

Corso: Fondamenti, Linguaggi e Traduttori

Paola Giannini

Introduzione ai compilatori

- **Processamento dei linguaggi**: preparare un programma ad essere eseguito su un computer
- Assicura che il programma conformi con la semantica intesa
- Traduce in una forma più facilmente eseguibile da computer. I due estremi:
 - **Interprete**, gira il programma esaminando la struttura dei costrutti e simulandone le azioni
 - **Compilatore**, traduce il programma in istruzioni macchina eseguibili direttamente nel computer

- Il termine **compilatore** fu coniato da by Grace Murray Hopper nel 1950.
- La traduzione era una “compilazione” di una sequenza di sottoprogrammi selezionati da una libreria in linguaggio macchina, si parlava di **programmazione automatica**.
- Primo linguaggio compilato è stato il FORTRAN (fine anni 50). Procedimento ad hoc!
- Il compilatore effettuava “ottimizzazioni” per rendere competitivo il codice macchina prodotto rispetto alla programmazione in assembler. (L’efficienza era vitale, date le capacità limitate dei computer!)
- I compilatori attuali sono costruiti in modo modulare, comunque il processo di mettere insieme un compilatore efficiente è ancora un lavoro complesso.

Oltre la traduzione di linguaggio di programmazione ad alto livello a istruzioni macchina eseguibili.

- Queste note sono scritte in $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ un linguaggio di formattazione di testi e compilate per produrre un file pdf (ma potrei produrre anche PostScript)
- PostScript è un linguaggio di programmazione che viene tradotto ed eseguito da una stampante o da un visualizzatore postscript per produrre una forma leggibile di un documento. Un linguaggio per la rappresentazione dei documenti indipendente da come e dove i documenti sono stati creati e da come verranno visualizzati.
- Verilog e VHDL sono linguaggi per la creazione di circuiti VLSI. Vengono processati da un **silicon compiler**, che, come un compilatore “tradizionale”, comprende e applica le regole di progettazione che determinano la possibilità di realizzare il circuito.

- Le tecniche utilizzate per lo sviluppo dei compilatori trovano impiego in numerosi problemi.
 - Le tecniche usate in un analizzatore lessicale possono essere utilizzate negli editor di testi, nei sistemi di “information retrieval”.
 - Le tecniche usate in un parser possono essere utilizzate nei sistemi di processamento delle interrogazioni come SQL.
 - Molti software che hanno un front-end complesso possono utilizzare tecniche utilizzate nel progetto dei compilatori, es: risolutori di equazioni, processamento del linguaggio naturale

- **linguaggio macchina puro**

- codice per un particolare set di istruzioni macchina non assumendo l'esistenza di alcun sistema operativo o libreria di funzioni
- questo approccio è molto raro, relegato a linguaggi per implementazione di sistemi operativi o applicazioni "embedded"

- **linguaggio macchina "augmented"**

- i compilatori generano codice per un particolare set di istruzioni macchina arricchito con routine di sistema operativo o di supporto (I/O, allocazione di memoria), che vanno linkate al codice oggetto

- **Assembler**

- Viene prodotto un file di testo contenente il codice sorgente assembler. Un certo numero di decisioni nella generazione del codice (target delle istruzioni di salto, forma degli indirizzi, ...) sono lasciate all'assemblatore.
- Molte volte il C é usato come assembler. Il C viene chiamato **linguaggio assembler universale**, per il fatto che è relativamente a basso livello, ma anche indipendente dalla piattaforma.

Codice generato (3)

- Codice per una Macchina Virtuale

- Permette di generare codice eseguibile indipendente dall'hardware
- Se la macchina virtuale è semplice, il suo interprete può essere facile da scrivere
- Questo approccio penalizza la velocità di esecuzione di un fattore da 3:1 a 10:1. La compilazione "Just in Time" (JIT) può tradurre porzioni di codice virtuale in codice nativo.

- Vantaggi

- semplificare un compilatore fornendo le primitive adatte (ad esempio come chiamate di metodo, manipolazione di stringhe,...)
- diminuzione della dimensioni del codice generato se le istruzioni sono progettate per una particolare linguaggio di programmazione (per esempio codice JVM per Java)
- trasportabilità del compilatore
- possibilità di verificare che il codice non contenga istruzioni "maliziose"

Traduzione Diretta dalla Sintassi

- Sarebbe difficile progettare un algoritmo di traduzione di un linguaggio complesso senza prima analizzare la frase sorgente nei suoi costituenti definiti dalla sintassi.
- L'analisi sintattica riduce la complessità del problema del calcolo della traduzione, decomponendo il problema in sotto-problemi più semplici. (In modo simile l'analisi lessicale riduce la complessità dell'analisi sintattica.)
- Per ogni costituente della frase sorgente il traduttore esegue le azioni appropriate per preparare la traduzione, azioni che consistono nel raccogliere informazioni sui nomi (nella tabella dei simboli), nel fare dei controlli semantici, ecc..
- Il lavoro del traduttore è modularizzato secondo la struttura descritta dall'albero sintattico della frase.
- Questo tipo di traduttori è detto **guidato dalla sintassi**.

❶ approccio in due passate:

- a) il parser sintetizza l'Abstract Syntax Tree (AST), che è un albero di parsing in forma astratta nella fase di analisi sintattica
- b) si utilizza l'AST per l'analisi semantica e poi per la sintesi dell'output

❷ approccio semplificato (una passata):

- si sintetizza l'output direttamente durante l'analisi sintattica, senza costruire l'AST
- nel primo caso si ha a disposizione l'AST per fare analisi semantica e elaborazione dell'output (si possono fare diverse passate sull'AST).
- nel secondo caso non si memorizza l'albero, ma si è legati a come si fa il parsing (se top-down) o bottom-up. Non si ha a disposizione tutto l'albero per cui si possono fare SOLO elaborazioni semplici.

Abstract Syntax tree e Parse tree

- il **parse tree** è l'albero corrispondente alla derivazione di una stringa della grammatica: il parse tree contiene tutti i dettagli del parsing (cioè i nodi per tutti i simboli usati nella derivazione)
- l'**AST** è definito per rendere possibile l'analisi semantica e la traduzione ed è una astrazione del parse tree (ha molti meno nodi!)

Grammatica di un semplice linguaggio di programmazione

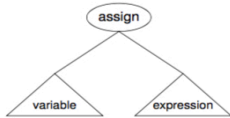
```
Start  → Stm $  
Stmt   → id assign E  
       | if lparen E rparen Stmt else Stmt fi  
       | while lparen E rparen Stmt od  
       | { StmLs }  
Stmts  Stmts ; Stmt  
       | Stmt  
E       → E plus T  
       → E minus T  
       | T  
T       → id  
       | intNum  
       | floatNum
```

Dove in blu sono i token (i terminali della grammatica). Per esempio **lparen** token per “(”, **rparen** token per “)”, **assign** token per “=”, **plus** token per “+” e **minus** token per “-”, ecc..

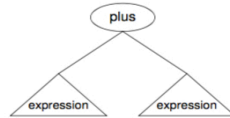
Sintetizzano l'informazione derivata dal parsing. Alcuni alberi standard per costrutti di linguaggi.

- `if`
- `while`
- `assegnamento`
- `operatori binari`
- `blocco`

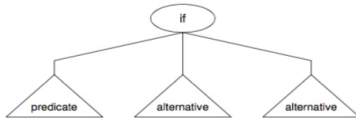
AST dei costrutti del linguaggio



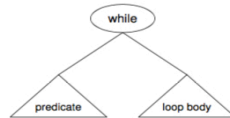
(a)



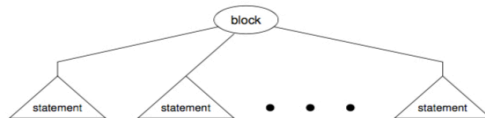
(b)



(c)

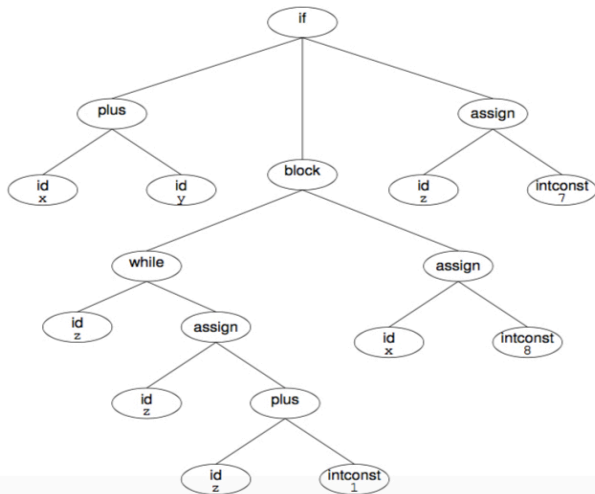


(d)



(e)


ABS del programma

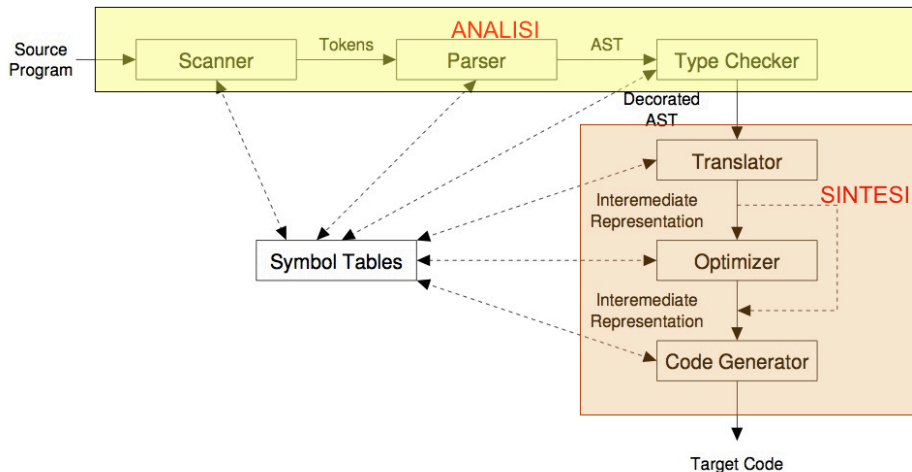


Ci sono due fasi fondamentali in un compilatore:

- ❶ **Analisi:** viene creata una rappresentazione intermedia del programma sorgente.
 - Analisi Lessicale
 - Analisi Sintattica
 - Analisi Semantica
- ❷ **Sintesi:** viene creato, partendo dalla rappresentazione intermedia, il programma target equivalente
 - Generazione di codice intermedio
 - Ottimizzazione
 - Generazione di Codice

Fasi di compilazione di Compilatore a due passate

Ogni fase trasforma da una rappresentazione del programma sorgente in un'altra rappresentazione. 



- Legge il programma sorgente carattere per carattere e
 - ritorna i **tokens** del programma sorgente,
 - scopre eventuali errori lessicali, ed
 - elimina informazioni non necessarie (commenti).
- Un **token** descrive un insieme di caratteri che hanno lo stesso significato (es: identificatore, operatori, keywords, numeri, delimitatori etc).
- Le **espressioni regolari** sono utilizzate per descrivere i token.
- Gli **automi a stati finiti deterministici** possono essere usati per implementare un analizzatore lessicale.

Scanner (2)

Input:

```
// Assegna a newVal oldVal + 1 - 3,2  
newVal = oldVal + 1 - 3.2
```

Output:



```
<id : newVal><assign><id : oldVal><plus><intNum : 1><minus><floatNum : 3.2>
```



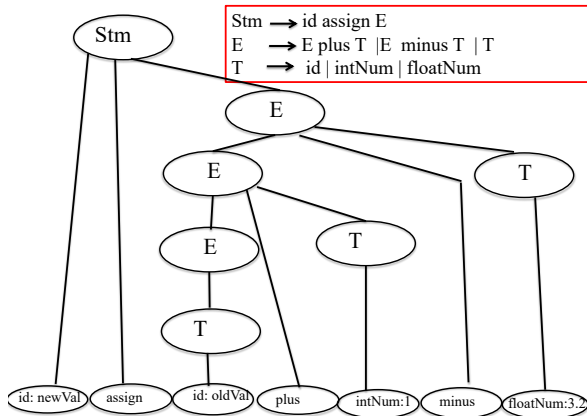
- La sintassi di un linguaggio è specificata per mezzo di una **grammatica context-free, CFG**.
- Il parser legge la sequenza di tokens e riconosce la struttura sintattica, sotto forma di un albero sintattico. Cioè riconosce quali produzioni sono state usate per generare la sequenza di token.
- Un analizzatore sintattico verifica se un dato programma è derivabile dal simbolo iniziale della grammatica del suo linguaggio, cioè soddisfa le regole derivate dalle produzioni della sua CFG.
- In questa fase si trovano errori sintattici.

Parser (2)

Dalla sequenza di **token**

`<id : newVal><assign><id : oldVal><plus><intNum : 1><minus><floatNum : 3.2>`

ottengo un **parse tree** dal quale posso capire le produzioni che sono state usate per generare la stringa dalla grammatica:



Differenze fra Analisi Lessicale e Sintattica

- Quali costrutti di un programma sono riconosciuti da un analizzatore lessicale, e quali da un analizzatore sintattico?
 - L'analizzatore lessicale tratta solo i costrutti del linguaggio **non propriamente ricorsivi**.
 - L'analizzatore sintattico consente di trattare i costrutti del linguaggio **ricorsivi**.
 - L'analizzatore lessicale semplifica il lavoro dell'analizzatore sintattico, riconoscendo i componenti più piccoli, i tokens, del programma sorgente.
 - L'analizzatore sintattico lavora sui tokens riconoscendo le strutture del linguaggio.

Consideriamo un linguaggio di espressioni.



$$E \rightarrow E + E \mid E * E \mid (E) \mid Id \mid N$$

$$Id \rightarrow L \mid L Id$$

$$N \rightarrow C \mid C N$$

$$L \rightarrow a \mid b \mid \dots \mid z$$

$$C \rightarrow 0 \mid 1 \mid \dots \mid 9$$

sequenza di lettere
sequenza di cifre

- 1 Cosa rende questo linguaggio NON regolare (libero da contesto)?
- 2 Il linguaggio generato da E ha qualche "sottolinguaggio" regolare?

In relazione a come viene costruito il parse tree ci sono due differenti tecniche:

① Top-Down



- La costruzione inizia dalla radice e procede verso le foglie.
- Efficienti parser top-down possono facilmente essere costruiti a mano.
- **Parsing a discesa ricorsiva predittivo** (implementato con funzioni mutuamente ricorsive),
- **Parsing predittivo non a discesa ricorsiva** (implementato usando l'automa pushdown).
- Viene prodotta la derivazione “Leftmost” della stringa analizzata.

② Bottom-Up

- La costruzione inizia dalle foglie e procede verso la radice, producendo la la derivazione “Rightmost” della stringa analizzata.
- Di norma efficienti parser bottom-up parsers sono creati con l'ausilio di tools software.
- I parser Bottom-up sono anche conosciuti con il nome di **parser shift-reduce**.
- Ci sono molte variazioni di LR Parsing: LR, SLR, LALR



- Controlla che le variabili siano correttamente dichiarate e che i tipi siano corretti.
- Queste informazioni non possono essere rappresentate con una grammatica context-free.
- In questa fase, nel caso di compilatore a 2 passate, il type-checker **decora l'AST** aggiungendogli le **informazioni di tipo**.
- Questa fase dipende dalle regole semantiche del linguaggio sorgente (è indipendente dal linguaggio target).



- Raggiungibilità del codice
- Inizializzazione variabili prima dell'uso
- Corretto ritorno di valori da funzioni/metodi



- Un compilatore può produrre una rappresentazione esplicita del codice intermedio.
- Questo **codice intermedio** (o **Intermedie Representation = IR**) è indipendente dall'architettura del computer per cui si genera codice, ma può essere più o meno ad alto livello.
- Esempio:

`newVal = oldVal + 1 - 3.2`

potrebbe essere tradotto in

```
ADD id2,#1,temp1
SUB temp1,#3.2,temp2
MOV temp2,id1
```

- La generazione del codice è **diretta dalla sintassi**.

- La generazione del codice richiede che venga catturata la **semantica dinamica** (cosa fa a runtime) di un costrutto.
- Per esempio l'AST di un "while" contiene due sotto-alberi, uno per controllare l'espressione di controllo, e l'altro per il corpo del ciclo, ma non dice che un "while" ripete il body. Questo è catturato quando un AST di un ciclo while viene tradotto.
- Nella IR, la nozione di verificare il valore dell'espressione di controllo e dell'esecuzione condizionale del corpo del ciclo diventa esplicito.

- Il codice intermedio ha poco della macchina di destinazione. Le informazioni sulla macchina di destinazione (operazioni disponibili, indirizzamento, registri, ecc.) sono riservate per la fase finale di generazione del codice.
- In semplici compilatori non ottimizzati il traduttore genera il codice di riferimento direttamente, senza utilizzare una IR.
- In compilatori complessi, ad esempio **GNU Compiler Collection (GCC)** prima si genera un IR di alto livello (orientato al linguaggio sorgente) e poi questo viene tradotto in un IR a basso livello (orientato alla macchina). Questo approccio consente di separare le dipendenze derivanti dal sorgente e dal target.

- Il codice IR generato dal traduttore viene analizzato e trasformato in codice IR equivalente ottimizzato.
- Un **ottimizzatore** ben progettato può incrementare in modo significativo la velocità di esecuzione.
- Ottimizzazioni possono essere
 - spostare o eliminare operazioni non necessarie
 - rimuovere codice non raggiungibile

Static single assignment (SSA)

- Codice intermedio nel quale ogni variabile è assegnata esattamente una volta ed è definita prima del suo uso.
- Le variabili originali sono replicate in versioni in modo tale da tracciarne la catena di definizione ed uso.
- Questo formato semplifica e migliora i risultati che si possono ottenere nella fase di ottimizzazione, semplificando le proprietà delle variabili. Per esempio:

$$\begin{aligned}y &= 1 \\ y &= 2 \\ x &= y\end{aligned}$$

il primo assegnamento è inutile, ma non è facile da capire, mentre nella sua forma SSA

$$\begin{aligned}y_1 &= 1 \\ y_2 &= 2 \\ x_1 &= y_2\end{aligned}$$

il risultato è immediato!

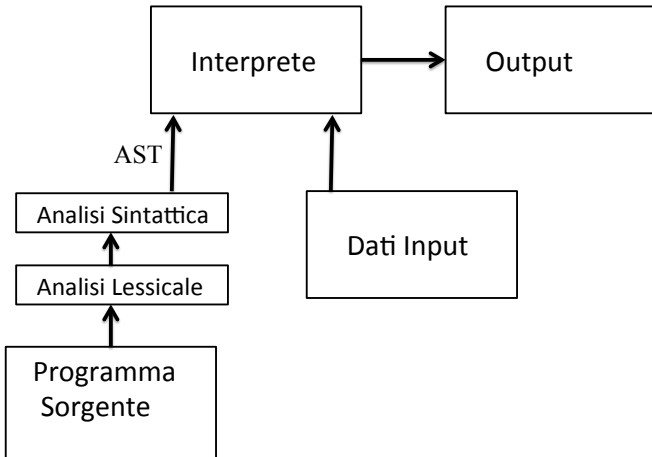
- Il codice IR prodotto dal traduttore viene tradotto nel **codice della macchina target**.
- IL programma target è normalmente un codice oggetto “rilocabile” contenente codice macchina.
- Questa fase fa uso di informazioni dettagliate sulla macchina target e include ottimizzazione legate alla macchina specifica quali l’allocazione dei registri e lo scheduling del codice.
- Il generatore di codice può essere piuttosto complesso. Per produrre buon codice target bisogna considerare di molti casi particolari.
- È possibile generare i **generatori di codice** in modo automatico, definendo dei template che mettano in corrispondenza le istruzioni di un IR a basso livello con quelle della macchina target.
- Il compilatore GNU GCC è un compilatore fortemente ottimizzato che usa files di descrizione per più di dieci architetture PC, e di almeno due linguaggi (C and C++).

- Una **tabella dei simboli** mantiene l'associazione fra gli identificatori e le informazioni ad essi associati (tipo, definizione, ecc.).
- È condivisa dalle varie fasi della compilazioni
- Ogni volta che un identificatore viene usato, la tabella di simboli consente di accedere alle informazioni raccolte quando è stato definito.

Strumenti per la realizzazione di compilatori

- **Generatori di scanner:** producono analizzatori lessicali (input in generale sono espressioni regolari che descrivono i token).
- **Generatori di parser:** producono AST (input la grammatica). Inoltre si possono specificare “attributi” e regole per realizzare analizzatori semantici.
- **Traduttori diretti dalla sintassi:** producono collezioni di routine che visitano l’AST e generano il codice intermedio.
- **Generatori di codice:** prendono regole per tradurre da IL a linguaggio macchina (soluzioni alternative selezionate con “template matching”)

- Ci sono due tipi diversi di interpreti che supportano l'esecuzione di programmi: interpreti di una macchina (astratta/concreta), e interpreti di un linguaggio.
- **Interpreti di Macchine**
 - Simulano l'esecuzione di un programma compilato per una particolare architettura.
 - Java usa un interprete bytecode per simulare l'effetto del programma compilato per la Java Virtual Machine (JVM).
 - I programmi come SPIM simulano l'esecuzione di programma MIPS su un computer non-MIPS (uso didattico).
- **Interpreti di Linguaggi**
 - L'interprete del linguaggio simula l'effetto dell'esecuzione di un programma senza compilarlo in un particolare linguaggio macchina. Per l'esecuzione viene usato **direttamente l'AST**.



- È possibile **aggiungere** codice a tempo di esecuzione.
- È possibile **generare** codice a tempo di esecuzione.
- In Python e Javascript, per esempio, qualsiasi variabile stringa può essere interpretata come una espressione ed eseguita.
- Il tipo di una variabile può cambiare dinamicamente. I tipi vengono testati a tempo di esecuzione (non compilazione).
- Gli interpreti supportano l'indipendenza dalla macchina. Tutte le operazioni sono eseguite nell'interprete.

- Non è possibile effettuare ottimizzazione del codice
- Durante l'esecuzione il testo del programma è continuamente riesaminato, i tipi e le operazioni sono ricalcolate anche ad ogni uso.
- Per linguaggi molto dinamici questo rappresenta un overheads 100:1 nella velocità di esecuzione rispetto al codice compilato.
- Per linguaggi più statici (come C o Java), il degrado della velocità è dell'ordine di 10:1.
- Il programma sorgente spesso non è compatto come se fosse compilato. Ciò può causare una limitazione nella dimensione dei programmi eseguibili.