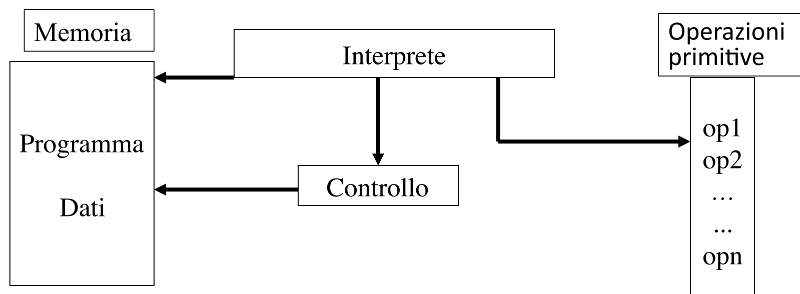
MACCHINE ASTRATTE, LINGUAGGI, INTERPRETAZIONE, COMPILAZIONE

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"



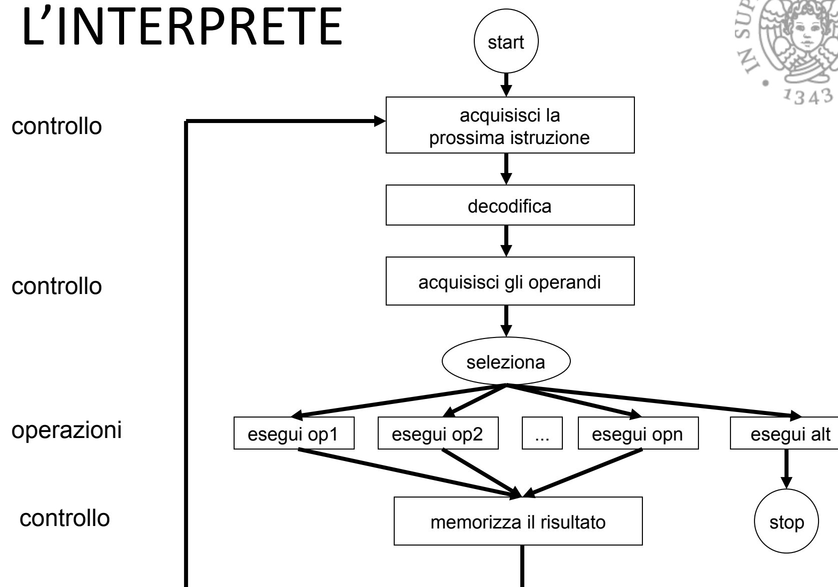
2

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
- $\mathbf{L_M}$ 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 $\mathbf{L_M}$
  - 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** $\mathbf{M_O}$, 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

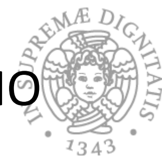
6

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**?)

7

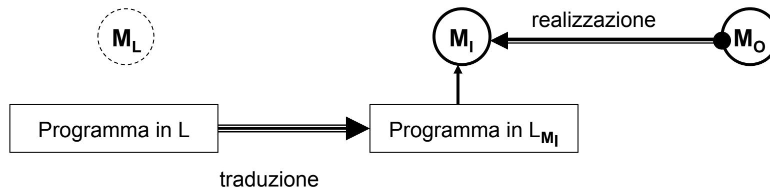
IMPLEMENTARE UN LINGUAGGIO



- **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

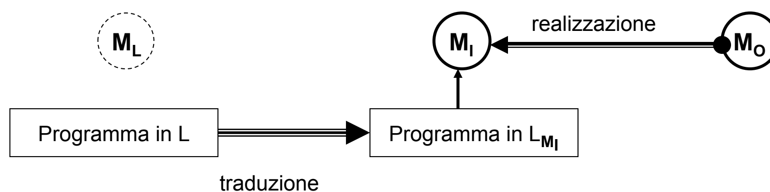
8

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

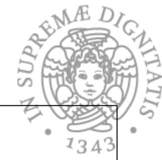
11

A CHE SERVE IL SUPPORTO A TEMPO DI ESECUZIONE?

- un esempio da un linguaggio antico (FORTRAN): 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

12

IL COMPILATORE C



- il supporto a tempo di esecuzione contiene
  - varie strutture dati
    - ✓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

13

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

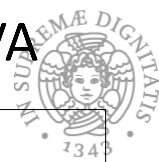
14

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

- Linguaggi di programmazione:
- Due aspetti essenziali della nostra visione (intendo quella del corso)
  - semantica formale
    - ✓ eseguibile, implementazione ad altissimo livello
  - implementazioni o macchine astratte
    - ✓ interpreti e supporto a tempo di esecuzione

21

Perche?

- 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

22

Perche?

- Diventare un programmatore consapevole
  - Migliore comprensione delle caratteristiche dei linguaggi di programmazione
  - Comprendere le tecniche di implementazione
  - Migliore intuizione del comportamento del proprio codice

Perche'?

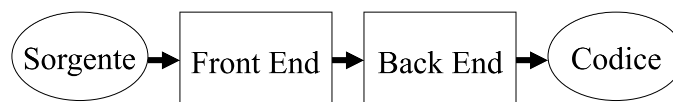
- Miscela affascinante di teoria e pratica
  - Applicazione immediata e diretta della teoria
    - ✓Tecniche di analisi statica: inferenza dei tipi
  - Tecniche algoritmiche (problemi NP-hard)
    - ✓Allocazione delle risorse a run-time

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

Compilatore

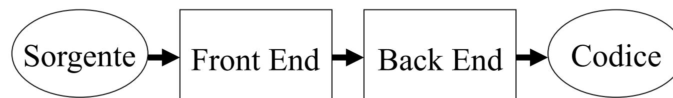


Front end: fasi di analisi:
Legge il programma sorgente e determina la sua struttuta sia sintattica che semantica

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

26

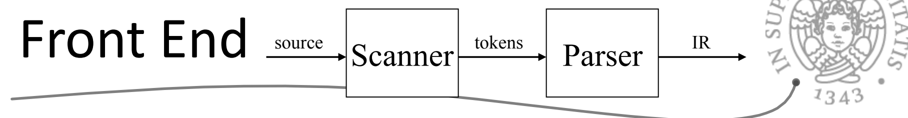
Compilatore



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

27

Front End



- Due fasi principali
  - Scanner: compito di trasformare il programmma sorgente nel lessico (tokens)
  - 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

Token

- Token: La costituente lessicale del linguaggio
  - Operatori & Punteggiatura: {}[]!+-=*;: …
  - Parole chiaves: if while return class …
  - Identificatori: ….
  - Costanti: int, floating-point character, string, …

Scanner: un esempio

- Input

```
// codice stupido
if (x >= y) y = 42;
```

- Token:

IF	LPAREN	ID(x)	GEQ	ID(y)

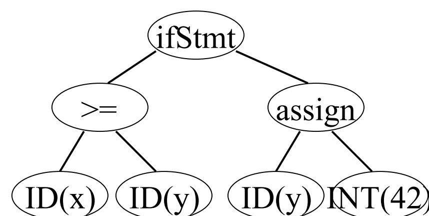
RPAREN	ID(y)	BECOMES	INT(42)	SCOLON

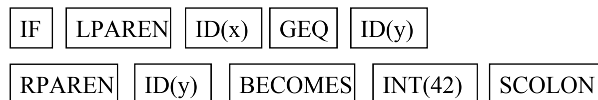
Parser: Output (IR)

- 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 parliamo anche nel seguito

Parser: AST

- Abstract Syntax Tree (AST)





Analisi semantica (statica)

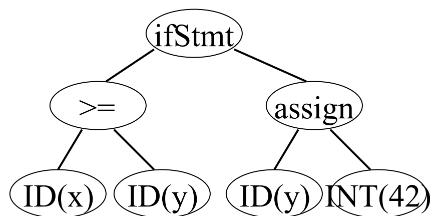
- Tipicamente dopo la fase di parsing
  - Type checking
  - Uso e allocazione delle risorse
  - Ottimizzazione del codice

Back End

- Cosa fa?
  - Traduce il codice intermedio nel linguaggio della macchina ospite
  - Usare le risorse della macchina ospite in modo effettivo

Il risultato complessivo

- Input
  if (x >= y)
    y = 42;



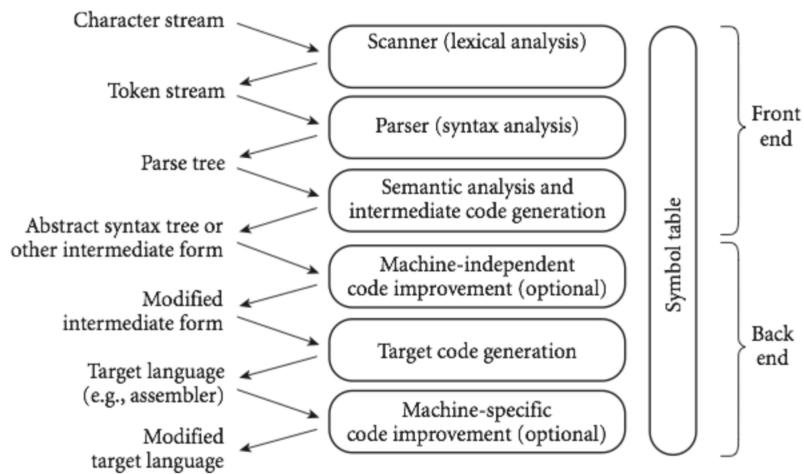
- Output

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

METTIAMO INSIEME LE COSE

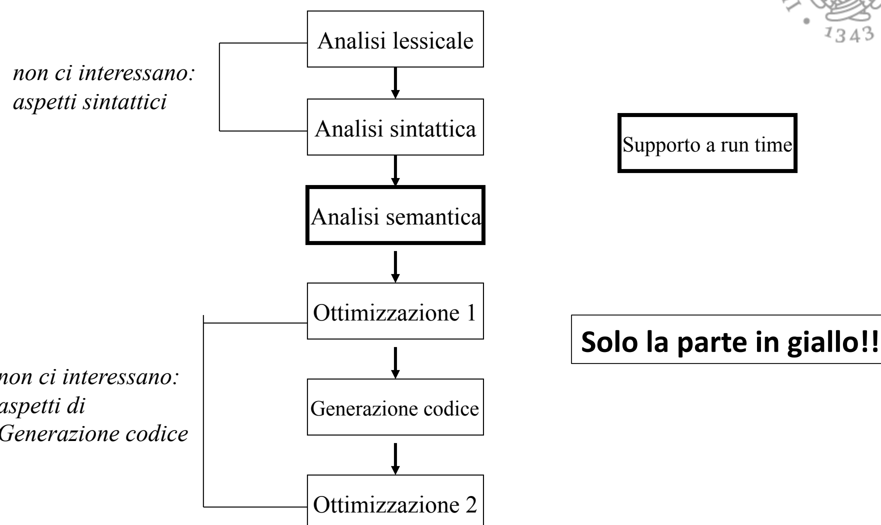
36

Struttura di un compilatore



37

Cosa ci interessa?



Solo la parte in giallo!!

38

JIT Compiler

- *Idea*: compilare il byte code nel codice nativo durante l'esecuzione.
- *Vantaggi*:
  - Programma continua a essere portatile;
  - Esecuzioni "ottimizzate" (code inlining)
- *Svantaggi*:
  - Run-time system molto complicato; (ottimizzare long-running activations)
  - Costo della compilazioni JIT.
- Noi non ne parliamo.

39