

Automati e Linguaggi Formali

Struttura di un compilatore e fasi principali

01 Ottobre 2014

A.A. 2014-2015
Enrico Mezzetti
emezzett@math.unipd.it



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Linguaggi di programmazione

- **Linguaggio di programmazione** e' un formalismo per descrivere calcoli e computazioni (i.e., programmi)
- **Programma**, per essere eseguito, deve essere tradotto in una forma che puo' essere capita da un calcolatore \Rightarrow **compilatore**
- Principi e tecniche per il riconoscimento/compressione dei linguaggi utilizzate nella progettazione di compilatori
 - Ricorrenti in molte altre aree dell'informatica
 - linguaggi di programmazione
 - architettura
 - teoria dei linguaggi
 - algoritmi
 - ingegneria del software
 - ...

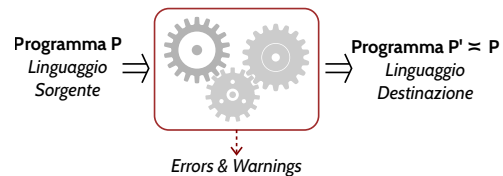


Automati e Linguaggi Formali – A.A. 2014-2015
Docente: Enrico Mezzetti

2 of 20

Categorie di compilatori

- **Compilatore**: traduttore di programmi



- Cattura gli errori identificati durante la traduzione
- Se P' e' eseguibile allora

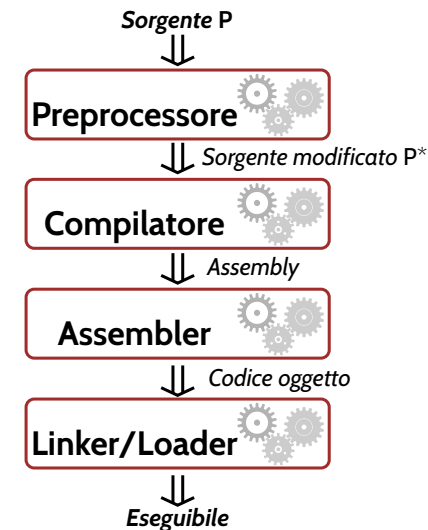
input \rightarrow **P'** \rightarrow output

- **Interprete**: esecutore diretto di programmi

sorgente \rightarrow **Interprete** \rightarrow output
input \rightarrow

- Generalmente piu' lento
- Soluzioni ibride

Ruolo effettivo dei compilatori



Automati e Linguaggi Formali – A.A. 2014-2015
Docente: Enrico Mezzetti

3 of 20



Automati e Linguaggi Formali – A.A. 2014-2015
Docente: Enrico Mezzetti

4 of 20

Struttura di un compilatore

Front-end (Analisi)

- Identifica elementi base del *programma sorgente* e li inserisce in una struttura grammaticale;
- Crea una rappresentazione intermedia del programma sorgente;
- Segnala possibili errori
- Genera *tavola dei simboli* e applica ottimizzazioni

Back-end (Sintesi)

- Costruisce il *programma destinazione* partendo da rappresentazione intermedia e tavola dei simboli, applicando ulteriori ottimizzazioni

- **Sequenza di fasi:** **analisi lessicale**, **analisi sintattica**, **analisi semantica**, generazione del codice intermedio, ottimizzazione del codice, generazione del codice



Analisi lessicale (scanning)

- Legge programma sorgente come flusso di caratteri
 - Identifica sequenze complete e significative (**lexemi**)
- Per ogni lexema genera un **token**

$\langle \text{nome-token}[\text{valore-attributo}] \rangle$
simbolo astratto \uparrow \uparrow posizione nella symbol table

- Esempio: `posizione = iniziale + velocita' * 60`

- `posizione` \Rightarrow $\langle \text{id},1 \rangle$ (posiz. 1 in symbol table)
- `=` \Rightarrow $\langle = \rangle$ (no valore-attributo)
- `iniziale` \Rightarrow $\langle \text{id},2 \rangle$
- `+` \Rightarrow $\langle + \rangle$
- `velocita'` \Rightarrow $\langle \text{id},3 \rangle$
- `*` \Rightarrow $\langle * \rangle$
- `60` \Rightarrow $\langle 60 \rangle$
- $\langle \text{id},1 \rangle \langle = \rangle \langle \text{id},2 \rangle \langle + \rangle \langle \text{id},3 \rangle \langle * \rangle \langle 60 \rangle$

- Approcci: **linguaggi regolari**, **automi a stati finiti**, **espressioni regolari**



Analisi sintattica (parsing)

- Opera sull'output dell'analisi lessicale (insieme di token)
- *Parser* genera una rappresentazione intermedia ad albero
 - Indica la struttura grammaticale della sequenza di token
- **Albero sintattico**
 - Nodo interno = operazione, figli = operandi
 - Indica l'ordine in cui eseguire le operazioni (precedenza tra operatori)
- Approcci: **grammatiche libere da contesto**, **automi a pila**, **linguaggi liberi da contesto**



Analisi sintattica (parsing) - 2

- Esempio di costruzione

$\langle \text{id},1 \rangle \langle = \rangle \langle \text{id},2 \rangle \langle + \rangle \langle \text{id},3 \rangle \langle * \rangle \langle 60 \rangle$
(posizione = iniziale + velocita' * 60)

```
=
 / \
<id,1> +
      / \
      <id,2> *
           / \
           <id,3> 60
```

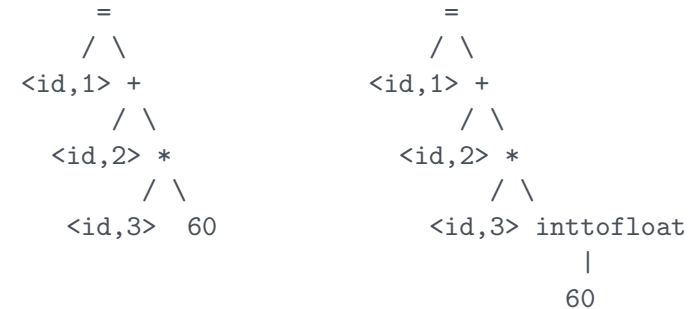
- Costruzione bottom-up:

- 1 Moltiplicare il lexema di $\langle \text{id},3 \rangle$ (*velocita'*) con 60
- 2 Sommare il risultato con il valore di *iniziale*
- 3 Memorizzare il risultato nella locazione *posizione*



- Opera sull'output delle fasi precedenti (tabella e albero)
 - Controllo della *consistenza semantica* del programma (rispetto alla definizione del linguaggio)
 - Collezione informazioni sui tipi e le mette nella tavola dei simboli (per la fase di generazione del codice)
- **Type checking** (controllo dei tipi):
 - Tipi degli operandi devono essere consistente con quelli attesi dall'operatore
 - E.g., un indice di un array deve essere un intero (errore se ad esempio floating-point)
- **Coercion** (conversione forzosa)
 - Se un'operando si applica a piu' tipi (e.g., int o float)
 - se un operando e' int e l'altro e' float applichiamo una conversione a float

- Esempio di Coercion
 - posizione, iniziale, e velocita' sono dichiarati float
 - 60, invece, e' un int
 - Operazione * richiede una coercion



- Rappresentazioni intermedie di basso livello
 - Semplici e facili da tradurre in codice macchina
- Tipico esempio **codice a tre indirizzi** (*three-address code*)
 - Istruzioni simili all'assembly con tre operandi per istruzione
- Nel solito esempio otterremmo:


```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```
- Proprieta'
 - Al piu' un operatore sulla destra → le istruzioni danno l'ordine in cui effettuare le operazioni (* ← +)
 - Un nome temporaneo generato per contenere il valore calcolato da ogni istruzione (~registri virtuali)

- Miglioramento del codice intermedio in funzione di un migliore codice oggetto
 - Piu' veloce, piu' piccolo, meno energia necessaria, ecc.
- Esempio:


```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

$\xrightarrow{\text{Diventa}}$

```
t1 = id3 * 60.0
id1 = id2 + t1
```
- opt1 Conversione da integer a floating-point fatta a tempo di compilazione (e non a tempo di esecuzione), e quindi eliminata dal codice
- opt2 t3 era usato solo per trasmettere un valore ad id1

Generazione del codice

- Da rappresentazione intermedia a codice oggetto
- Registri o locazioni di memoria per ogni variabile del codice
- Da ogni istruzione in codice intermedio a sequenza di istruzioni in codice macchina

Esempio:

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Diventa →

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

- gen1 Il primo operando specifica una destinazione, F: floating-point
- gen2 id3 in R2, poi moltiplicazione con 60.0 e risultato in R2, poi id2 in R1, poi somma di R1 e R2 in R1, poi R1 in id1



Strumenti per la costruzione di un compilatore

- **Generatori di parser:** generano automaticamente un parser dalla descrizione della grammatica di un linguaggio
- **Generatori di scanner:** generano automaticamente un analizzatore lessicale dalla descrizione tramite espressioni regolari dei token di un linguaggio
- **Generatori di generatori di codice:** da regole per tradurre ogni operazione in una sequenza di istruzioni in linguaggio macchina



Ruolo dell'analisi lessicale

- Prima fase del processo di compilazione
 - Legge i caratteri del programma e li raggruppa in *lexemi*,
 - Per ogni lexema produce un *token* → Parser
 - Lexemi che rappresentano indentificatori vanno nella tabella dei simboli
- Altre operazioni ausiliarie
 - Elimina spazi bianchi e commenti, mantiene le corrispondenze con i file sorgenti, espande le macro, ...
- Due fasi effettive
 - **Scanning** ► compattazione della stringa di caratteri
 - **Analisi lessicale** ► estrazione tokens da lexemi
- Diagramma o altro modo di descrivere i lexemi di ogni token
 - Useremo **automati a stati finiti** e **espressioni regolari** (dimostreremo sono equivalenti)



Token, pattern, lexemi

- **Token:** coppia <nome token, valore attributo>
 - nome token: simbolo astratto che rappresenta un'unità lessicale (parola chiave, identificatore, ecc.)
- **Pattern:** descrizione della forma che i lexemi di un token possono avere
 - Token parola-chiave ⇒ il pattern e' la sequenza di caratteri che formano la parola chiave
 - Token identificatore ⇒ il pattern e' pi complesso e pu corrispondere a piu' stringhe di caratteri
- **Lexema:** sequenza di caratteri del programma sorgente che rispetta il pattern di un token
- Esempi:

Token	Pattern	Matching lexemes
if	caratteri i e f	if
id	caratteres seguito da altri caratteri e/o cifre	posizione, iniziale
number	numero	3.14159, 0, 6.2



Classi tipiche di token

- Classi di token (dipendono dal linguaggio)
 - *Keywords*: un token per ogni parola chiave
 - *Operators*: vari token per gli operatori
 - *Identifiers*: un token per tutti gli identificatori
 - *Constants*: vari token per le costanti (numeri, letterali, ecc.)
 - *Punctuation*: per altri simboli (parentesi, virgola, ecc.)
- **Nome token** → quale tipo di token
- **Valore-attributo** → identifica uno specifico lexema come istanza del token
 - Più attributi necessari ▶ puntatore a tabella dei simboli



Esempio

- Comando Fortran $E = M * C ** 2$
- Token generati:
 - <id, puntatore ad E nella tavola dei simboli>
 - <assign-op>
 - <id, puntatore ad M nella tavola dei simboli>
 - <mult-op>
 - <id, puntatore a C nella tavola dei simboli>
 - <exp-op>
 - <number, valore intero 2>
- Non è sempre banale riconoscere i lexemi (ambiguità)



Input buffering

- Esempio: per sapere di essere alla fine di un identificatore, dobbiamo almeno leggere un carattere che non è né una lettera né una cifra
- Esempio (C): < potrebbe essere un operatore oppure l'inizio di <=
- Esempio (Fortran 90, spazi sono irrilevanti):
 $DO\ 5\ I = 1.25$ vs. $DO\ 5\ I = 1,25$
 - $DO5I=1.25$ → simple assignment
 - $DO\ 5\ I=1, 25$ → repeat line 5, 25 times using I as a counteril primo lexema è la parola chiave DO (comando for)
- Spesso è necessario leggere oltre la fine di un lexema per intercettarlo
- Due puntatori per scorrere l'input: uno all'inizio del lexema corrente, uno per leggere oltre, finché non si trova un lexema



Specifiche dei token

- Metodi formali per la specifica dei token
 - Linguaggi regolari
 - Espressioni regolari per descrivere i token
 - Automi a stati finiti per descrivere analizzatori lessicali
 - passaggio da espressione regolare ad automa

