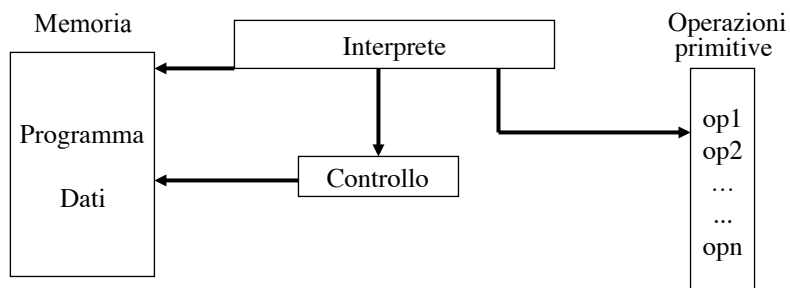


Macchine astratte, linguaggi, interpretazione, compilazione

1

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"



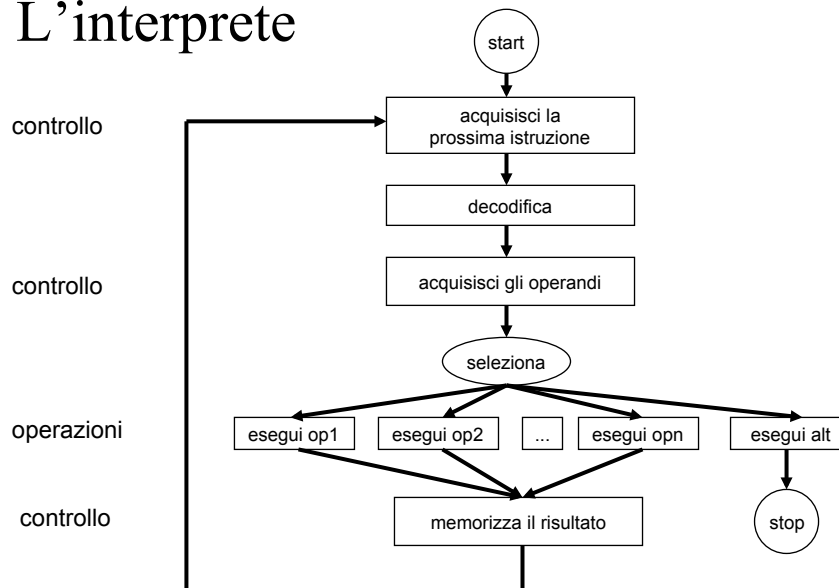
2

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
 -

3

L'interprete



4

Il linguaggio macchina

- **M** macchina astratta
- **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 **L_M**
 - tipi di dato primitivi
 - costrutti di controllo
 - per controllare l'ordine di esecuzione
 - per controllare acquisizione e trasferimento dati

5

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 M₀**, già esistente (implementata)
- è importante la realizzazione dell'interprete di **M**
 - può coincidere con l'interprete di **M₀**
 - **M** è realizzata come **estensione** di **M₀**
 - altri componenti della macchina possono essere diversi
 - può essere diverso dall'interprete di **M₀**
 - **M** è realizzata su **M₀** in modo **interpretativo**
 - altri componenti della macchina possono essere uguali

6

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?**)

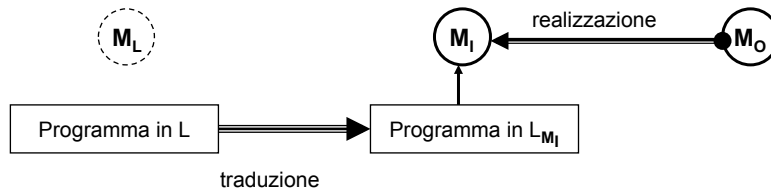
7

Implementare un linguaggio

- ☛ L linguaggio ad alto livello
- ☛ M_L macchina astratta di L
- ☛ M_O macchina ospite
- ☛ implementazione di L 1: **interprete** (puro)
 - M_L è realizzata su 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 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
- ☛ due casi limite che nella realtà non esistono quasi mai

8

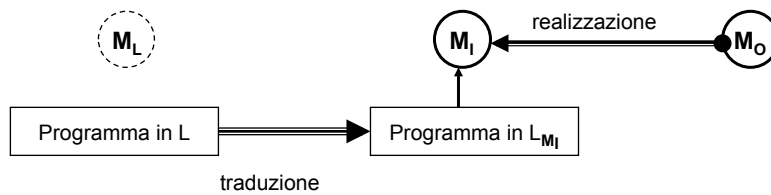
La macchina intermedia



- ☛ 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

9

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

10

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

11

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 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

12

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

13

Implementazioni miste

- ☛ quando l'interprete della macchina intermedia M_I non coincide con quello della macchina ospite M_O
- ☛ esiste un ciclo di interpretazione del linguaggio intermedio L_{M_I} realizzato su 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

14

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

15

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

16

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

- macchina intermedia M_I realizzata per estensione sulla macchina ospite M_O (rts, nessun interprete)
 - C, C++, PASCAL

☛ implementazione mista

- traduzione dei programmi da L a L_{M_I}
- i programmi L_{M_I} sono interpretati su M_O
 - Java
 - i “compilatori” per linguaggi funzionali e logici (LISP, PROLOG, ML)
 - alcune (vecchie!) implementazioni di Pascal (Pcode)

17

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”
-

18

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

19

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

20

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 caratteristiche possibili dell'implementazione
 - l'implementatore la realizza
 - il programmatore la deve conoscere per utilizzare al meglio il linguaggio

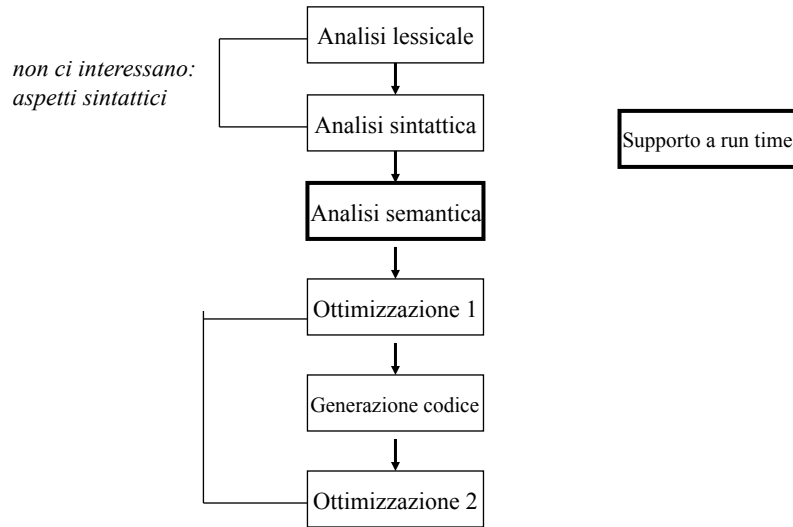
21

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
- ☛ per capire meglio, guardiamo la struttura di un tipico compilatore

22

Struttura di un compilatore



23