

The background is a light blue gradient with several realistic water droplets of various sizes scattered across it. Some droplets are in the top left, some in the bottom right, and others in the center. They have highlights and shadows, giving them a 3D appearance.

FASI DELLA COMPILAZIONE

LINGUAGGI SIMBOLICI

Ogni compilatore utilizza **linguaggi simbolici** definiti da:

- **alfabeto**, insieme dei simboli usati
- **lessico**, insieme delle parole che formano il linguaggio (**vocabolario**)
- **sintassi**, insieme delle regole per la formazione delle frasi (**istruzioni**)
- **semantica**, significato da associare ad ogni parola ed istruzione

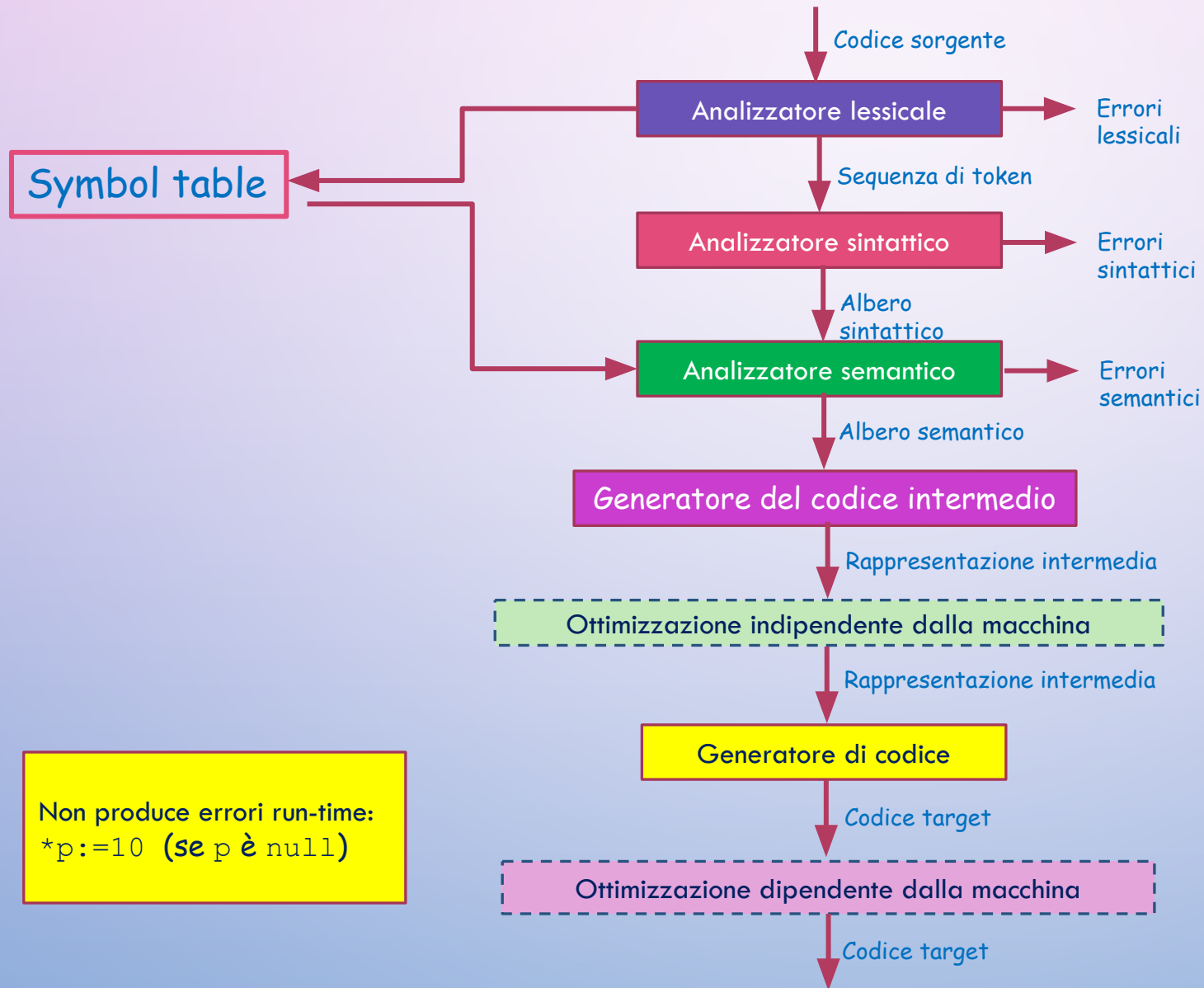
FASI DELLA COMPILAZIONE

- Analisi lessicale
- Analisi sintattica
- Analisi semantica
- Generazione del codice intermedio
- Ottimizzazione
- Generazione del codice target o codice oggetto



Analisi

Sintesi



Errore lessicale: simbolo
non riconosciuto

Errore sintattico:
If x:=3 then...

Errore semantico:
x:=y+s
Con x,y int e
s string

ANALISI E SINTESI

- La prima parte della compilazione, detta **analisi** (Analisi lessicale, Analisi sintattica, Analisi semantica, Generazione del codice intermedio) dipende solo dal tipo di linguaggio che deve essere tradotto viene anche detta "**front end**" del compilatore.
- La seconda parte, detta **sintesi** che consiste delle fasi di ottimizzazione e creazione del codice target insieme alle ottimizzazioni dipendenti dalla macchina viene detta "**back end**" del compilatore.

ANALISI LESSICALE O SCANNING

In questa fase **l'analizzatore lessicale (scanner)** suddivide il codice sorgente in **lessemi** (elementi lessicali o terminali del linguaggio). Per ogni lessema produce un **token**.

I vari lessemi, una volta isolati, vengono memorizzati in una o più tabelle (tabelle dei descrittori o **symbol table**) dove saranno contenute tutte le indicazioni necessarie per identificare e caratterizzare i singoli lessemi (nome, tipo, visibilità,...; nel caso di nomi di funzioni, il numero di parametri, come sono passati, il tipo restituito,...)

Un **token** può essere pensato come una coppia **<nome del token, attributo>**

Il **nome del token** è un nome astratto utile durante l'analisi sintattica, **l'attributo** è il puntatore nella symbol table alla entry contenente i dati di questo specifico token.

COSTRUZIONE DELLA SYMBOL TABLE

Supponiamo di avere il seguente segmento di programma:

Token

```
T_While  
T_LeftParen  
T_Identifier y  
T_Less  
T_Identifier z  
T_RightParen  
T_OpenBrace  
T_Int  
T_Identifier x  
T_Assign  
T_Identifier a  
T_Plus  
T_Identifier b  
T_Semicolon  
T_Identifier y  
T_PlusAssign  
T_Identifier x  
T_Semicolon  
T_CloseBrace
```

```
while (y < z) {  
  int x = a + b;  
  y += x;}
```

Symbol table

Loc mem	Nome simbolo	proprietà
1	y	int,
2	z	int,
3	x	int
4	a	const
5	b	const

LISTA DEI TOKEN

```
while (y < z) {  
  int x = a + b;  
  y+= x;}  
}
```

I token corrispondenti a lessemi della symbol Table sono usualmente indicati con <nome lessema, attributo>, per esempio <T_identifier, y> mentre gli altri (come per esempio +), possono essere indicati semplicemente con <nome lessema> (per esempio <+>)

Token	Lessema
T_while	While
T_leftparent	(
T_identifier	y,z,x,a,b,
T_less	<
T_assign	=
T_openbrace	{
T_int	Int
T_semicolon	;
T_plus	+
....	...

COMPITI DELLO SCANNER

Lo Scanner o analizzatore lessicale ha il compito di «preparare» il programma all'analisi sintattica. In particolare ha i seguenti compiti:

- **Costruzione delle symbol table**
- **Trasformazione del codice sorgente in una forma semplificata,** sostituendo i token ai corrispondenti simboli, in maniera che il codice sia pronto per l'analisi sintattica.

SECONDA FASE: ANALISI SINTATTICA

Token

```
T_While  
T_LeftParen  
T_Identifier y  
T_Less  
T_Identifier z  
T_RightParen  
T_OpenBrace  
T_Int  
T_Identifier x  
T_Assign  
T_Identifier a  
T_Plus  
T_Identifier b  
T_Semicolon  
T_Identifier y  
T_PlusAssign  
T_Identifier x  
T_Semicolon  
T_CloseBrace
```

Symbol table

Analizzatore
sintattico

Insieme di alberi sintattici

ANALISI SINTATTICA (SYNTAX ANALYSIS O PARSING)

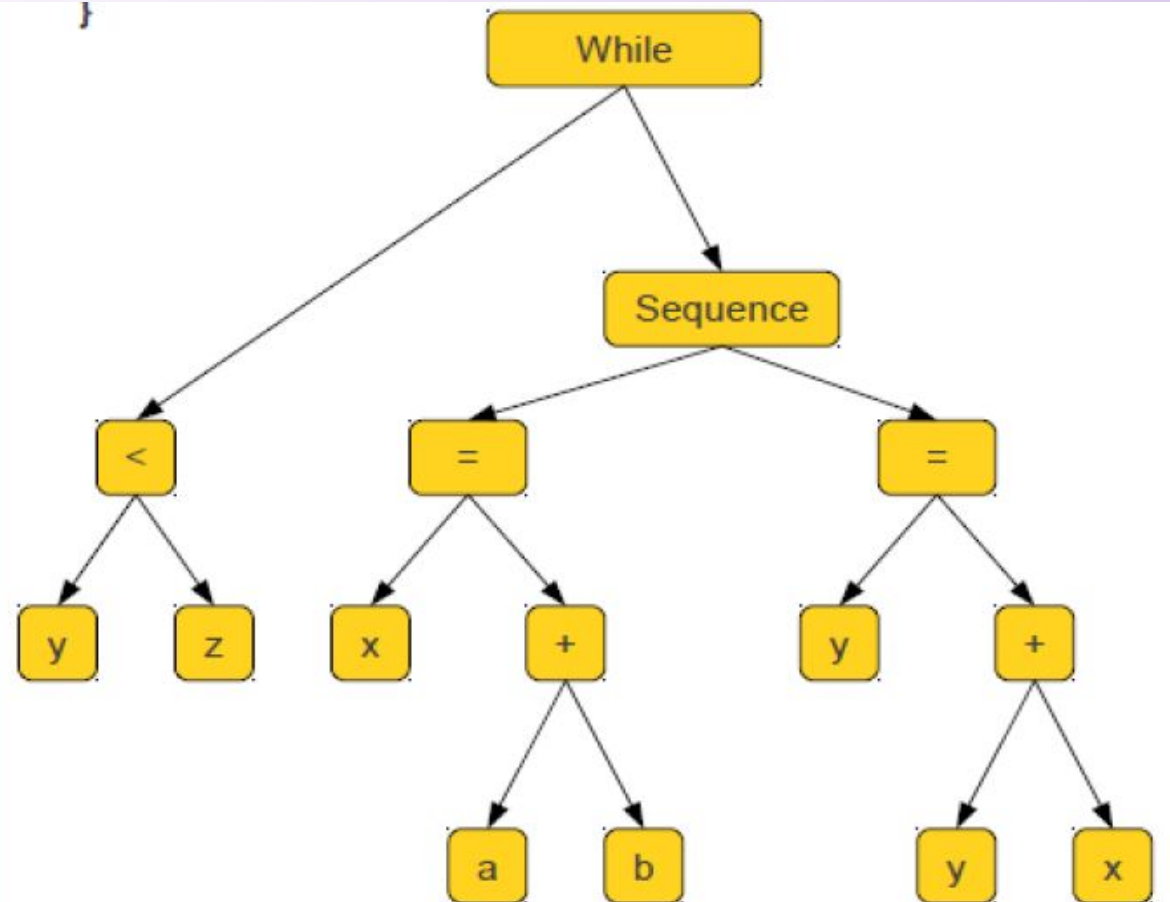
In questa fase l'**analizzatore sintattico (parser)** usa i token (i nomi dei token) per definire la **struttura grammaticale** della sequenza di token, ovvero determina la struttura delle singole istruzioni e controlla se sono rispettate le regole della sintassi del linguaggio.

- I token diventano i **simboli terminali** della grammatica che definisce il linguaggio.
- Il parser adopera il **codice semplificato** e la **symbol table** costruiti nella fase di analisi lessicale e genera per ogni istruzione il corrispondente **albero sintattico (syntax tree)**.
- L'output della fase di analisi sintattica è **l'insieme degli alberi sintattici** che descrivono il programma.

ANALISI SINTATTICA: OUTPUT

```
while (y < z) {  
  int x = a + b;  
  y += x;  
}
```

Albero sintattico



ANALISI SEMANTICA

L'analizzatore semantico utilizza l'**albero sintattico** e le informazioni nella **symbol table** per verificare la consistenza del programma sorgente rispetto alla definizione del linguaggio. L'analisi semantica consiste nell'interpretazione del "significato" delle strutture prodotte nella fase precedente controllando che esse siano legali e significative, le variabili coinvolte siano dichiarate e definite, che i tipi siano corretti, che gli operatori siano usati correttamente,

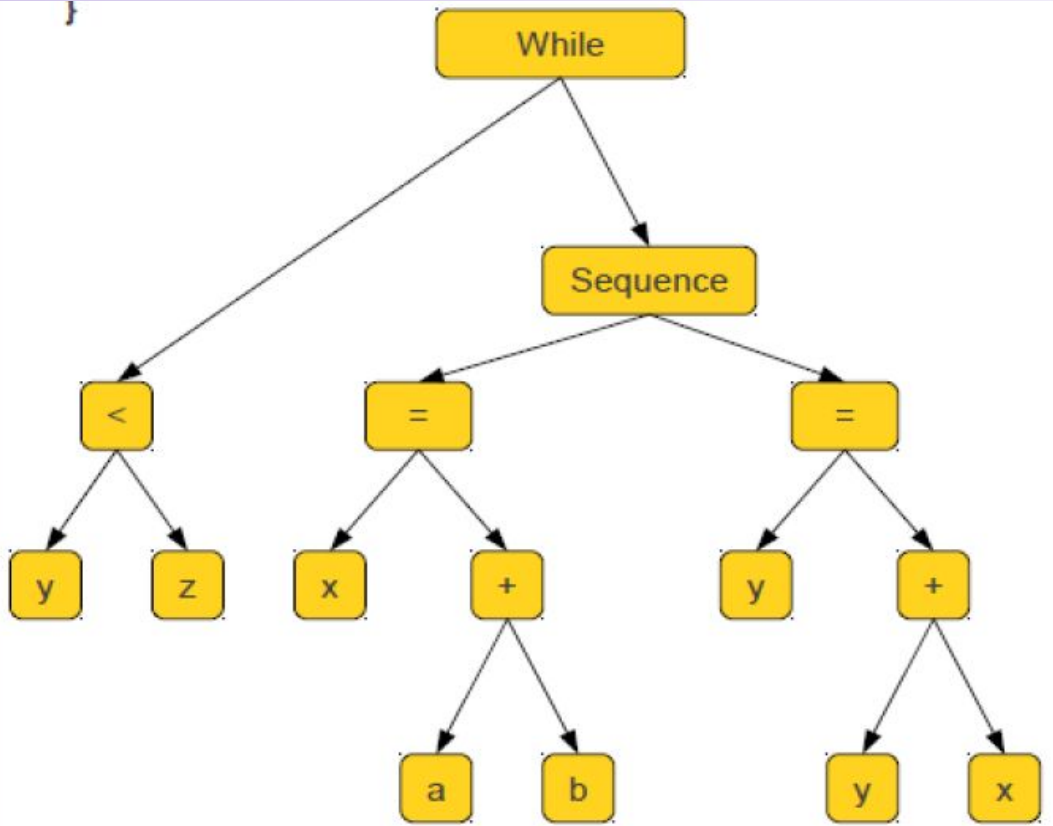
Una fase fondamentale è il **controllo dei tipi** o **type checking** che verifica se gli operandi sono adeguati rispetto all'operatore. Per **Esempio**:

- Controlla che l'indice di un array sia un intero
- Applica le regole di visibilità degli identificatori

Il linguaggio potrebbe permettere anche alcune conversioni di tipo dette coercizioni. Per esempio per svolgere un'operazione di somma tra intero e float, converte l'intero in float

L'output di questa fase è l'**albero sintattico decorato o annotato**

ANALISI SEMANTICA



Analizzatore
semantico

Albero decorato

ANALISI SEMANTICA

Usa il **syntax tree** e la **symbol table** del programma sorgente per controllarne la consistenza semantica con le definizioni del linguaggio. Si controlla, per esempio, che le variabili coinvolte siano dichiarate e definite, che i tipi siano corretti, che gli operatori siano usati correttamente,

Una parte importante dell'analisi semantica è il **type checking** dove il compilatore controlla che ogni operatore abbia gli operandi giusti. Per esempio:

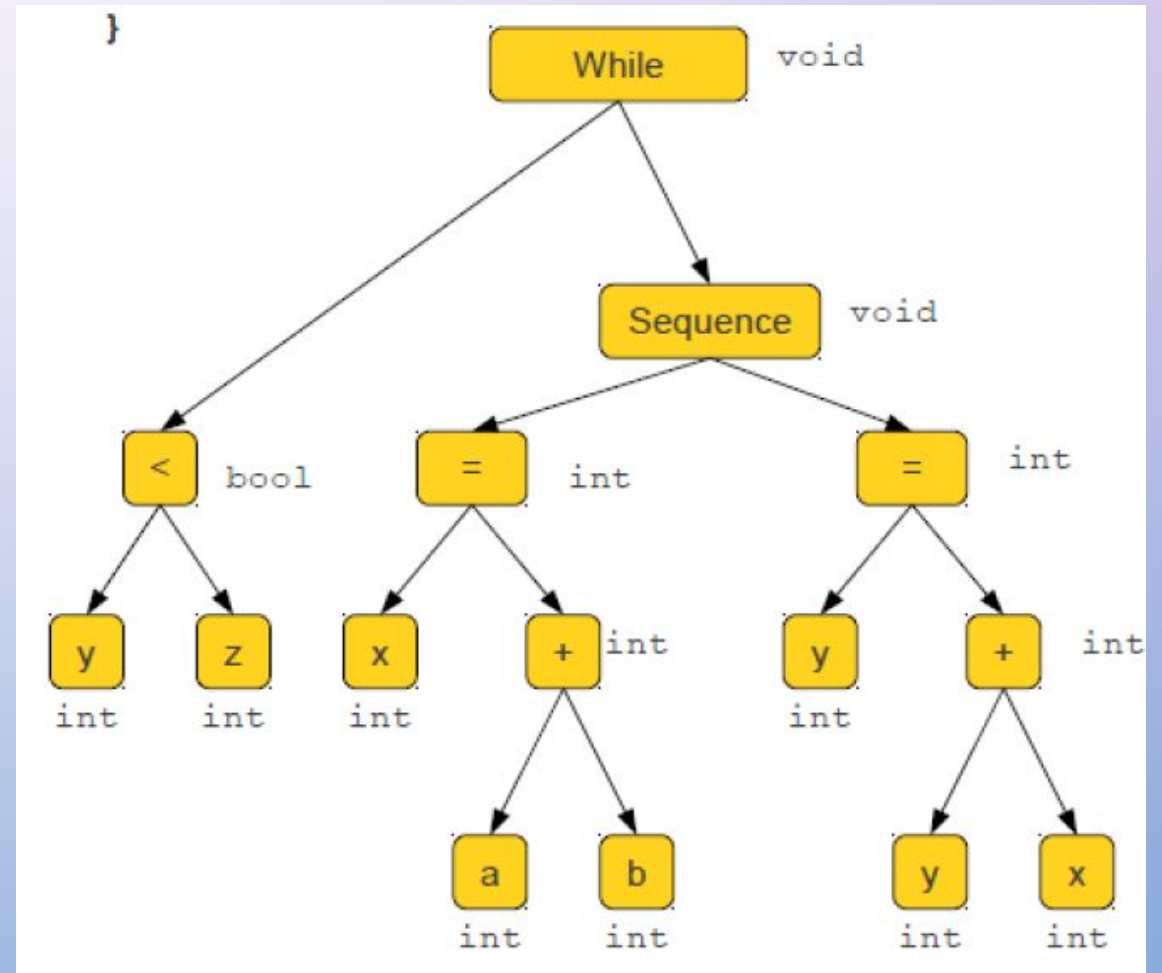
- Controlla che l'indice di un array sia un intero
- Applica le regole di visibilità degli identificatori
- Applica le conversioni di tipo nel caso di operatori che possono essere applicati ad operandi di diverso tipo (coercions)

Questa fase produce **l'albero sintattico decorato**

OUTPUT DELL'ANALIZZATORE SEMANTICO

Analizzatore
semantico

output



SEMANTICA STATICA E SEMANTICA DINAMICA

Semantica statica: indipendente dai dati su cui opera il programma sorgente.

```
Var i : real;
```

```
A : array [1..100] of integer;
```

```
.....
```

```
i:=3.5;       Errore semantico (semantica statica)
```


```
A[i]:=3;
```

Semantica dinamica: dipendente dai dati su cui opera il programma sorgente.

```
Var i : real;
```

```
A : array [1..100] of integer;
```

```
.....
```

```
Read(i);       Se viene inserito da tastiera il valore 3.5 si verifica un  
errore semantico (semantica dinamica)
```

```
A[i]:=3;
```

L'analizzatore semantico si occupa della **semantica statica**, mentre la **semantica dinamica** spetta all'interprete o al **supporto esecutivo**.

GENERAZIONE DEL CODICE INTERMEDIO

In questa fase si provvede alla generazione del codice intermedio che solitamente è una rappresentazione di basso livello (più vicina alla macchina).

Esistono diversi modelli usati per la generazione del codice intermedio (execution model). In ogni caso deve avere due proprietà importanti: essere facile da produrre e facile da tradurre in codice target.

Uno dei più usati è il **three-address-code** in cui tutte le istruzioni hanno la forma di istruzioni in assembly con tre operandi per istruzione:

Id := id1 operator id2

TRADUZIONE IN CODICE INTERMEDIO

```
while (y < z) {  
  int x = a + b;  
  y += x;  
}
```



```
Loop: x = a + b  
      y = x + y  
      _t1 = y < z  
      If _t1 goto Loop
```

In 3AC (three address code):

- ogni istruzione ha al più un operatore sul lato destro.
- Le istruzioni fissano l'ordine con cui vengono eseguite le operazioni
- vengono generati nomi temporanei per memorizzare i valori intermedi
- alcune istruzioni possono avere meno di 3 operandi

FASE DI OTTIMIZZAZIONE DEL CODICE

In questa fase il codice intermedio viene analizzato e trasformato in codice ad esso equivalente ma più efficiente. Questa fase può essere molto complessa e lenta.

La maggior parte dei compilatori permette di disattivare l'ottimizzazione per velocizzare la traduzione; altri non hanno ottimizzazione.

In questa fase si tratta di **ottimizzazioni indipendenti dalla macchina.**

Esistono anche ottimizzazioni dipendenti alla macchina operate dal post-ottimizzatore.

Il problema della **generazione del codice ottimale è indecidibile.**

FASE DI OTTIMIZZAZIONE

Esempio

```
while (y < z) {  
  int x = a + b;  
  y += x;  
}
```

Codice iniziale

```
Loop: x = a + b  
      y = x + y  
      _t1 = y < z  
      If _t1 goto Loop
```

Codice intermedio

```
      x = a + b  
Loop: y = x + y  
      _t1 = y < z  
      If _t1 goto Loop
```

Codice intermedio ottimizzato

OTTIMIZZAZIONI INDIPENDENTI DALLA MACCHINA

- Calcolare una sola volta le espressioni ripetute, purché le variabili non mutino di valore tra le ripetizioni:

- $a[i*k] := a[i*k] + 1$

($i*k$ si può calcolare una sola volta)

- Eliminare le moltiplicazioni per 1 e le somme di 0.
- Portare fuori da un ciclo le operazioni che non dipendono dal valore dell'indice:

```
For i:=1 to N do  
  Begin  
    ...  
    j:=r/s;  
    ...  
  End;
```

($j:=r/s$ si può portare fuori)

OTTIMIZZAZIONE

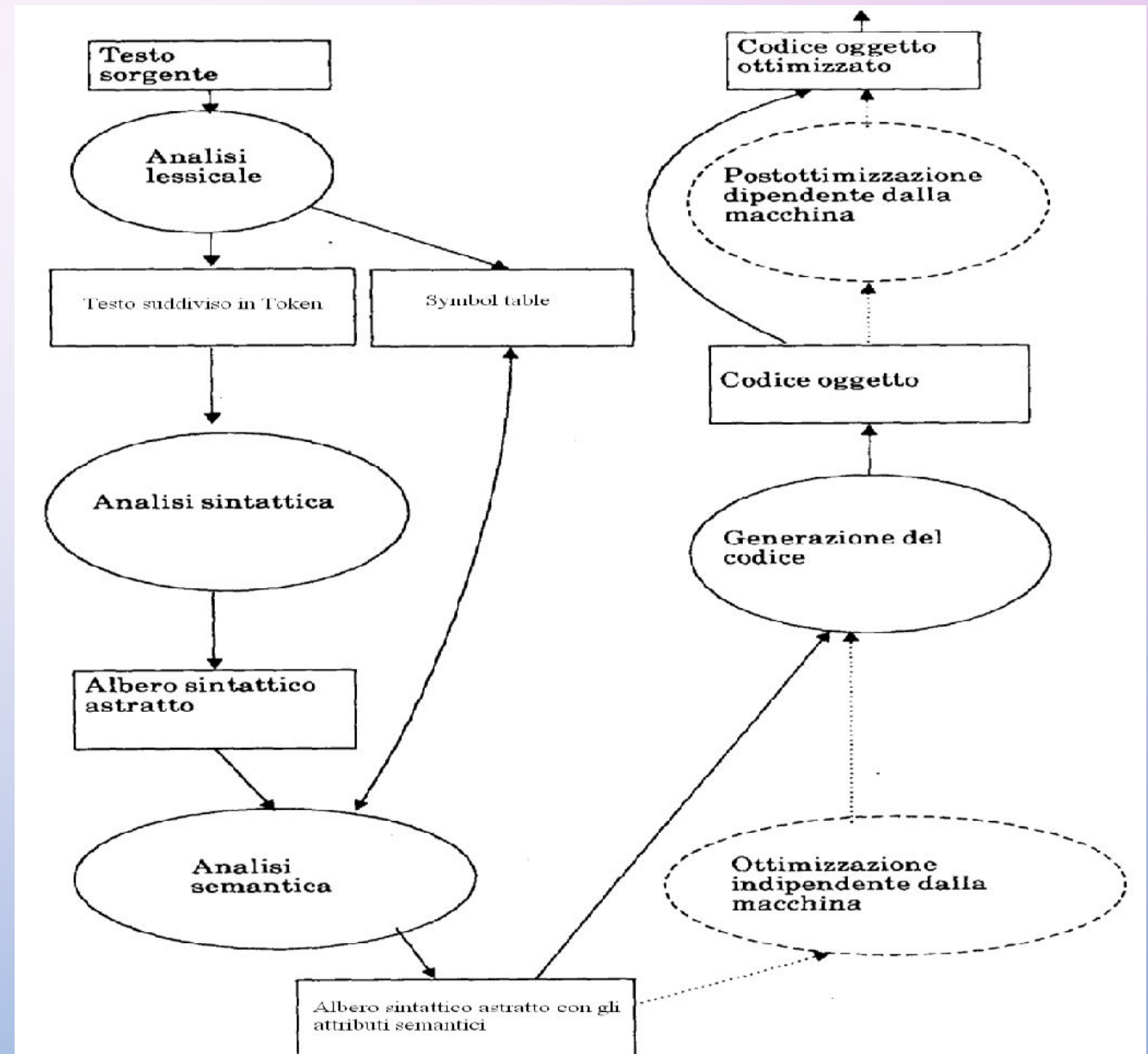
ESEMPIO

```
_t1 = b * c  
_t2 = _t1 + 0  
_t3 = b * c  
_t4 = _t2 + _t3  
a = _t4
```

```
_t1 = b * c  
_t2 = _t1 + _t1  
a = _t2
```

Il codice a sinistra e il codice a destra della figura hanno lo stesso risultato, ma il secondo è più semplice perché vengono eliminate le somme di valori nulli e le rivalutazioni delle stesse espressioni

Il processo di compilazione
è riassunto nella figura a
fianco



GENERAZIONE DEL CODICE TARGET

In questa fase si provvede alla traduzione del codice intermedio eventualmente ottimizzato nel linguaggio della "target machine".

Generalmente il codice oggetto è scritto in **codice macchina o assembly**

Se il linguaggio target è il codice-macchina, allora per ogni variabile del programma sono assegnati registri e locazioni di memoria.

NOTA: Le decisioni sull'allocazione della memoria vengono prese o durante la generazione del codice intermedio o durante la generazione del codice oggetto.

GENERAZIONE DEL CODICE OGGETTO

Esempio

```
while (y < z) {  
  int x = a + b;  
  y += x;  
}
```

Codice iniziale

```
Loop: x = a + b  
      y = x + y  
      _t1 = y < z  
      If _t1 goto Loop
```

Codice intermedio

```
      x = a + b  
Loop: y = x + y  
      _t1 = y < z  
      If _t1 goto Loop
```

Codice intermedio
ottimizzato

```
      add $1, $2, $3  
Loop: add $4, $1, $4  
      slt $6, $4, $5  
      beq $6, loop
```

Codice oggetto

OTTIMIZZAZIONE

Esempio

```
while (y < z) {  
  int x = a + b;  
  y += x;  
}
```

Codice iniziale

```
Loop: x = a + b  
      y = x + y  
      _t1 = y < z  
      If _t1 goto Loop
```

Codice intermedio

```
      x = a + b  
Loop: y = x + y  
      _t1 = y < z  
      If _t1 goto Loop
```

Codice intermedio
ottimizzato

```
      add $1, $2, $3  
Loop: add $4, $1, $4  
      slt $6, $4, $5  
      beq $6, loop
```

Codice oggetto

```
      add $1, $2, $3  
Loop: add $4, $1, $4  
      blt $1, $5, loop
```

Codice oggetto
ottimizzato

GESTIONE DEGLI ERRORI

Durante tutte le fasi della compilazione si possono riscontrare degli errori. Essi possono essere tali da bloccare o meno il processo.

Spesso accade che un errore ne provochi in conseguenza tanti altri.

Conviene non preoccuparsi se alla prima compilazione l'elenco degli errori è molto lungo.

Il problema di trovare tutti gli errori di un programma è indecidibile

FASI E PASSATE

Pur avendo descritto separatamente e sequenzialmente le varie fasi della compilazione, in realtà queste possono essere svolte con modalità e in tempi diversi.

Inoltre la compilazione può avvenire in una o più “passate”.

Ad esempio, le fasi del front-end possono essere raggruppate in una sola passata, l'ottimizzazione del codice intermedio in un'altra passata e poi un'ultima passata per il back-end.

In generale, una passata può coincidere con:

- una singola fase o parte di essa;
- più fasi;
- parti di più fasi;

QUANTE PASSATE?

Un compilatore ad una sola passata è normalmente più veloce.

Alcuni linguaggi sono stati progettati perché potessero essere compilati in una sola passata (Pascal).

In altri casi le possibilità offerte dai linguaggi di programmazione impediscono la compilazione in una sola passata (esempio: linguaggi che prevedono una **mutua ricorsione** o linguaggi in cui le variabili possono essere dichiarate in qualunque parte del programma)

Uno svantaggio della compilazione in singola passata è che non è possibile effettuare delle ottimizzazioni "sofisticate" necessarie per generare codice di alta qualità. Può essere difficile, infatti, valutare il numero di volte che un'espressione deve essere analizzata per produrre una sua buona ottimizzazione.

REALIZZARE UN COMPILATORE

Il primo compilatore fu scritto in linguaggio assembly (e non c'erano altre alternative). Vale il seguente principio:

Se si dispone di un compilatore C_1 per un determinato linguaggio L_1 è possibile utilizzare il linguaggio L_1 per implementare un compilatore C_2 per un altro linguaggio L_2 .

E' anche possibile scrivere un compilatore utilizzando una versione minimale dello stesso linguaggio (**bootstrapping**).

Il compilatore può girare sulla stessa macchina sulla quale girerà il codice target oppure su una macchina diversa (**cross compiler**)

REALIZZARE AUTOMATICAMENTE UN COMPILATORE

Esistono strumenti software per generare automaticamente parti di un compilatore.

Ad esempio:

- **flex** generatore di analizzatori lessicali
- **bison** generatore di analizzatori sintattici

EVOLUZIONE DELLA COMPILER TECHNOLOGY

La rapida evoluzione delle architetture propone continue sfide ai creatori di compilatori:

- **Parallelismo a livello di istruzioni** (intel IA64). I compilatori possono ordinare le istruzioni in modo da rendere tale parallelismo più efficiente.
- **Parallelismo a livello di processori**. I compilatori possono generare codici paralleli per multiprocessori a partire da un codice sequenziale.
- **Gerarchie di memorie**, al livello di velocità di accesso e di spazio. Il compilatore può cambiare l'ordine delle istruzioni che accedono ai dati

Nello sviluppo delle architetture moderne i compilatori sono sviluppati nella fase di design dell'architettura stessa.