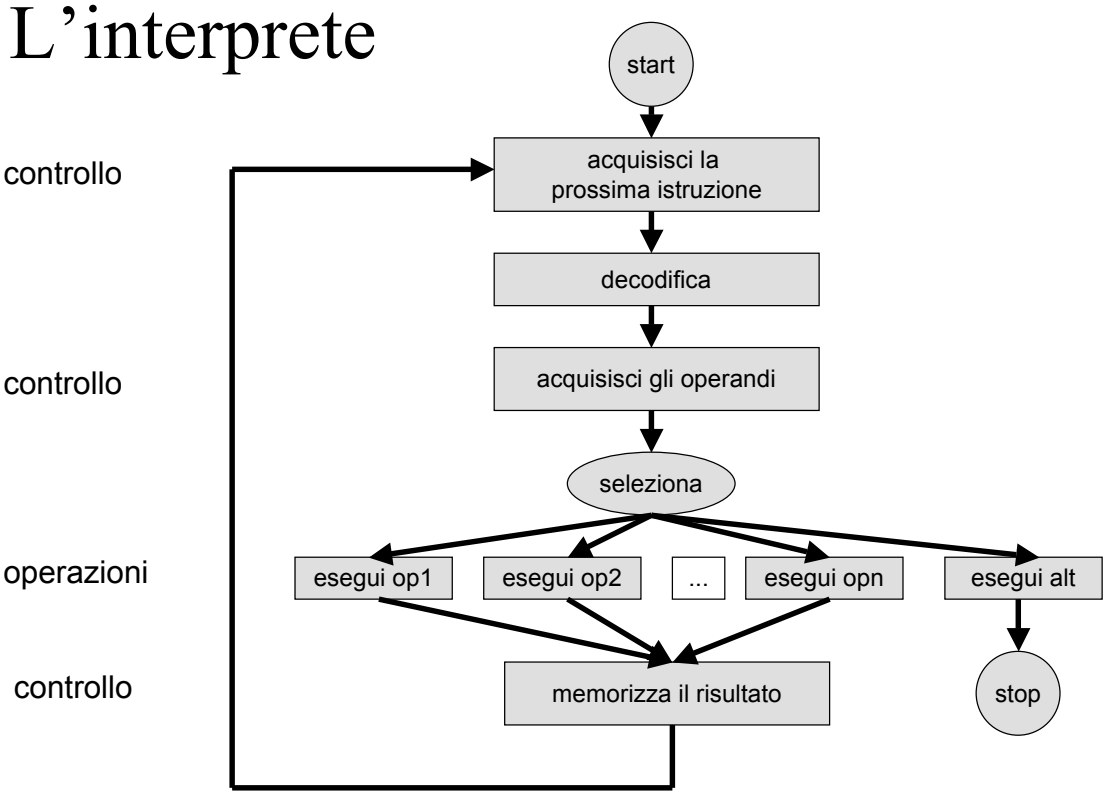
- una collezione di strutture dati ed algoritmi in grado di memorizzare ed eseguire programmi
- componenti della macchina astratta
  - interprete
  - memoria (dati e programmi)
  - controllo
  - operazioni "primitive"



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
  - …...

L'interprete



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

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** $\mathbf{M_O}$, già esistente (implementata)
- è importante la realizzazione dell'interprete di **M**
  - può coincidere con l'interprete di $\mathbf{M_O}$
    - **M** è realizzata come **estensione** di $\mathbf{M_O}$
    - altri componenti della macchina possono essere diversi
  - può essere diverso dall'interprete di $\mathbf{M_O}$
    - **M** è realizzata su $\mathbf{M_O}$ in modo **interpretativo**
    - altri componenti della macchina possono essere uguali

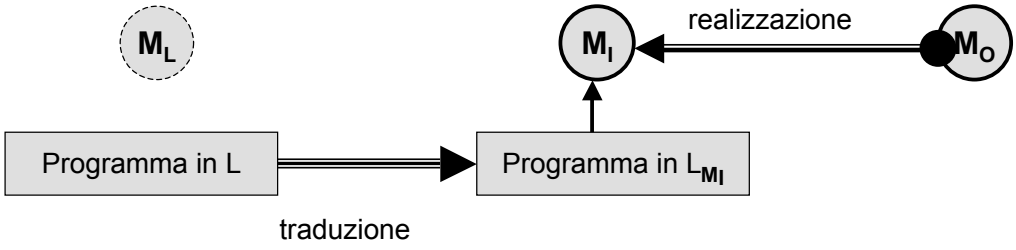
Dal linguaggio alla macchina astratta

- **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 $\mathbf{M_O}$
- se **L** è un linguaggio ad alto livello ed $\mathbf{M_O}$ è una macchina "fisica"
  - l'interprete di $\mathbf{M_L}$ è necessariamente diverso dall'interprete di $\mathbf{M_O}$
    - $\mathbf{M_L}$ è realizzata su $\mathbf{M_O}$ in modo interpretativo
    - l'implementazione di **L** si chiama **interprete**
    - esiste una soluzione alternativa basata su tecniche di traduzione (**compilatore**?)

Implementare un linguaggio

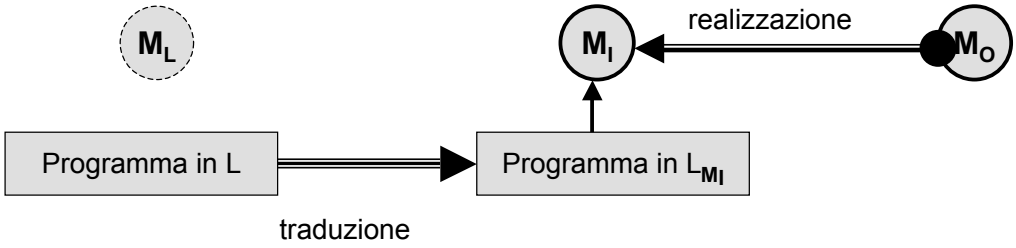
- **L** linguaggio ad alto livello
- $\mathbf{M_L}$ macchina astratta di **L**
- $\mathbf{M_O}$ macchina ospite
- implementazione di **L** 1: **interprete** (puro)
  - $\mathbf{M_L}$ è realizzata su $\mathbf{M_O}$ in modo interpretativo
  - scarsa efficienza, soprattutto per colpa dell'interprete (ciclo di decodifica)
- implementazione di **L** 2: **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

La macchina intermedia



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

Interpretazione e traduzione pura



- $\mathbf{M_L} = \mathbf{M_I}$ interpretazione pura
- $\mathbf{M_O} = \mathbf{M_I}$ traduzione pura
  - possibile solo se la differenza fra $\mathbf{M_O}$ e $\mathbf{M_L}$ è molto limitata
    - L linguaggio assembler di $\mathbf{M_O}$
  - 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 $\mathbf{M_I}$ coincide con quello della macchina ospite $\mathbf{M_O}$
- che differenza c'è tra $\mathbf{M_I}$ e $\mathbf{M_O}$?
  - il **supporto a tempo di esecuzione (rts)**
    - collezione di strutture dati e sottoprogrammi che devono essere caricati su $\mathbf{M_O}$ (estensione) per permettere l'esecuzione del codice prodotto dal traduttore (compilatore)
  - $\mathbf{M_I}$ = $\mathbf{M_O}$ + rts
- il linguaggio $\mathbf{L_{M_I}}$ è il linguaggio macchina di $\mathbf{M_O}$ esteso con chiamate al supporto a tempo di esecuzione

A che serve il supporto a tempo di esecuzione?

- un esempio da un linguaggio antico (FORTRAN)
  - praticamente una notazione "ad alto livello" per un linguaggio macchina
- 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'I/O) 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

Il caso del 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

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_{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

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

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

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)

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"
  - ….

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

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

Quando la macchina ospite è una "macchina ad alto livello"

- fino ad ora non abbiamo preso in considerazione le caratteristiche della macchina ospite $\mathbf{M_O}$ utilizzata nell'implementazione del linguaggio **L**
- il linguaggio $\mathbf{L_{MO}}$ è il linguaggio in cui vengono implementati
  - l'interprete di **L** se l'implementazione è interpretativa
  - l'interprete della macchina intermedia $\mathbf{M_i}$ di **L** se l'implementazione è mista
  - il supporto a tempo di esecuzione di **L** se l'implementazione è compilativa
    - non ha nessuna importanza, rispetto alle considerazioni che vogliamo fare, il linguaggio in cui sono implementati i traduttori da **L** a $\mathbf{L_{Mi}}$ o da **L** a $\mathbf{L_{MO}}$, negli ultimi due casi
- se $\mathbf{L_{MO}}$ è a sua volta un linguaggio ad alto livello, le implementazioni portano ad una stratificazione di macchine astratte che è necessario tener presente, soprattutto quando si voglia ragionare sulle prestazioni dei programmi di **L** in esecuzione
- in questo caso, infatti, l'implementazione di **L** viene eseguita "sopra il supporto a tempo di esecuzione" che realizza $\mathbf{M_O}$

Perché un linguaggio ad alto livello per implementarne un altro

- è più facile e meno costoso implementare qualunque insieme di algoritmi e strutture dati in un linguaggio ad alto livello piuttosto che in linguaggio macchina
  - e questo vale anche per l'implementazione di interpreti e supporti a tempo di esecuzione
- il prezzo che si paga è quello della possibile inefficienza introdotta dalla stratificazione di macchine astratte
  - che può dipendere dalle scelte fatte nell'implementazione
- la questione importante da capire è come l'implementazione tratta quei costrutti di **L**, che hanno un corrispettivo immediato nel linguaggio di implementazione
- discuteremo questo aspetto, prendendo in considerazione una implementazioe interpretativa di **L**, realizzata utilizzando il linguaggio C
  - l'interprete realizzato, una volta compilato dal compilatore C, gira sul supporto a tempo di esecuzione di C
  - con la sua pila, la heap, etc.
- rivedremo concretamente una situazione simile negli interpreti del linguaggio didattico, implementati in Ocaml

Interprete di **L** implementato in C

- supponiamo che **L** abbia alcuni costrutti
  - per esempio, astrazioni procedurali, ricorsione, allocazione dinamica su heap

  simili a quelli di C
- l'interprete di **L** può valutare tali costrutti in diversi modi, i cui casi limite sono
  - l'utilizzazione diretta del corrispondente costrutto di C
    - e quindi di quella parte del supporto a run time di C che lo realizza
  - la simulazione ex-novo del costrutto, con l'introduzione delle strutture dati necessarie
    - che potrebbero essere una replica molto simile di quelle del supporto di C
- è evidente che, con la prima scelta, lo strato aggiunto sopra C è minore
  - anche se l'implementazione risultante non è necessariamente più efficiente
- considerazioni molto simili si applicano alle altre classi di implementazioni
- nelle implementazioni miste in cui la macchina intermedia è definita "a priori" (Java, Prolog) è comunque molto difficile riuscire a riutilizzare le strutture dati del supporto di C

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

E il compilatore?

- la maggior parte dei corsi e dei libri sui linguaggi si occupano di compilatori
- perché noi no?
  - perché c'è un altro corso dedicato a questo argomento
  - perché delle cose tradizionalmente trattate con il punto di vista del compilatore, poche sono quelle che realmente ci interessano
- per capire meglio, guardiamo la struttura di un tipico compilatore

Struttura di un compilatore