

## Il Modello di un Compilatore

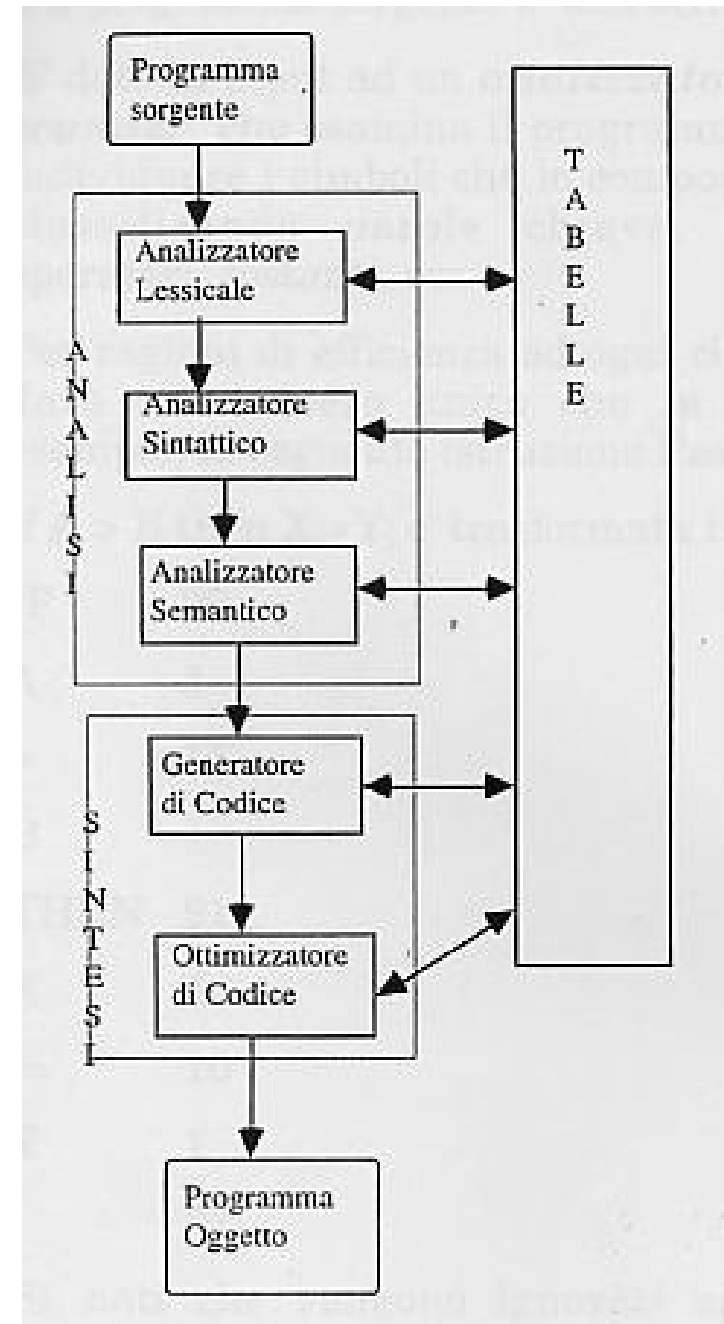
La costruzione di un compilatore per un particolare linguaggio di programmazione e' abbastanza complessa.

La complessità dipende dal linguaggio sorgente.

**Compilatore:** traduce il programma sorgente in programma oggetto.

Esegue:

- **analisi** del programma sorgente;
- **sintesi** del programma oggetto.



Un programma sorgente e' una stringa di simboli.

E' dato in input ad un *analizzatore lessicale* o *scanner* che esamina il programma sorgente per individuare i simboli che lo compongono (tokens) classificando parole chiave, identificatori, operatori, costanti, ecc.

Per ragioni di efficienza ad ogni classe di token è dato un numero unico che la identifica. Ad esempio, la seguente istruzione Pascal:

if A > B then X := Y; e' trasformata in:

IF	20
A	1
>	15
B	1
THEN	21
X	1
:=	10
Y	1
;	27

Si noti che vengono ignorati spazi bianchi e commenti. Inoltre alcuni scanner inseriscono label, costanti e variabili in tavole appropriate.

Un elemento della tavola per una variabile, ad esempio, contiene nome, tipo, indirizzo, valore e linea in cui e' dichiarata.

### Esempio:

```
X1: =a+bb* 12;  
X2: =a/2 + bb *12;
```

Viene trasformato dopo l'analisi lessicale nella seguente sequenza di token:

"X1"	Id
": ="	Op
"a"	Id
"+"	Op
"bb"	Id
"*"	Op
12	Lit
;	Punct
"X2"	Id
": ="	Op
"a"	Id
"/"	Op
2	Lit
"+"	Op
"bb"	Id
"*"	Op
12	Lit
;	Punct

Segue poi *l'analizzatore sintattico (o parser)*.

Individua la struttura sintattica della stringa in esame a partire dal programma sorgente sotto forma di token.

Identifica quindi espressioni, istruzioni, procedure.

**Esempio:** ALFA1:=5+A\*B

La sottostringa 5+A\*B viene riconosciuta come <espressione>, mentre la stringa completa come <assegnazione>, in accordo con la regola sintattica Pascal:

<assegnazione> ::= <variabile> := <espressione>

In realtà si utilizza una stringa semplificata del tipo:

id1 :=c2+id3\*id4

con accesso alla rappresentazione generata dallo scanner.

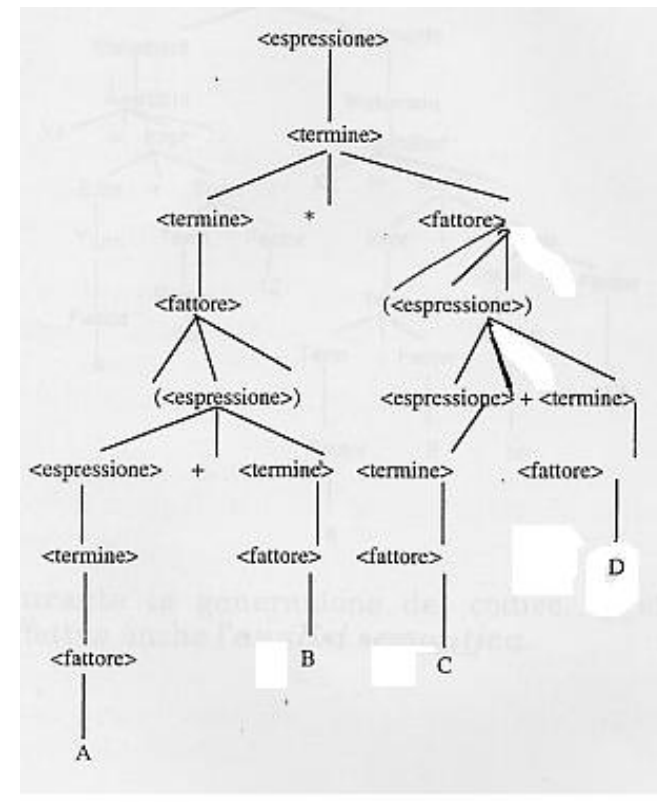
**Altro esempio:**

$(A+B)*(C+D)$

L'analisi produce le classi sintattiche <fattore>, <termine> ed <espressione>.

Il controllo sintattico si basa sulle *regole grammaticali* utilizzate per definire formalmente il linguaggio.

Durante il controllo (sintattico) si genera *l'albero di derivazione (albero sintattico)*.



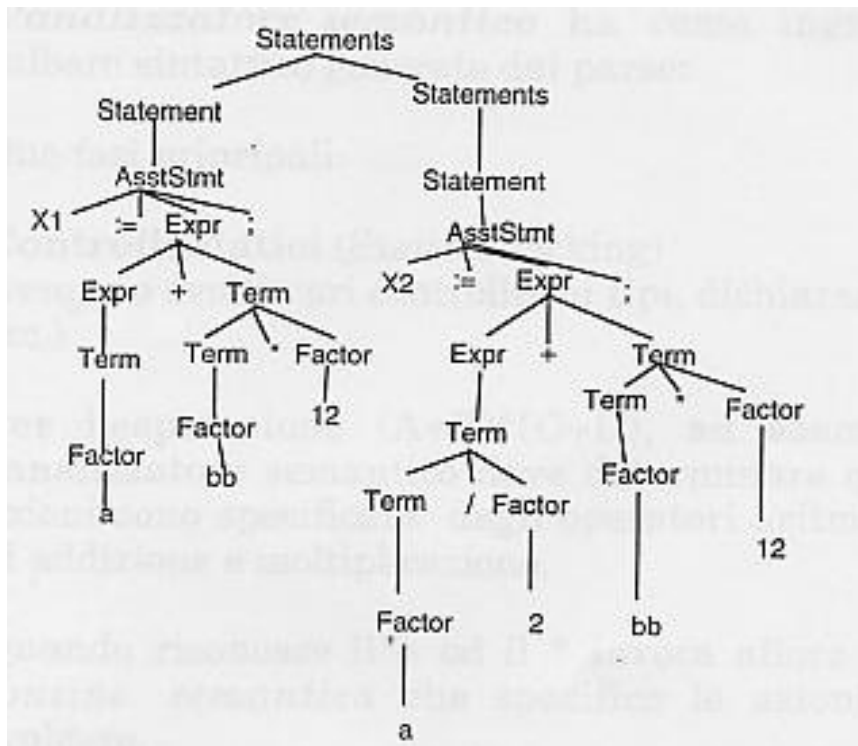
### Esempio:

Dalla lista di token del precedente esempio:

X1: =a+bb\*12;

X2: =a/2+bb\*12;

Si genera il seguente albero sintattico:



Durante la generazione del codice oggetto si effettua anche ***l'analisi semantica***.

***L'analizzatore semantico*** ha come ingresso l'albero sintattico generato dal parser.

Due fasi principali:

#### **Controlli statici** (Static Checking)

(vengono svolti vari controlli sui tipi, dichiarazioni ecc.)

Per l'espressione  $(A+B)*(C+D)$ , ad esempio, l'analizzatore semantico deve determinare quali azioni sono specificate dagli operatori aritmetici di addizione e moltiplicazione.

Quando riconosce il + od il \* invoca allora una *routine semantica* che specifica le azioni da svolgere.

Ad esempio, che gli operandi siano stati dichiarati, abbiano lo stesso tipo ed un valore.

#### **Generazione di una rappresentazione intermedia (IR)**

Spesso la parte di analisi semantica produce anche una forma intermedia di codice sorgente. Ad esempio può produrre il seguente insieme di quadruple:

(+,A,B,T1)  
(+,C,D,T2)  
(\* ,T1,T2,T3)

Od altri tipi di codice intermedio.

Presentiamo un codice intermedio che rimuove dall'albero sintattico alcune delle categorie intermedie e mantiene solo la struttura essenziale (**albero sintattico astratto**).

Tutti i nodi sono token.

Le foglie sono operandi, mentre i nodi intermedi operatori.

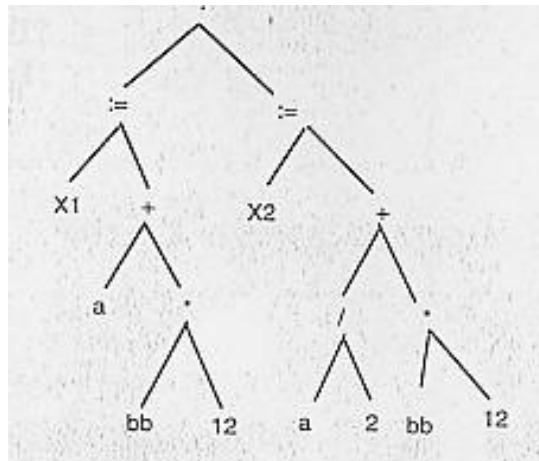
### Esempio:

Dall' albero sintattico precedente relativo a:

X1: =a+bb\* 12;

X2: =a/2+bb\*12;

genera il seguente albero sintattico astratto:



Spesso a valle dell'analizzatore semantico ci può essere un ottimizzatore del codice intermedio.

## Propagazione di costanti

Si consideri il seguente codice:

X: =3;

A: =B+X;

Si può ottimizzare come:

X: =3;

A: =B+3;

evitando un accesso alla memoria.

## Eliminazione di sotto-espressioni comuni

A: =B\*C;

D: =B\*C;

Si trasforma in:

T: =B\*C;

A: =T;

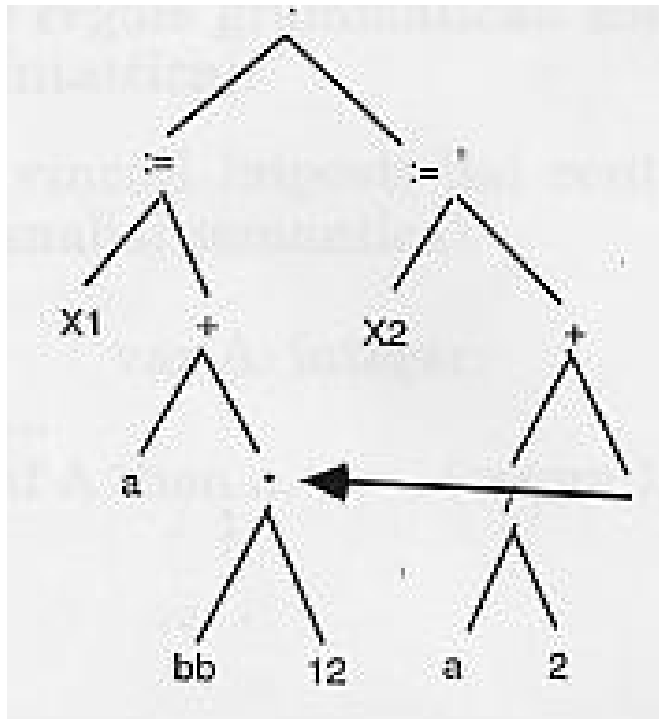
D: =T;

Nel caso di:

X1: =a+bb\* 12;

X2: =a/2+bb\*12;

Otteniamo il seguente albero sintattico astratto ottimizzato (in realtà diventa un grafo):



## IN SOMMARIO:

### Verifica della correttezza sintattica e semantica di un programma:

E' svolta in fase di compilazione.

In particolare verifica che:

- i simboli utilizzati siano legali, cioè appartengano all'alfabeto (analisi lessicale).
- le regole grammaticali siano rispettate (analisi sintattica).
- i vincoli imposti dal contesto siano rispettati (analisi semantica).

**Es:** var A: integer;

...

if A then... (errore!).

L'uscita dell'analizzatore semantico è passata al **generatore di codice** che trasla la forma intermedia in linguaggio assembler o macchina.

C'e' una fase di preparazione prima della generazione del codice oggetto:

- **allocazione della memoria** (può essere allocata staticamente od è uno stack o heap la cui dimensione cambia durante l'esecuzione);

- **allocazione dei registri**. Ovviamente l'accesso ai registri è più rapido che non a locazioni di memoria. I valori acceduti spesso andrebbero messi nei registri.

### **Esempio:**

Nel caso di:

$X1 := a + bb * 12;$

$X2 := a/2 + bb * 12$

potremmo pensare di allocare l'espressione  $bb * 12$  al registro 1, ed una copia del valore di  $a$  al registro 2 assieme al valore  $a/2$ . Le variabili si potrebbero allocare sullo stack con  $a$  al top, e poi, nell'ordine,  $bb$ ,  $X1$ ,  $X2$ . IL registro  $S$  punta al top dello stack.

Segue poi la vera e propria generazione di codice

### **Esempio**

Dalle quadruple precedenti si possono produrre le seguenti istruzioni assembler:

```
LOADA A
LOADB B
STOREA T1
LOADA C
LOADB D
STOREA T2
LOADA T1
LOADB T2
MULT
STOREA T3
```

## Esempio:

Nel caso di:

$X1 := a + bb * 12;$

$X2 := a/2 + bb * 12$

potremmo generare (per una macchina di nostra invenzione) il seguente codice:

PushAddr	X2	Mette l'indirizzo di X2 nello stack
PushAddr	X1	Mette l'indirizzo di X1 nello stack
Push	bb	Mette bb nello stack
Push	a	Mette a nello stack
Load	1(S),R1	Mette bb in R1
Mpy	#12,R1	Mette $bb * 12$ in R1
Load	(S),R2	Mette a in R2
Store	R2,R3	Copia a in R3
Add	R1,R3	Mette $a + b * 12$ in R3
Store	R3,@2(S)	Mette $a + b * 12$ in X1
Div	#2,R2	Mette $a/2$ in R2
Add	R1,R2	Mette $a/2 + bb * 12$ in R2
Store	R2,@3(S)	Mette $a/2 + bb * 12$ in X2

## Nota:

(S), 1(S), 2(S) ecc, significa accedere al contenuto del top dello stack, ad una posizione successiva, due posizioni successive ecc.

@A indica che si vuole accedere alla locazione il cui valore è puntato da A (indirizzamento indiretto).

L'uscita del generatore di codice è passata all'**ottimizzatore di codice** presente nei compilatori più sofisticati.

Ad esempio può ottimizzare il codice precedente come segue:

LOADA	A
LOADB	B
STOREA	T1
LOADA	C
LOADB	D
LOADB	T1
MULT	
STOREA	T3

Esistono infatti sia ottimizzazioni **indipendenti** dalla macchina (ad esempio la rimozione di istruzioni invarianti all'interno di un loop, fuori dal loop) sia **dipendenti** (ad esempio ottimizzazione dell'uso dei registri).



## I passi di un compilatore

Qui abbiamo presentato tutte le fasi in modo separato, ma spesso sono combinate.

Scanner e parser possono essere eseguiti in sequenza uno dopo l'altro, producendo prima tutti i token e poi l'analisi sintattica, oppure lo scanner è chiamato dal parser ogni volta che necessita un nuovo token.

Nel primo caso lo scanner ha esaminato l'intero programma sorgente prima di passare il controllo al parser e quindi ha compiuto un intero **passo** separato.

A volte il parser, l'analizzatore semantico ed il generatore di codice sono combinati in un singolo passo. Alcuni compilatori sono solo ad un passo, altri anche fino a 30!

Abbiamo ignorato altri aspetti importanti della compilazione:

- 1 - Error Detection e Recovery;
- 2 - Le Tabelle dei Simboli prodotte dai vari moduli;
- 3 - La Gestione della Memoria implicata da alcuni costrutti del linguaggio di alto livello.

Le fasi di più semplice progettazione, con un apparato formale ben sviluppato e quindi facilmente automatizzabili sono scanner e parser, mentre maggiore difficoltà si trova nella progettazione di analizzatori semantici, generatori ed ottimizzatori di codice.

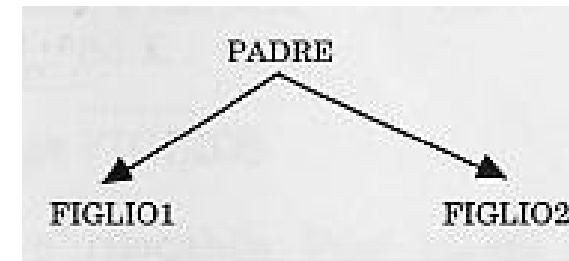
## Linking e Caricamento

Il programma oggetto prodotto dal compilatore contiene una serie di riferimenti esterni (es. riferimenti a programmi di libreria, funzioni).

I riferimenti esterni vengono risolti dal **linker**.

### Esempio:

Struttura del programma



PADRE indirizzi relativi	da 0 a 300
FIGLIO 1	da 0 e 120
FJGLIO2	da 0 a 150

Il linker riceve in ingresso questi tre moduli e genera un unico modulo con riferimento ad indirizzi contigui a partire da un indirizzo simbolico *ind*.

Ogni riferimento a moduli esterni viene sostituito con l'indirizzo così calcolato.

Indirizzo	Contenuto	Commento
Ind	inizio PADRE	
...	.....	
...	salta a ind + 301	Rif. A FIGLIO1
...	.....	
...	salta a ind + 421	Rif. aFIGLIO2
...	.....	
<i>ind</i> + 300	fine PADRE	
<i>ind</i> + 301	inizio FIGLIO1	
...	.....	
...	.....	
<i>ind</i> + 420	fine FIGLIO1	
<i>ind</i> + 421	inizio FIGLIO2	
...	.....	
...	.....	
<i>ind</i> + 570	fine FIGLIO2	

Il programma è rilocabile. Può essere allocato in diverse zone di memoria cambiando *ind* (indirizzamento relativo).

Fase di caricamento compiuta dal **loader** che assegna un valore numerico ad *ind*, trasformando gli indirizzi relativi in assoluti.