



AA 2014-2015

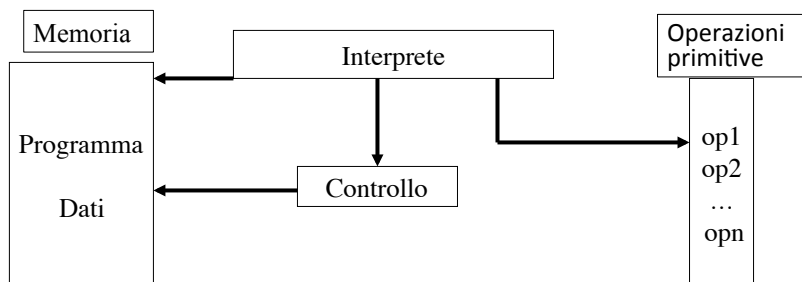
16. Macchine astratte, linguaggi, interpretazione, compilazione

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

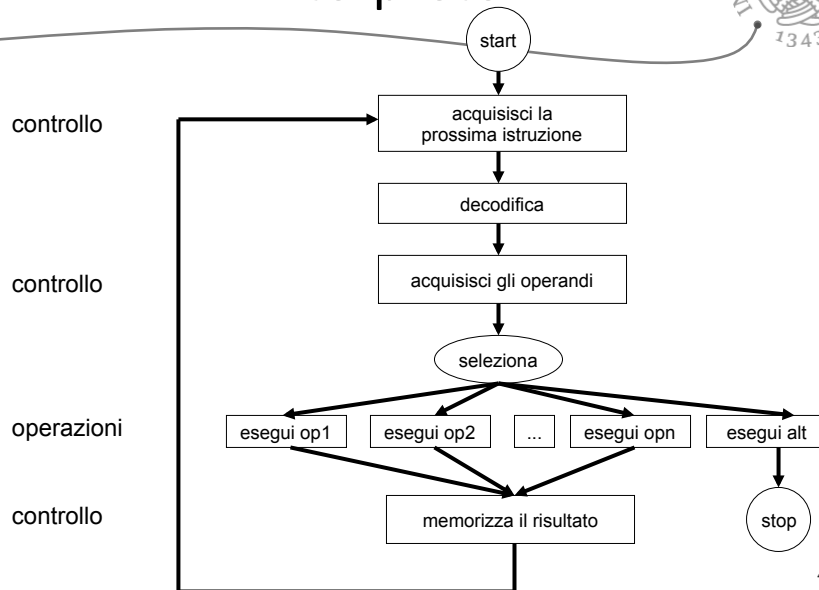
Componente di controllo



- Una collezione di strutture dati ed algoritmi per
 - acquisire la prossima istruzione
 - gestire le chiamate e 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
- ✎ Alle 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

Implementare macchine astratte



- ✎ **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

Da linguaggio a 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 e **M_O** 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

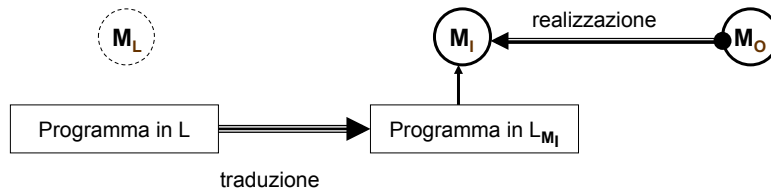
Implementare un linguaggio



- **L** linguaggio ad alto livello
- **M_L** macchina astratta di **L**
- **M_O** macchina ospite
- **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
 - il problema è quello della dimensione del codice prodotto
- **Casi limite che nella realtà non esistono quasi mai**

8

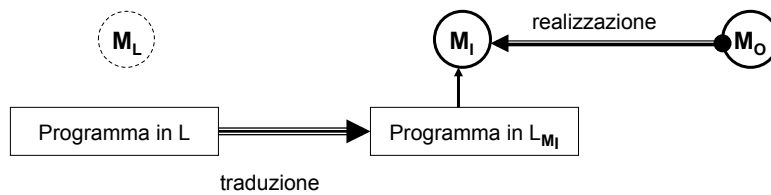
La macchina intermedia



- ☞ **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

Intepretazione e traduzione pure



- ☞ **$M_L = M_I$** 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_1 coincide con quello della macchina ospite M_0
- ☞ Che differenza esiste tra M_1 e M_0 ?
 - il **supporto a tempo di esecuzione (rts)**
 - ✓ collezione di strutture dati e sottoprogrammi che devono essere caricati su M_0 (estensione) per permettere l'esecuzione del codice prodotto dal traduttore (compilatore)
 - $M_1 = M_0 + rts$
- ☞ Il linguaggio L_{M_1} è il linguaggio macchina di M_0 esteso con chiamate al supporto a tempo di esecuzione

11

A cosa serve il rts?



- ☞ Un esempio da un linguaggio antico (**FORTTRAN**): 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 a una routine (indipendente dal particolare programma)
 - ✓ tale routine deve essere caricata su M_0 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 routine 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 M_I è diverso da 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 più macchine ospiti
 - si deve reimplementare l'interprete del linguaggio intermedio
 - non è necessario reimplementare 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 routine 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
 - ✓ il 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 essere risolte solo 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 rts



- ☛ Due aspetti essenziali nella nostra visione (intendendo quella del corso) dei linguaggi di programmazione
 - semantica formale
 - ✓ eseguibile, implementazione ad altissimo livello
 - implementazioni o macchine astratte
 - ✓ interpreti e supporto a tempo di esecuzione

21

Perché?



- ☛ 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

22

Perché?



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

23

Perché?



- **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

24

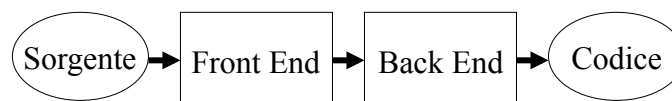
E il compilatore?



- 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

Compilatore



Front end: fasi di analisi

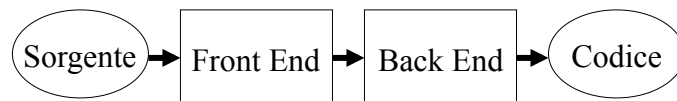
Legge il programma sorgente e determina la sua struttura sia sintattica che semantica

Back end: sintesi

Genera il codice nel linguaggio macchina, programma equivalente al programma sorgente

26

Compilatore

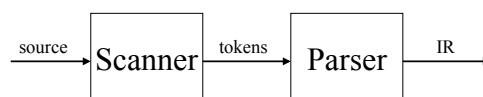


Aspetti critici

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

27

Front End



Due fasi principali

- scanner: compito di trasformare il programma sorgente nel lessico (token)
- 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 chiave: 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)

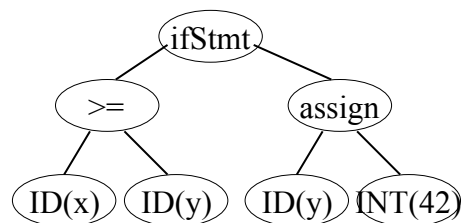


- Formati differenti
- Formato tipico riconosciuto: albero di sintassi astratta (abstract syntax tree)
 - la struttura sintattica essenziale del programma senza gli aspetti di zucchero sintattico
 - ne parliamo anche nel seguito

Parser: AST



- Abstract Syntax Tree (AST)



IF LPAREN ID(x) GEQ ID(y)
RPAREN ID(y) BECOMES INT(42) SCOLON

AST



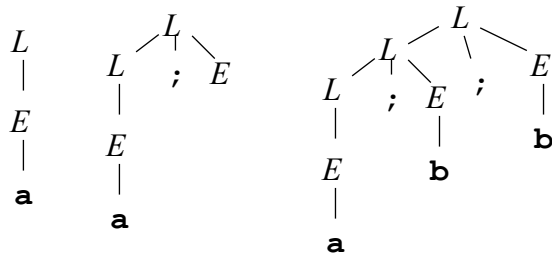
- 👁️ Gli alberi di sintassi astratta sono particolarmente rilevanti perché mostrano la struttura semantica significativa dei programmi.
- 👁️ Noi nel seguito considereremo sempre la sintassi astratta!!
 - senza considerare gli aspetti di dettaglio quali precedenza operatori, ambiguità, etc.

33

AST: esempi



G: $L \rightarrow L ; E \mid E$
 $E \rightarrow a \mid b$

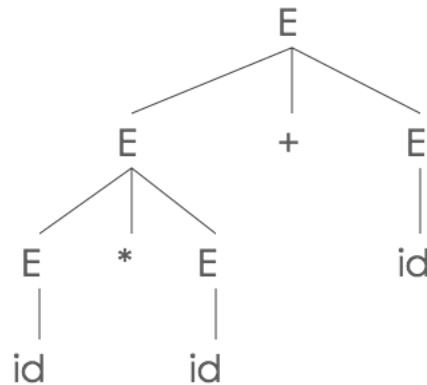


34

Derivazioni e AST



E
 $\rightarrow E + E$
 $\rightarrow E * E + E$
 $\rightarrow id * E + E$
 $\rightarrow id * id + E$
 $\rightarrow id * id + id$



35

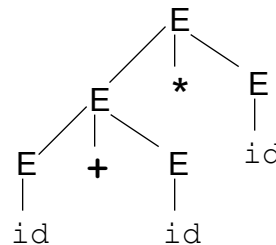
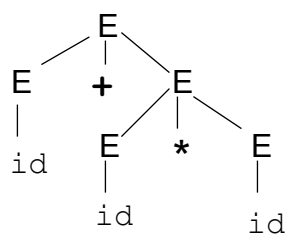
Ambiguità



Programma corretto con AST diversi

Esempio

$E \rightarrow E + E \mid E * E \mid id$



36

Come si risolve?



- Esistono più metodi
- Ad esempio, codificare nelle regole della grammatica la precedenza degli operatori

$$E \rightarrow E' + E \mid E'$$

$$E' \rightarrow id * E' \mid id \mid (E) * E' \mid (E)$$

37

Morale



- La teoria (grammatiche e linguaggi formali) aiuta a strutturare le grammatiche in modo tale da evitare i problemi come quello dell'ambiguità
 - Tanti altri ancora...
- Tutte queste problematiche le vedrete nella magistrale...

38

Sintassi astratta



La sintassi astratta di un linguaggio è espressa facilmente coi tipi di dato algebrici di Ocaml

- ogni categoria sintattica diventa un tipo di dato algebrico di Ocaml

BNF

BoolExp =

| True
| False
| NOT BoolExp
| BoolExp AND BoolExp

Algebraic Data Type

Type BoolExp =

| True
| False
| Not of BoolExp
| And of BoolExp * BoolExp

39

Esempio



Nome.	Produzione grammaticale
EAdd.	Exp ::= Exp "+" Exp1 ;
ESub.	Exp ::= Exp "-" Exp1 ;
EMul.	Exp1 ::= Exp1 "*" Exp2 ;
EDiv.	Exp1 ::= Exp1 "/" Exp2 ;
EInt.	Exp2 ::= Integer ;

```
type exp =  
  EAdd of exp * exp  
| ESub of exp * exp  
| EMul of exp * exp  
| EDiv of exp * exp  
| EInt of int
```

40

AST in Java



- ☞ Potremmo codificare la sintassi astratta di un linguaggio anche in Java
- ☞ In che modo?
 - ogni categoria sintattica è una classe astratta
 - ogni costruttore sintattico è una sottoclasse che estende la classe astratta

41

AST in Java (esempio)



- ☞

```
public abstract class Exp { ... }
```
- ☞

```
public class ESub extends Exp {  
    public final Exp exp_1, exp_2;  
    public ESub(Exp p1, Exp p2) {  
        exp_1 = p1; exp_2 = p2;  
    }  
    :  
}
```

42

Analisi semantica (statica)



- Tipicamente dopo la fase di parsing
 - type checking
 - uso e allocazione delle risorse
 - ottimizzazione del codice

Back End



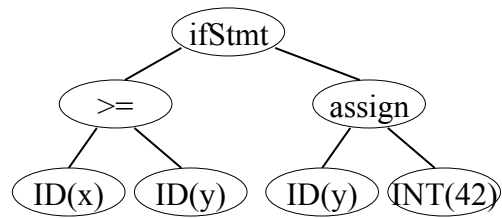
- Cosa fa?
 - traduce il codice intermedio nel linguaggio della macchina ospite
 - usa le risorse della macchina ospite in modo effettivo



Il risultato complessivo

Input

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



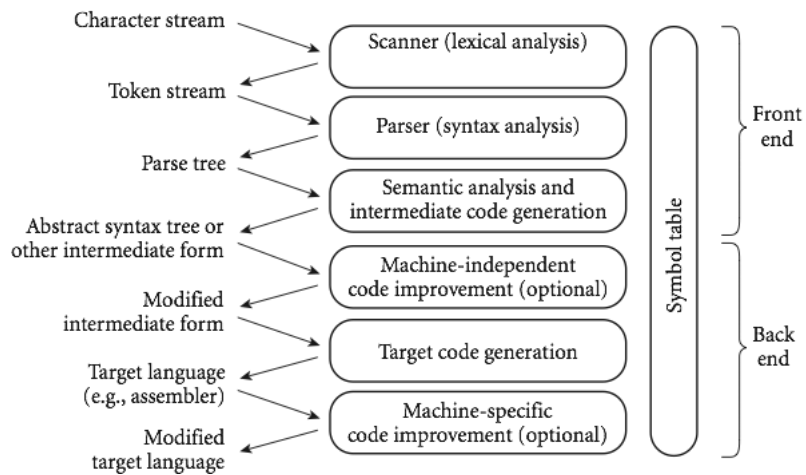
Output

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



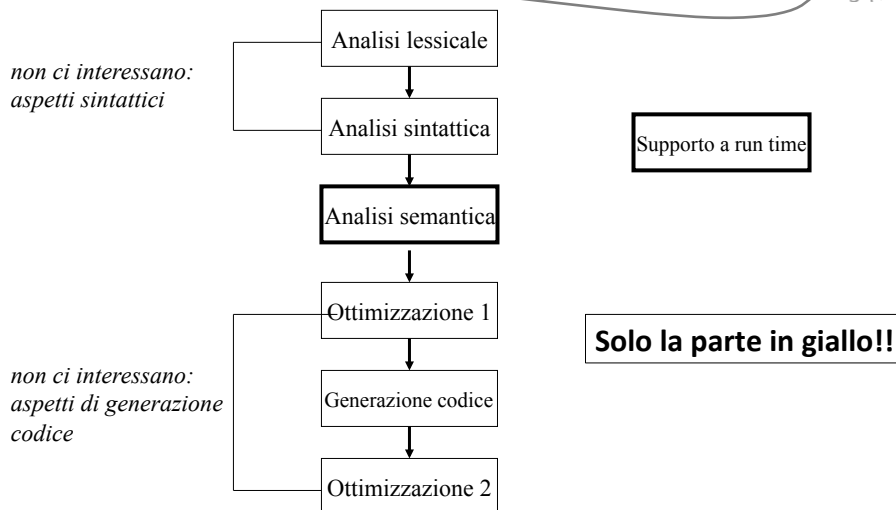
Mettiamo insieme le cose

Struttura di un compilatore



47

Cosa ci interessa?



48

JIT compiler



- 👁️ Idea: compilare il byte-code nel codice nativo durante l'esecuzione
- 👁️ Vantaggi
 - programma continua a essere portatile;
 - esecuzioni "ottimizzate" (code inlining)
- 👁️ Svantaggi
 - rts molto complicato (ottimizzare long-running activation)
 - costo della compilazioni JIT
- 👁️ Noi non ne parliamo!