COMPILATORI

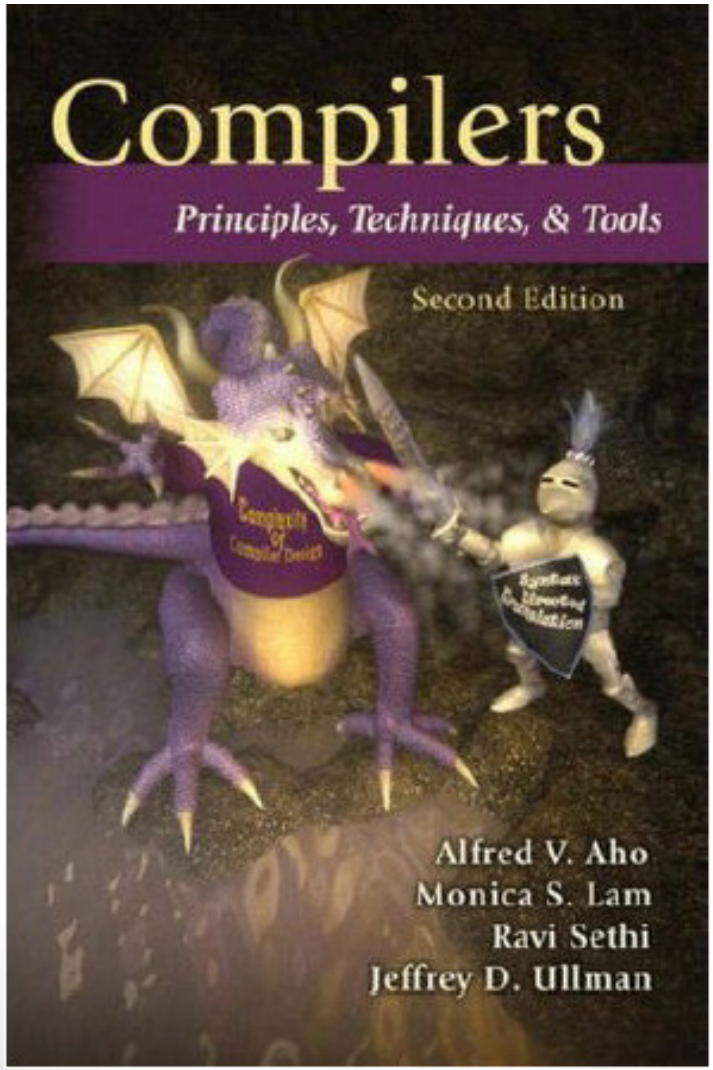
Introduzione al corso

Lunedì 1 Ottobre 2012

Informazioni generali sul corso

- CFU: **6** ORE: **48**
- Docente:
  - **Prof. Marinella Sciortino**
- Orario delle lezioni (laboratorio via Ingrassia)
  - Lunedì 9:30 – 11:30
  - Giovedì 9:30 – 11:30
- Modalità di svolgimento degli esami
  - Discussione relazione su progetto assegnato almeno 7 giorni prima (aspetti pratici)
  - Prova orale (aspetti teorici)
- Possibili prove in itinere
- Il materiale del corso sarà disponibile sul sito del corso di laurea.

Testo di riferimento



- **A. Aho, M. Lam, R. Sethi, J. Ullman**
  Compilers, Principles, Techniques & Tools
  Addison Wesley

- **A. Aho, M. Lam, R. Sethi, J. Ullman**
  COMPILATORI. Principi, Tecniche e strumenti
  Addison Wesley

Altri Testi consigliati

- Giorgio Bruno
  Linguaggi Formali e Compilatori
  Utet Libreria

- Stefano Crespi Reghizzi
  Linguaggi Formali e Compilazione
  Pitagora Editrice

- …

- Manuali di Flex e Bison

In quale ambito si inseriscono gli argomenti del corso?

Contesto:
Linguaggi di Programmazione

- I linguaggi di programmazione sono gli strumenti di base dei programmatori.
- Tutti i software che girano su tutti i computer sono scritti in un qualche linguaggio di programmazione.
- Prima che un programma possa “girare” ovvero essere mandato in esecuzione in un computer deve essere prima trasformato (tradotto) in sequenze di istruzioni elementari eseguibili da un computer.
- Si usano spesso, a volte “inconsapevolmente” nei processori di linguaggi di programmazione.

Processore di un linguaggio di programmazione

- E' un qualsiasi sistema che manipola programmi espressi in un particolare linguaggio. Consente di scrivere i programmi e prepararli per l'esecuzione.
- Può incorporare una varietà di sistemi tra cui:
  - **Editors.** Consente la digitazione, modifica e salvataggio in un file.
  - **Traduttori e Compilatori.** Un traduttore traduce un testo scritto in un linguaggio in un altro. In particolare, un compilatore traduce programmi da linguaggio ad alto livello a linguaggio a basso livello, quindi prepara per l'esecuzione su una macchina. Prima di effettuare la traduzione un compilatore controlla la presenza di errori sintattici e contestuali.
  - **Interpreti.** Un interprete prende un programma espresso in un particolare linguaggio e lo esegue immediatamente. Questa modalità di esecuzione, che omette la fase di compilazione, è preferibile in ambienti interattivi (ad es. molti linguaggi di interrogazione dei database sono interpretati)

Processori come software separati

- La filosofia dei “tool separati” è ben esemplificata dal SO Unix.
- Ad Esempio, un utente UNIX che sviluppa un’applicazione sul gioco degli scacchi in Java usando SUN JDK.
  - Usa l’editor vi

    ```
    vi Chess.java
    ```
  - Invoca il compilatore javac

    ```
    javac Chess.java
    ```
  - Chiama l’interprete java sul codice in bytecode Chess.class

    ```
    java Chess
    ```

Processori o ambienti di sviluppo integrati (IDE)

- Esistono vari IDE. Per Java alcuni esempi sono Eclipse, NetBeans, Borland Jbuilder, Per C alcuni esempi sono Visual Studio, DEVC++, …
- L'utente può aprire, editare, compilare ed eseguire il programma.
- L'editor consente di distinguere tra variabili, parole chiave, commenti
- Il compilatore è integrato con l'editor, nel caso di errori di compilazione rimanda alla frase contenente presumibilmente l'errore
- Se ci sono errori di esecuzione, si rimanda nell'editor alla frase del codice oggetto in cui è fallita l'esecuzione

Obiettivi principali del corso

1. Illustrare la struttura, le tecniche di costruzione e il funzionamento di un moderno compilatore

2. Usare le tecniche per la realizzazione di compilatori mediante l'uso di strumenti automatici:
   - Flex (Fast Lexical analyzer generator)
   - Bison (Parser generator)

3. Applicare principi e tecniche per la progettazione dei compilatori in vari contesti legati all'analisi di testi.

Prerequisiti indispensabili !!!

- Lo studio dei compilatori richiede particolari conoscenze che coinvolgono:
  - Linguaggi di programmazione (in particolare il linguaggio C)
  - Informatica Teorica (linguaggi, automi, espressioni regolari e grammatiche)
  - Algoritmi (liste, pile, alberi, grafi, tabelle hash)

Perché e quando nascono i compilatori?

Alcune notizie storiche

Compilatori e Linguaggi

- L'interesse verso i compilatori è profondamente legato all'evoluzione dei linguaggi di programmazione.
- Esistono linguaggi a vari livelli di astrazione. I compilatori includono tecniche per sopperire alle inefficienze introdotte da astrazioni sempre più ad alto livello.

Linguaggi macchina: linguaggi a basso livello della I generazione

- Il linguaggio macchina è il linguaggio nativo del computer su cui un certo programma viene eseguito. Guida l'esecuzione della macchina.
- Esso consiste di sequenze di bit che sono interpretate (eseguite passo dopo passo) da un meccanismo interno al computer. Può essere espresso in binario o in esadecimale.
- Quando il primo computer elettronico apparve (nel 1945, modello di John von Neumann), era possibile usare solo il linguaggio macchina.
  Le operazioni realizzabili erano dette di basso livello erano strettamente legate al tipo di macchina e consistevano in:
  - Spostare dati da una locazione all'altra;
  - Confrontare due valori;
  - Sommare il contenuto di 2 registri;
  - …

Ad esempio sul processore Intel 80x86,
il codice ***C7 06 0000 0002***
dice alla CPU di inserire 2 nella locazione 0000.

Linguaggi assembly: linguaggi a basso livello della II generazione

- I **linguaggi assembly** sono nati negli anni ’50.
- Inizialmente erano soltanto rappresentazioni mnemoniche di istruzioni in linguaggio macchina, ovvero abbreviazioni o parole che rendevano più facile ricordare le sequenze di bit. Dopo, vennero aggiunte anche le istruzioni macro.

  *Ad esempio il codice precedente potrebbe diventare **mov x,2**.*
- Un programma in linguaggio assembly è tradotto in linguaggio macchina da un programma chiamato **assembler** che traduce una alla volta le istruzioni in assembly in codice macchina.

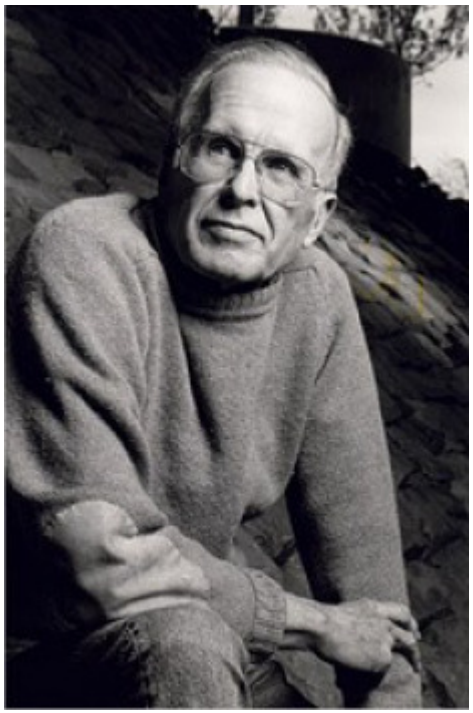
L'evoluzione dei linguaggi di programmazione: high-level languages

- I software per i primi computer erano scritti in linguaggio assembly. L'esigenza dei linguaggi ad alto livello (meno dipendenti cioè dalla macchina) nacque divenne prioritario il vantaggio di riusare software su diversi tipi di macchine.

  *L'istruzione della slide precedente potrebbe diventare **x=2**.*

Linguaggi ad alto livello e compilatori.

- Nascono i linguaggi ad alto livello e il termine ‘’Compilatore’’



- Il termine **compilatore** fu coniato da Grace Murray Hopper nel 1950. E’ stata una matematica, informatica e militare statunitense che ebbe un ruolo fondamentale nello sviluppo del COBOL.
- Il primo compilatore reale fu un compilatore FORTRAN sviluppato alla fine degli anni cinquanta da un team guidato da John Backus.

I primi linguaggi ad alto livello

- Nel 1957 venne introdotto il FORTRAN per il calcolo scientifico da John Backus dell'IBM, da cui derivò successivamente il BASIC (1964). Tali linguaggi prevedono :
  - istruzione IF
  - cicli WHILE e FOR
  - istruzioni CASE e SWITCH

- Il COBOL nacque nel 1961 per la gestione degli archivi commerciali, il LISP nel 1959 per il calcolo simbolico, l'ALGOL nel 1960.
- Nel 1967 nacque il SIMULA, che introdusse per primo il concetto (allora appena abbozzato) di oggetto software.

Nascono importanti linguaggi ad alto livello

- Nel 1970 Niklaus Wirth pubblica il Pascal, il primo linguaggio strutturato, a scopo didattico; nel 1972 dal BCPL nascono prima il B (rapidamente dimenticato) da cui Dennis Ritchie sviluppò il C, che invece fu fin dall'inizio un grande successo.
  Insieme a FORTRAN, ALGOL e COBOL costituiscono esempi del **paradigma di programmazione imperativo**
- Nel 1983 vede la luce Smalltalk, il primo linguaggio realmente e completamente ad oggetti, che si ispira al Simula e al Lisp. Esempi di linguaggi object-oriented odierni sono Eiffel, C++ nel 1986, e successivamente Delphi, JAVA nel 1995.
- In media un'istruzione in linguaggio ad alto livello corrisponde a 10 istruzioni in assembly.

Nota: Contestualmente alla nascita dei primi compilatori Noam Chomsky cominciò il suo studio dei linguaggi naturali e alla classificazione dei linguaggi in base alla potenza di calcolo delle grammatiche.

Nascono i linguaggi specializzati

- Sono i linguaggi progettati per applicazioni specifiche, come per esempio lo sviluppo di applicazioni commerciali.
- Hanno come obiettivo quello di ridurre al minimo gli sforzi per la programmazione, il tempo per lo sviluppo di un software e il relativo costo.
- Ad esempio:
  - SQL, per query in database
  - Postscript, per generare report
  - Mathematica, Matlab, per la manipolazione e l'analisi di dati
  - …

Nascono altri paradigmi, per esempio il paradigma logico

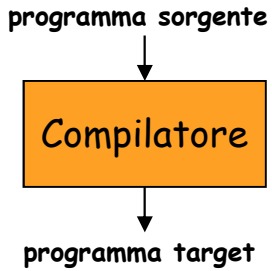
- Si risolve un problema usando i vincoli dati al problema stesso piuttosto che usare un l'algoritmo scritto da un programmatore.
- Usati principalmente in intelligenza artificiale, istruiscono il computer per trovare la soluzione.
- Ad esempio:
  - Linguaggi basati su linguaggi logici come Prolog, OPS5
  - Mercury
  - …

Evoluzione dei linguaggi e compilatori

- L'evoluzione dei linguaggi di programmazione è profondamente in relazione con quella dei compilatori.
- La progettazione di un compilatore richiede enormi sforzi legati alla scelta dei modelli da utilizzare e delle implementazioni. ☹
- Un compilatore deve tradurre in modo corretto un insieme potenzialmente infinito di programmi scritti in linguaggio sorgente. Il codice prodotto deve essere il più possibile efficiente, sia da un punto di vista del tempo di esecuzione che della memoria utilizzata. 😐
- La maggior parte degli utenti vedono un compilatore come una "black box". I compilatori consentono ai programmatori di ignorare i dettagli machine-dependent della programmazione ☺

Compilatore

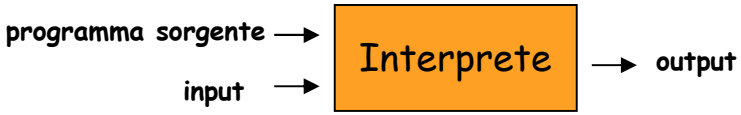
- Un compilatore è un programma capace di leggere un programma in un linguaggio (linguaggio sorgente) e tradurlo in un programma equivalente in un altro linguaggio (linguaggio destinazione o target).



- Un ruolo importante del compilatore consiste nel riportare gli eventuali errori nel programma sorgente incontrati durante il processo di traduzione.
- Se il programma target è programma in un linguaggio-macchina eseguibile, allora esso può essere chiamato dall'utente per processare un input e produrre un output.

Alternativa al compilatore: Interprete

- Invece di produrre il programma target come traduzione, un interprete esegue direttamente le operazioni specificate nel programma sorgente su input forniti dall'utente.



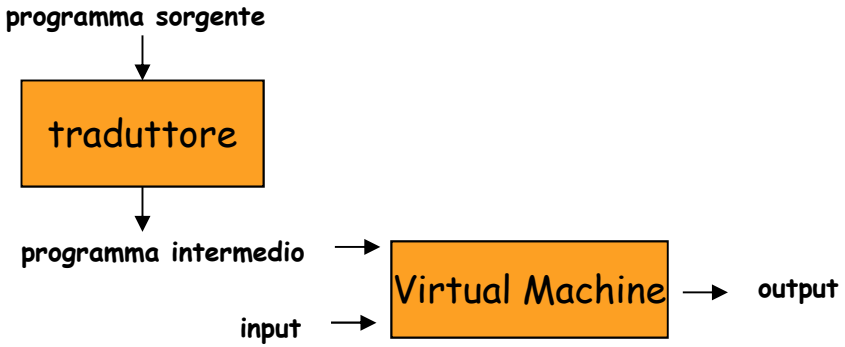
- Il programma target prodotto da un compilatore è solitamente 10 volte più veloce di quello prodotto da un interprete.
- Un interprete comunque può meglio eseguire una diagnostica degli errori poiché esegue il programma sorgente statement dopo statement.

Esempi di linguaggi interpretati

- Javascript
- Perl
- PHP
- Python
- …

Compilatori Ibridi

- Esistono anche compilatori ibridi che combinano compilazione e interpretazione. Ad esempio, un programma sorgente in JAVA viene prima compilato e trasformato in linguaggio di bytecode. I bytecode vengono poi interpretati o compilati da una virtual machine.



- Alcuni compilatori JAVA, detti compilatori just in time, traducono il bytecode in linguaggio macchina immediatamente prima di doverlo eseguire.

Macchina virtuale

- E’ un software che crea un ambiente virtuale in cui l'utente può eseguire alcune applicazioni senza tener conto del sistema operativo sottostante.

- Dal momento che esistono macchine virtuali scritte per diverse piattaforme, il programma compilato può "girare" su qualsiasi piattaforma su cui "gira" la macchina virtuale ("Write once, run everywhere").

- Progenitore delle odierne VM è la “macchina p”, cioè il calcolatore astratto per cui venivano (e vengono tuttora) compilati i programmi in Pascal nelle prime fasi della compilazione (producendo il cosiddetto p-code)

Esempi di codice per macchina virtuale

- ***p-code*** prodotto dal Pascal p-compiler per una ***Virtual Stack Machine***
- ***bytecode*** prodotto dai compilatori Java per una ***Virtual Java Machine*** progettata da Sun Microsystems.
- ***Common Intermediate Language (CIL)*** della piattaforma .NET eseguito dal ***Common Language Runtime (CLR)*** la macchina virtuale .NET progettata da Microsoft per funzionare con qualsiasi sistema operativo.

Coma lavora la Virtual Machine?

- I programmi applicativi vengono scritti in un linguaggio che viene compilato per questo calcolatore immaginario (cioè tradotti nelle sue istruzioni native) e, una volta compilati, vengono eseguiti sulla macchina virtuale software, che può agire o come interprete o come compilatore "al volo" (compilazione just in time: traducono il codice intermedio in linguaggio macchina immediatamente prima che venga processato l'input).

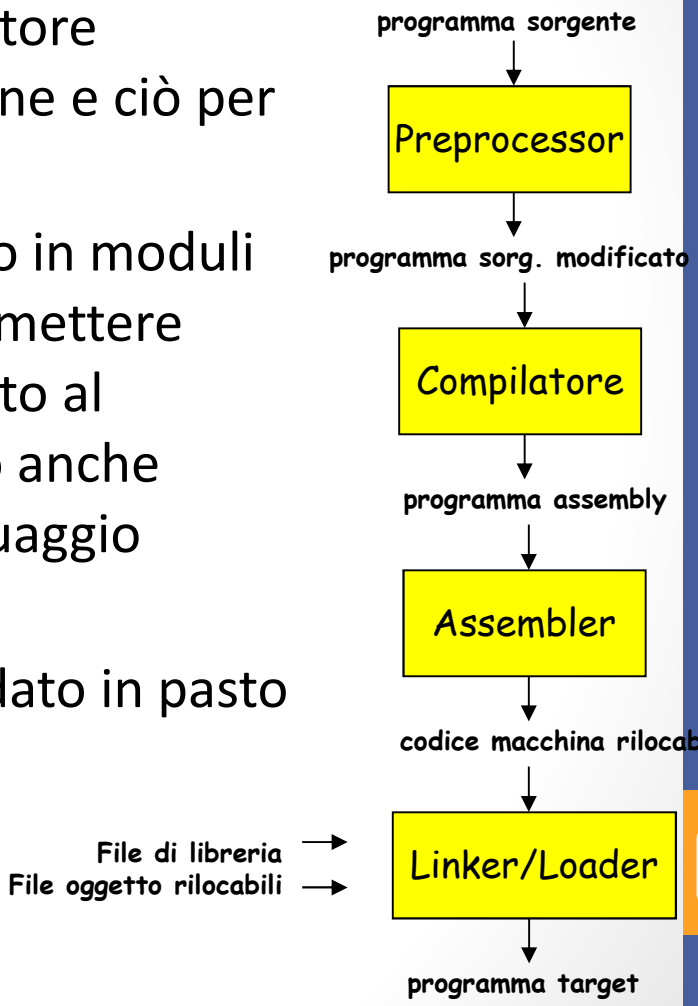
- Le più recenti implementazioni di virtual machine incorporano un JIT compiler:
  - La macchina virtuale Java HotSpot di Sun Microsystem
  - La macchina virtuale Common Language Runtime di Microsoft per la piattaforma .NET
  - La macchina virtuale Parrot per il linguaggio Perl
  - …

Struttura di un compilatore

Le fasi della compilazione

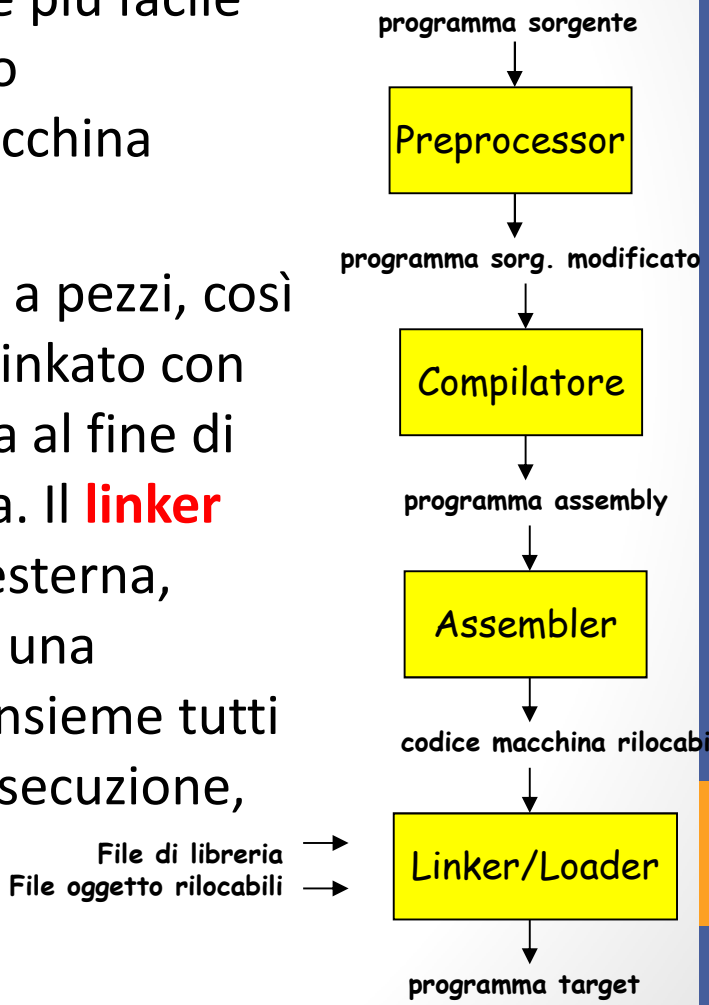
Linguaggi compilati: dal sorgente all'eseguibile

- Il programma target prodotto dal compilatore normalmente non è pronto per l'esecuzione e ciò per diverse ragioni.
- Un programma sorgente può essere diviso in moduli memorizzati in file separati. Il compito di mettere assieme il codice sorgente è a volte affidato al **preprocessor** o **precompilatore**. Esso può anche espandere le macro in statement del linguaggio sorgente.
- Il programma sorgente modificato viene dato in pasto al **compilatore**.



Linguaggi compilati: dal sorgente all'eseguibile

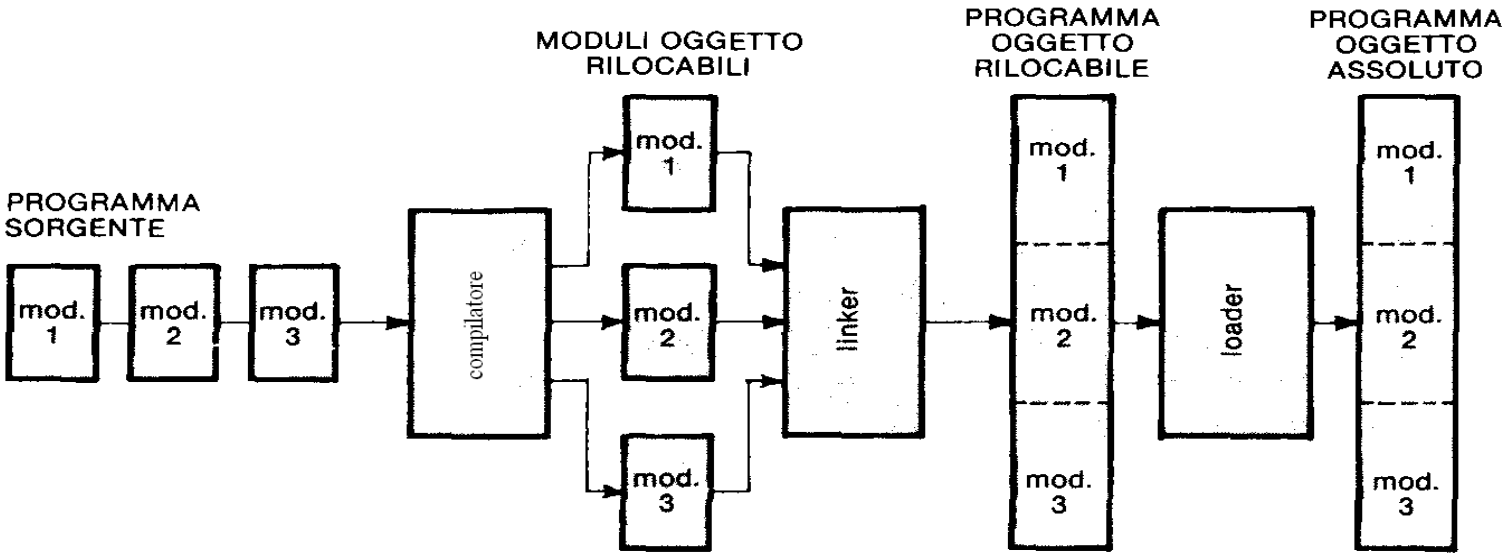
- Il compilatore può produrre un codice in linguaggio assembly che è più facile da produrre ed è più facile fare il debug. Allora il file viene processato dall'**assembler** che produce un codice macchina rilocabile.
- I programmi lunghi sono spesso compilati a pezzi, così il codice macchina rilocabile deve essere linkato con altri file in codice rilocabile e file di libreria al fine di produrre un codice che giri sulla macchina. Il **linker** assegna e risolve gli indirizzi di memoria esterna, poiché un codice in un file può riferirsi ad una locazione in un altro file. Il **loader** mette insieme tutti i file oggetto eseguibili in memoria per l'esecuzione, calcolandone gli indirizzi fisici



Il preprocessore

- In alcuni casi il codice sorgente subisce una trasformazione (***precompilazione***) prima di essere compilato.
- La precompilazione può essere definita come la trasformazione del codice sorgente in un altro.
- Esempi:
  - Traduzione del Lisp in C
  - Prime versioni del C++ (C con classi)
  - Utilizzo dell'SQL embedded
  - Adattare il programma in funzione dell'installazione (Compilazione condizionale)
  - Espansione delle macro

Linker e Loader



Questi strumenti sono presenti anche negli ambienti integrati di sviluppo

Il compilatore lavora con linguaggi simbolici

- Ogni linguaggio simbolico è formato 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

La struttura di un compilatore

- Il processo di compilazione consiste in una serie di passi chiamati

  fasi della compilazione

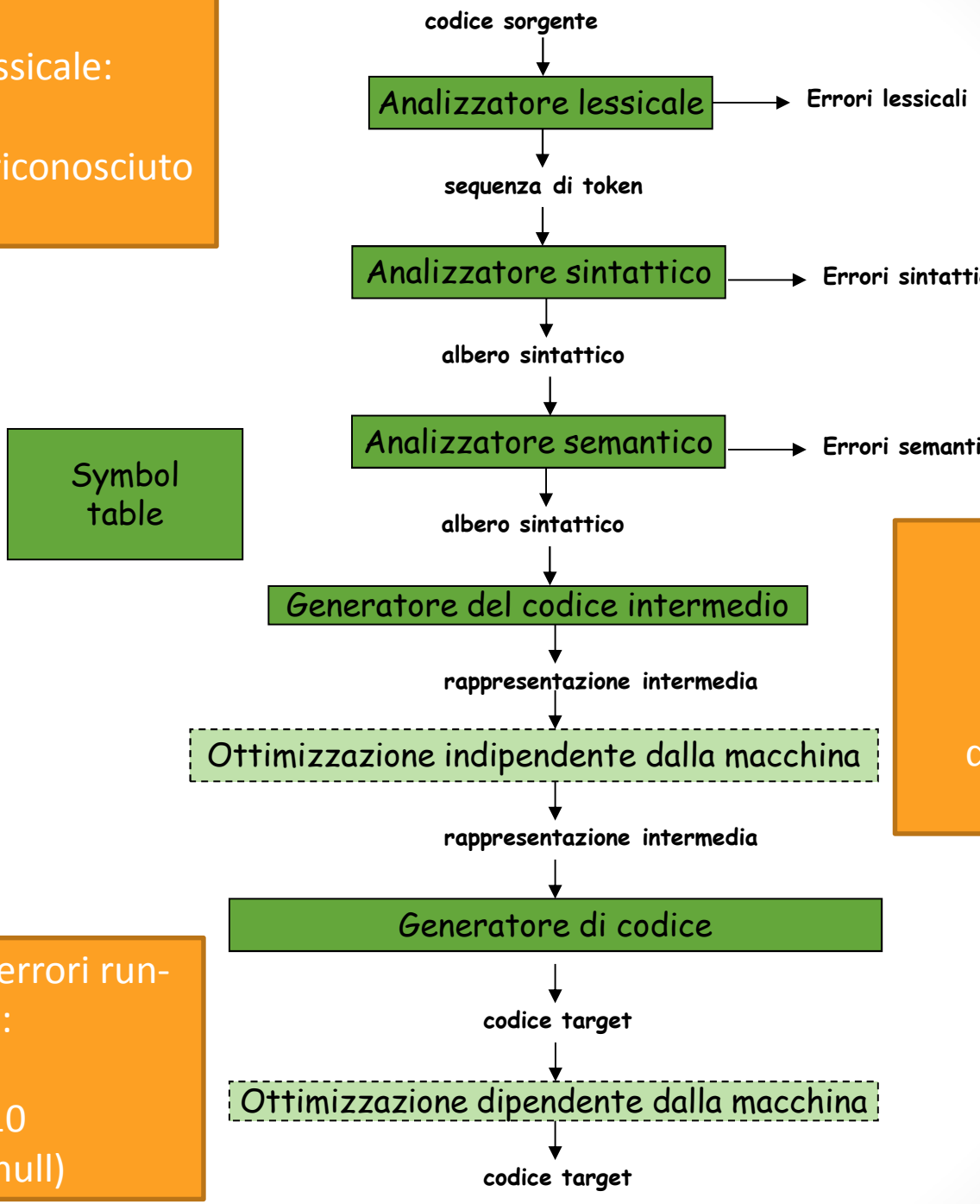
  - Analisi lessicale
  - Analisi sintattica
  - Analisi semantica
  - Generazione del codice intermedio

  } analisi

  - Ottimizzazione
  - Generazione del codice target
    o codice oggetto

  } sintesi

Le fasi della compilazione



Errore lessicale:

simbolo non riconosciuto

Symbol table

Errore sintattico:

if a:=5 then

Errore semantico:

y:=x+s

dove x,y sono interi
e s stringa

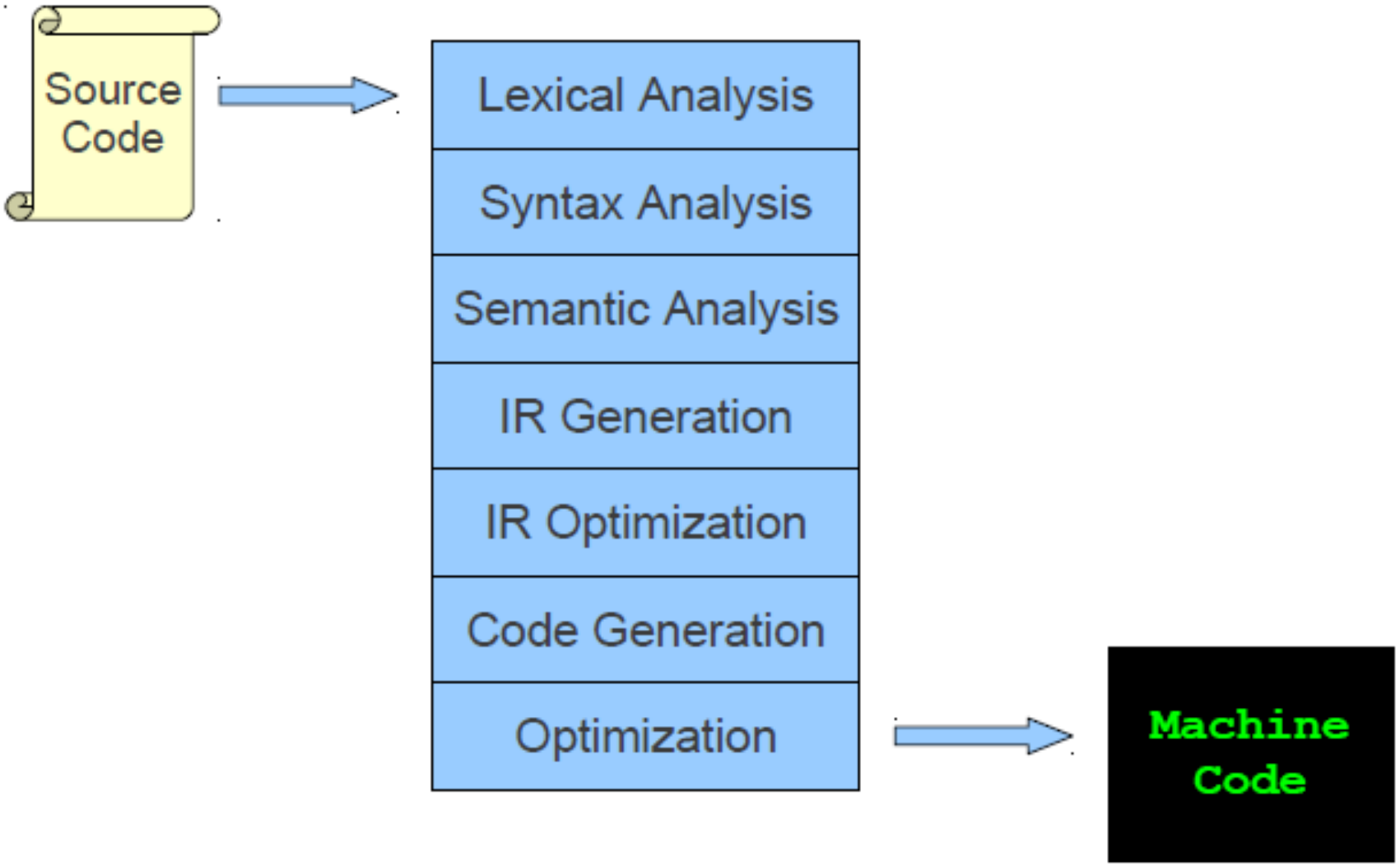
Non produce errori run-time:

*p:=10
(se p è null)

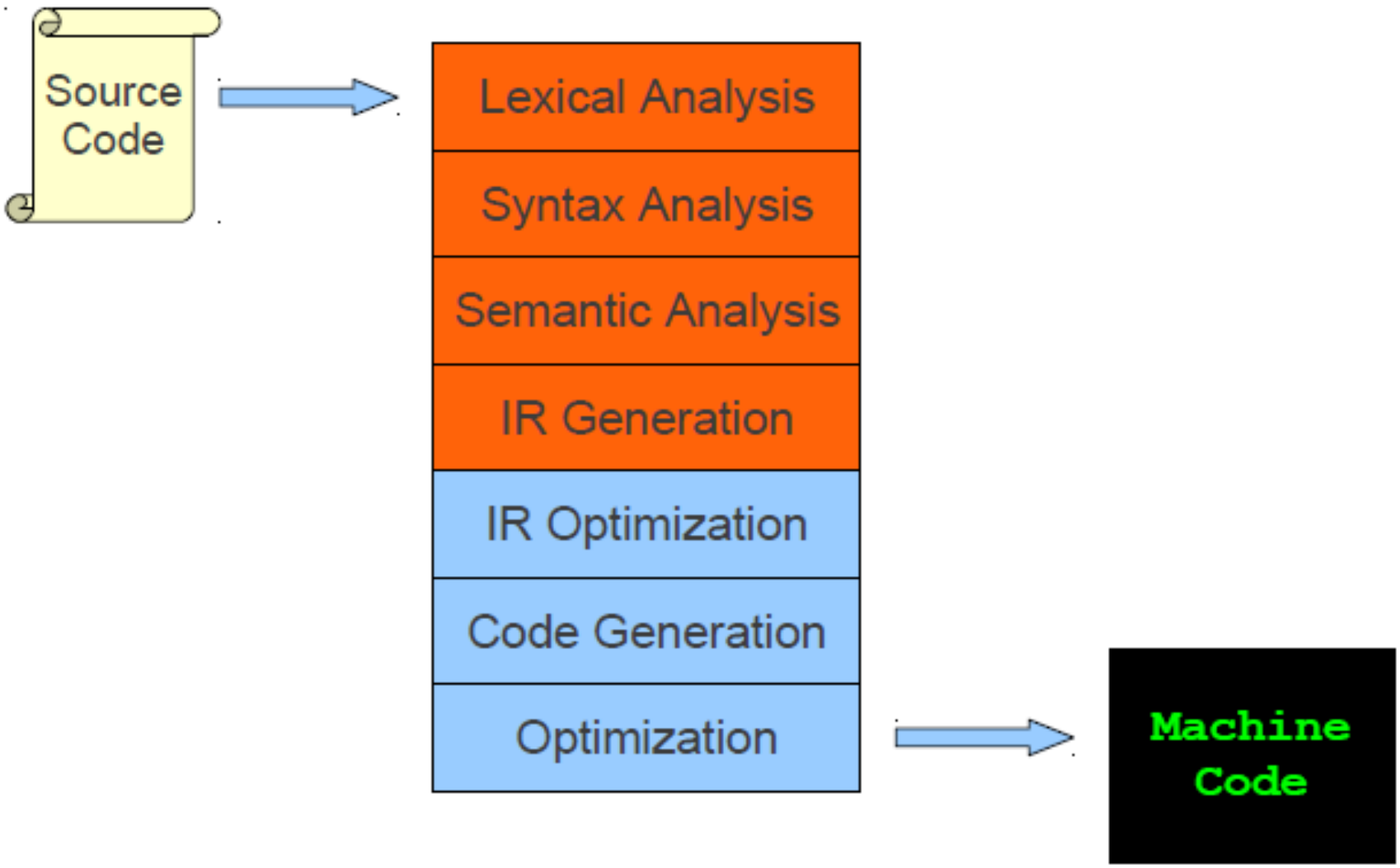
Front end & Back end

- La compilazione è stata suddivisa in due parti:
  - Analisi
  - Sintesi
- La prima parte, che dipende solo dal tipo di linguaggio che deve essere tradotto viene anche detta “front end” del compilatore.
- La seconda, insieme alle ottimizzazioni dipendenti dalla macchina viene detta “back end” del compilatore.

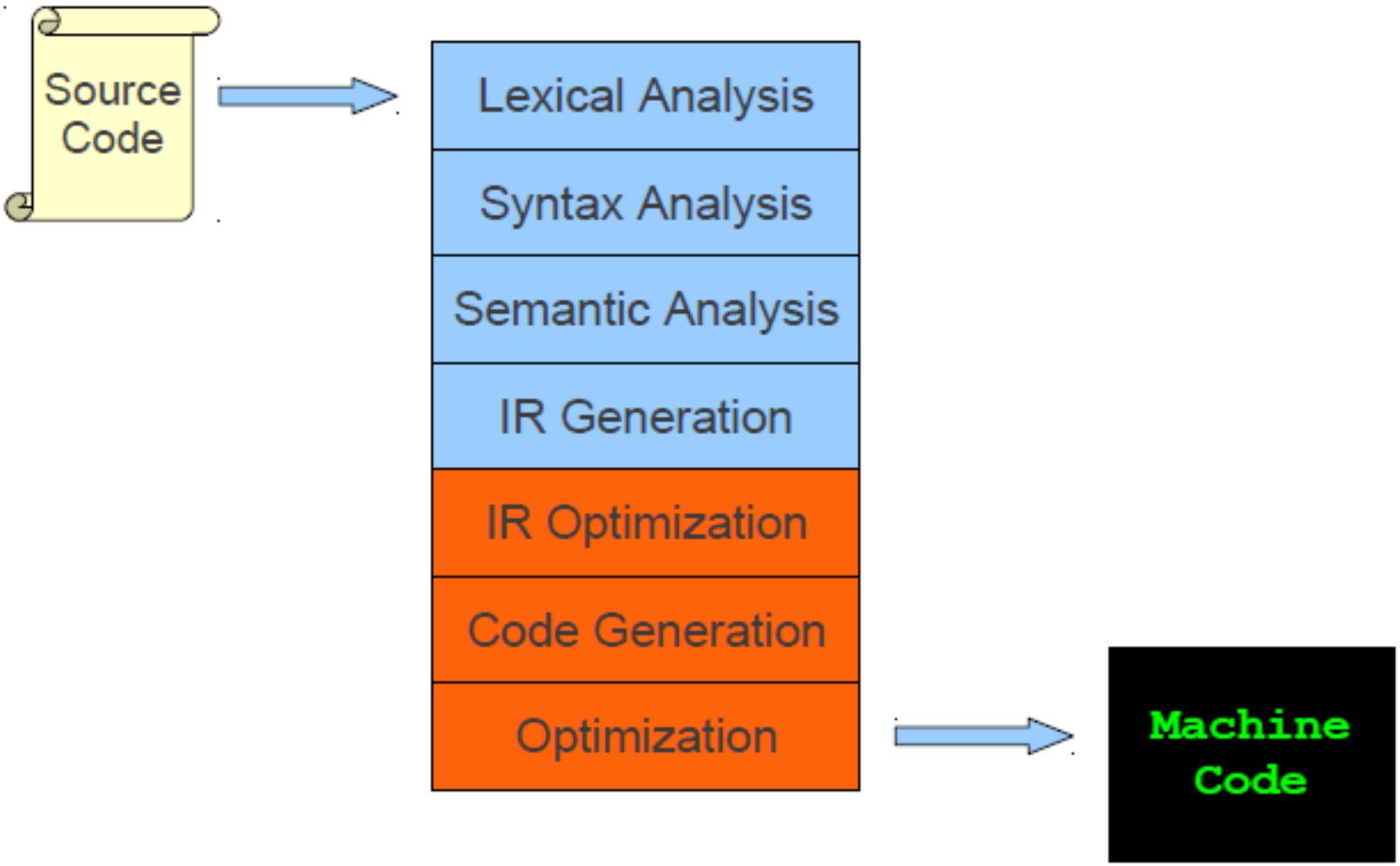
Facciamo un esempio



Front end


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Back end


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Analisi lessicale (lexical analysis o scanning)

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

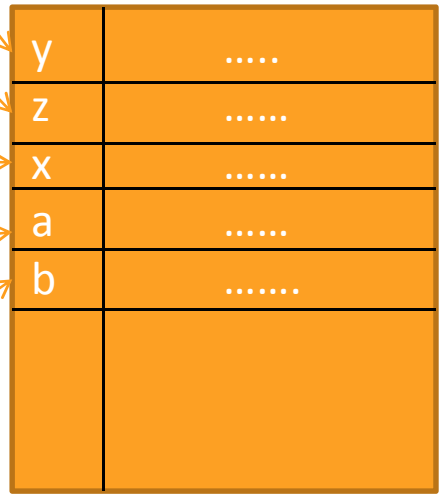
- I vari **lessemi**, una volta isolati, vengono memorizzati in una o più tabelle (tabelle dei descrittori o **symbol table**) dove saranno contenute tutte le indicazioni necessarie per identificare e caratterizzare i singoli **lessemi** (nome, tipo, visibilità,…;nel caso di nomi di funzioni, il numero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

Analisi lessicale

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

```
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

Individuazione dei lessemi e dei token

Per esempio nel seguente statement in C:

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

si individuano i seguenti elementi:

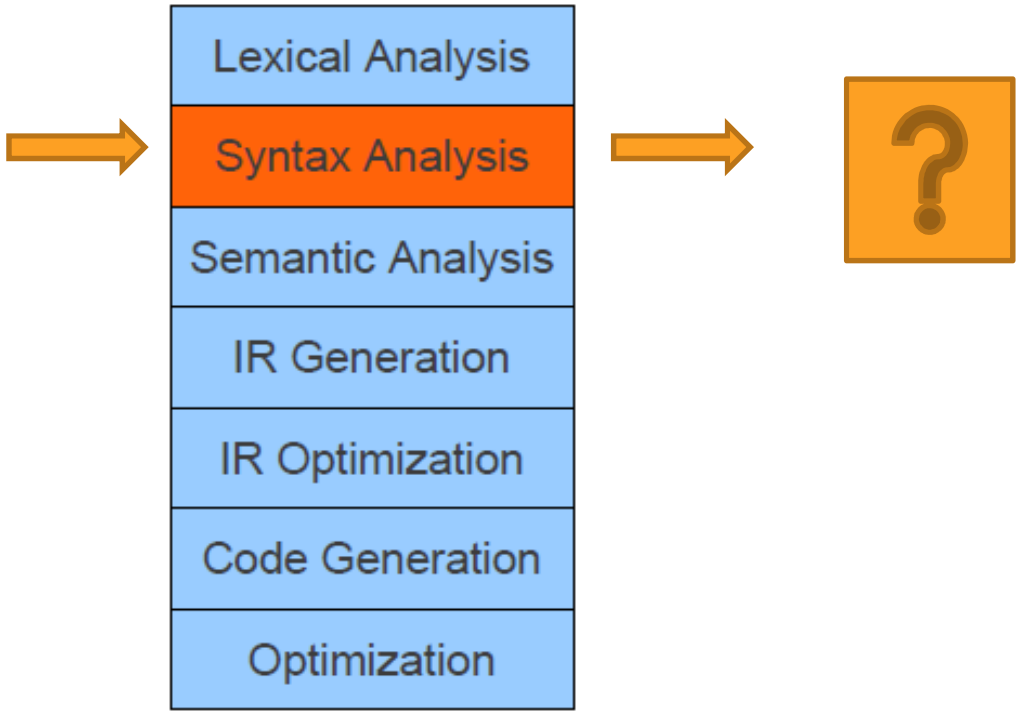
Token	**Lessema**
T_While	while
T_LeftParent	(
T_Identifier	y,z,x,a,b
T_Less	<
T_Assign	=
T_OpenBrace	{
T_int	int
T_semicolon	;
T_plus	+
….	
….	

Compiti dello scanner

- Costruzione delle symbol table
- Trasformazione del codice sorgente in forma semplificata, pronta per l'analisi sintattica.

Analisi sintattica

```
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
```



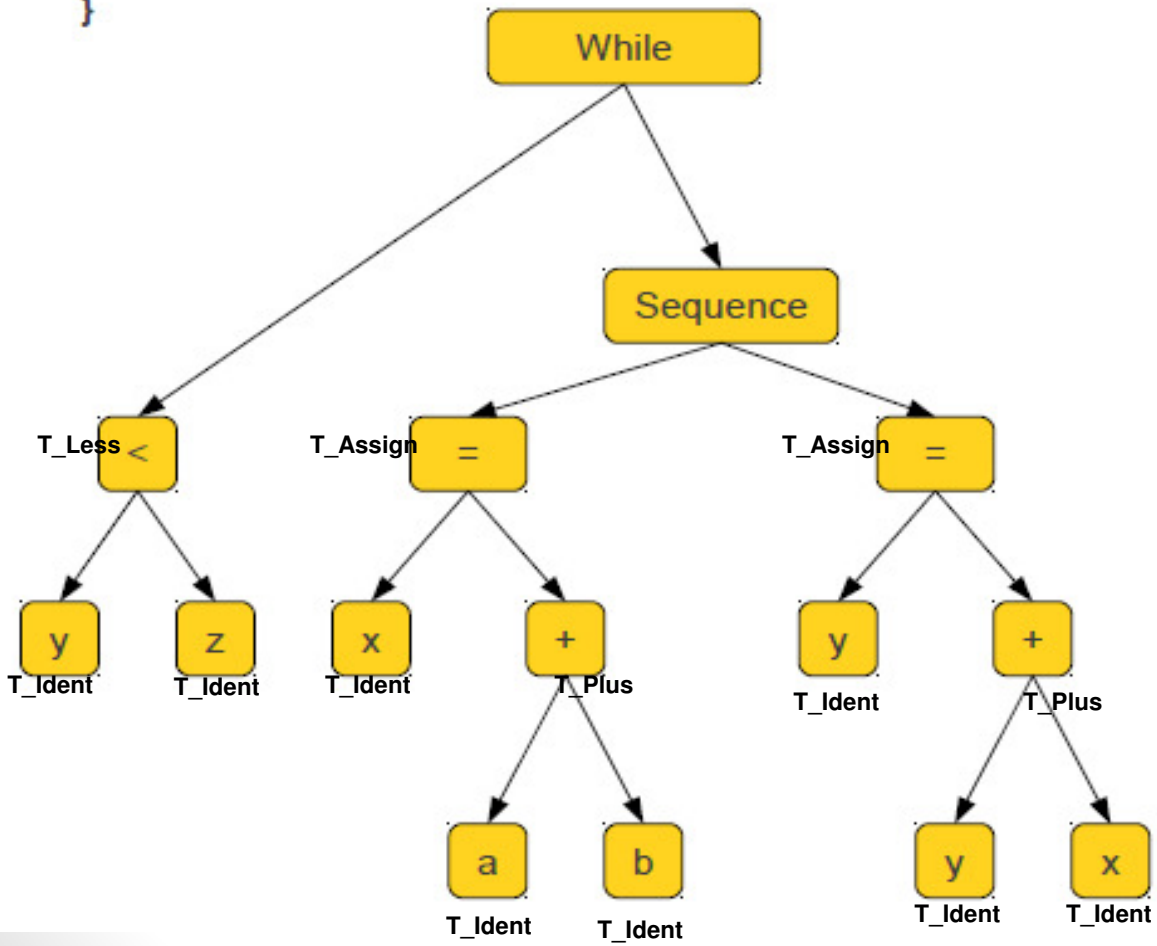
Analisi sintattica (syntax analysis o parsing)

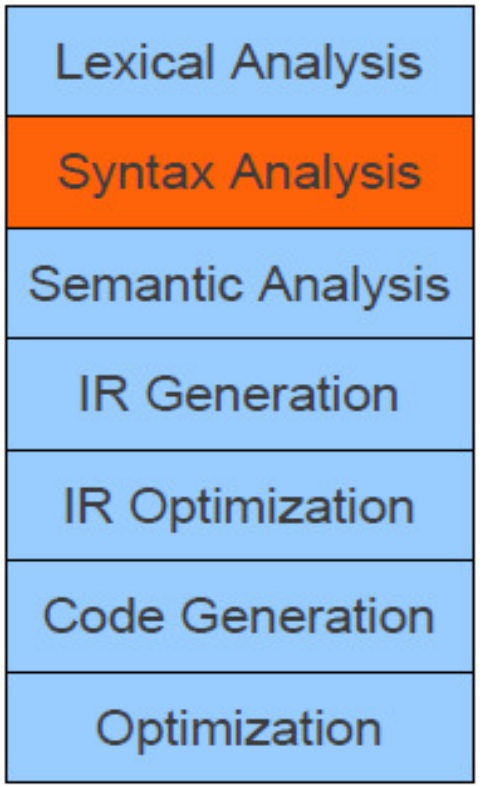
- In questa fase l'analizzatore sintattico (**parser**) usa i token (i nomi dei token) per definire la struttura grammaticale della sequenza di token, ovvero determina la struttura delle singole istruzioni e controlla se sono rispettate le regole della sintassi del linguaggio.
- I token diventano i simboli terminali della grammatica.
- Il parser adopera il codice semplificato e la symbol table costruiti nella fase precedente 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;
}
```

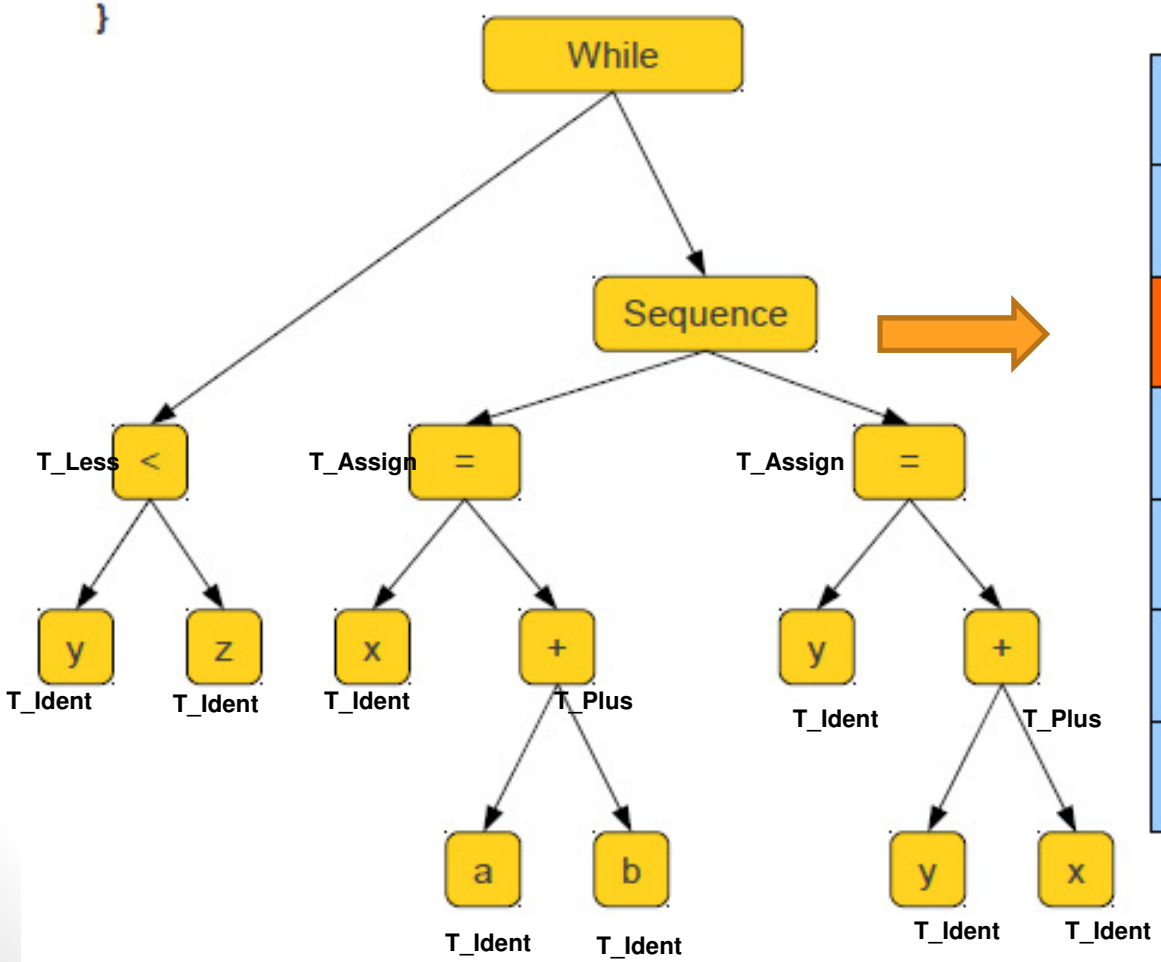
Le coppie <token,attributo> sono
i simboli terminali della grammatica

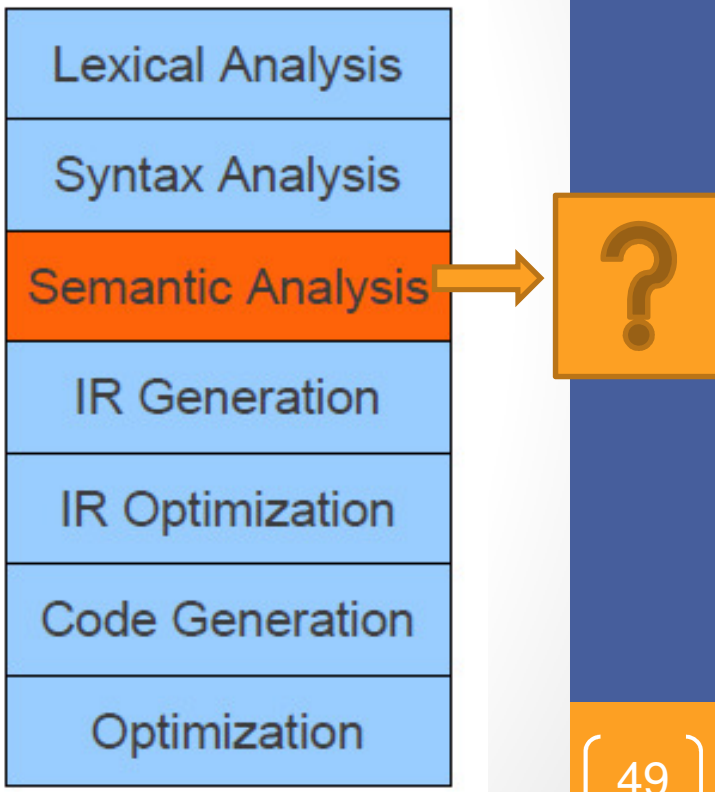




Analisi semantica

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





Analisi semantica (semantic analysis)

- Usa il syntax tree e la symbol table del programma sorgente per controllarne la consistenza semantica con le definizioni del linguaggio. L'analisi semantica consiste nell'interpretazione del "significato" delle strutture prodotte nella fase precedente controllando che esse siano legali e significative.

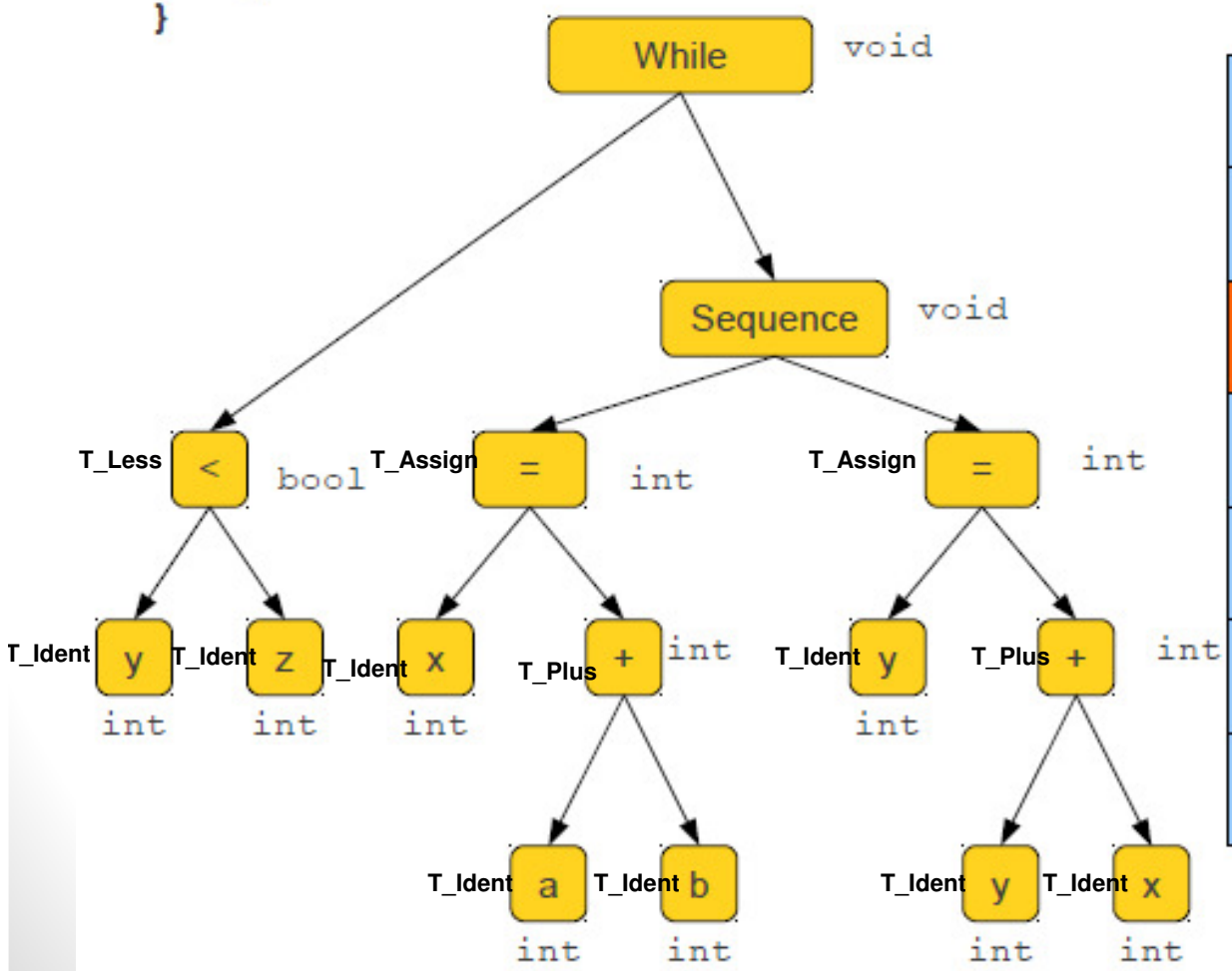
- Si controlla, per esempio, che le variabili coinvolte siano dichiarate e definite, che i tipi siano corretti, che gli operatori siano usati correttamente, ….

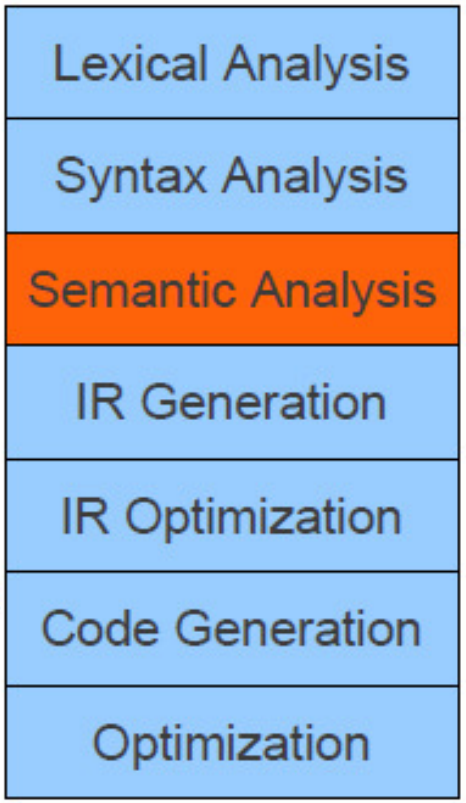
- Una parte importante dell'analisi semantica è il type checking dove il compilatore controlla che ogni operatore abbia gli operandi giusti. Per esempio:
  - controlla che l'indice di un array sia un intero
  - applica le regole di visibilità degli identificatori
  - applica le conversioni di tipo nel caso di operatori che possono essere applicati ad operandi di diverso tipo (coercions)

Produce l'albero sintattico decorato

Analisi semantica: output

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





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

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

```
var i : integer;
    a : array [1..100] of integer;
     ……….
     read(i);
     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 (Intermediate Representation)

- 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:

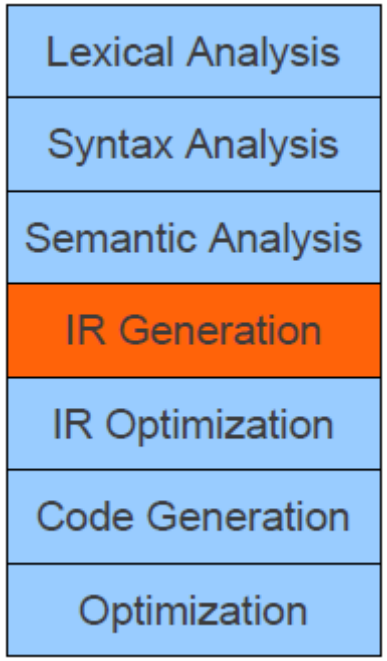
id := id1 operator id2

Codice Intermedio (Intermediate Representation)

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

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

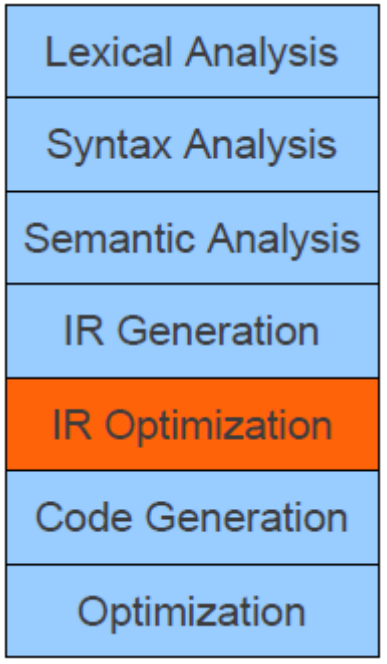
- In *3AC*:
  - Ogni istruzione ha al più un operatore sul lato destro.
  - Le istruzioni fissano l'ordine con cui vengono eseguite le operazioni
  - Vengono generati nomi temporanei per memorizzare i valori intermedi
  - Alcune istruzioni hanno meno di 3 operandi



Codice Intermedio ottimizzato

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

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



Altro esempio

- Vengono per esempio eliminate le somme di valori nulli e e le rivalutazioni delle stesse espressioni

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

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

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 dalla macchina* operate dal *post-ottimizzatore*.

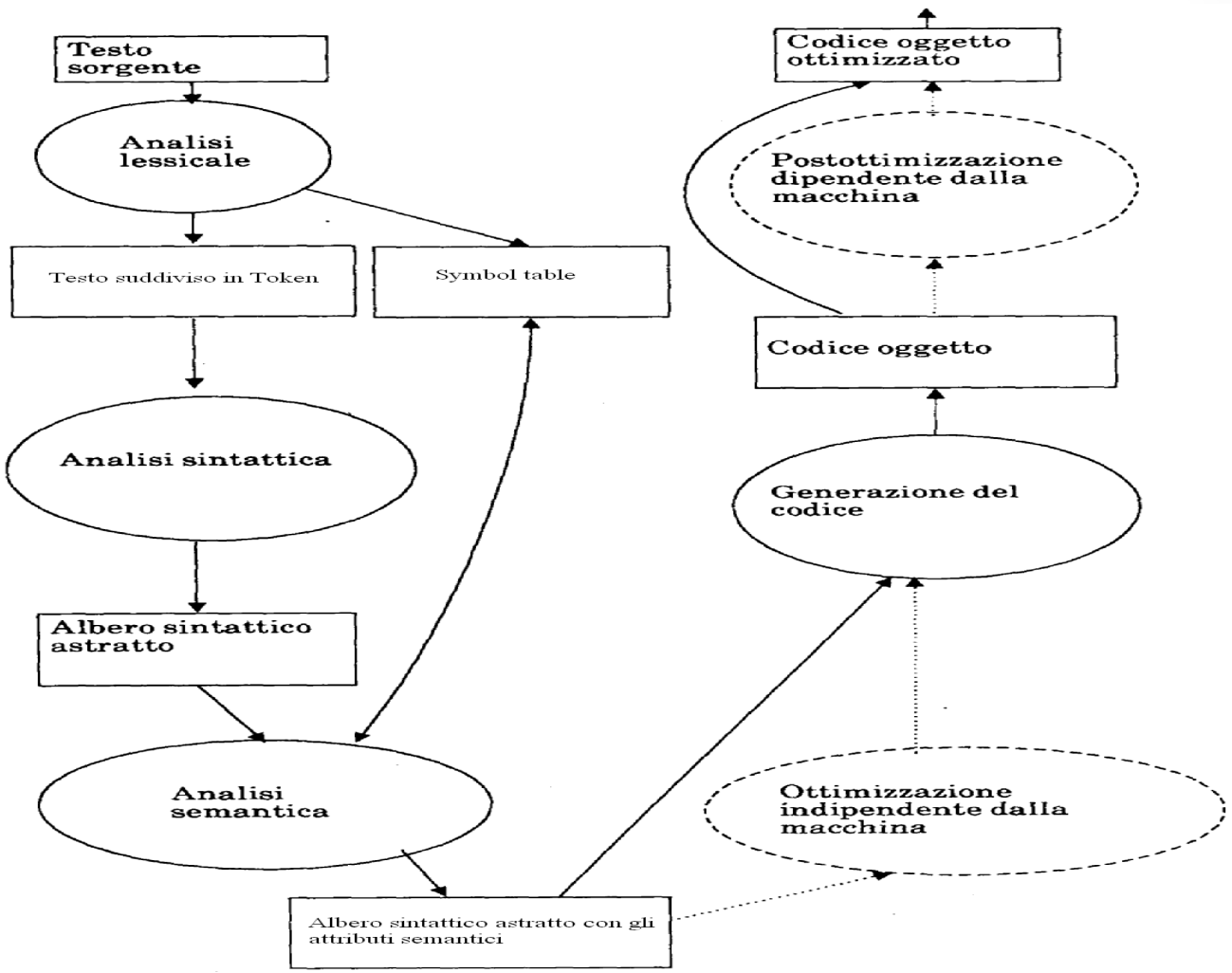
Esempi di ottimizzazioni indipendenti dalla macchina

- 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)

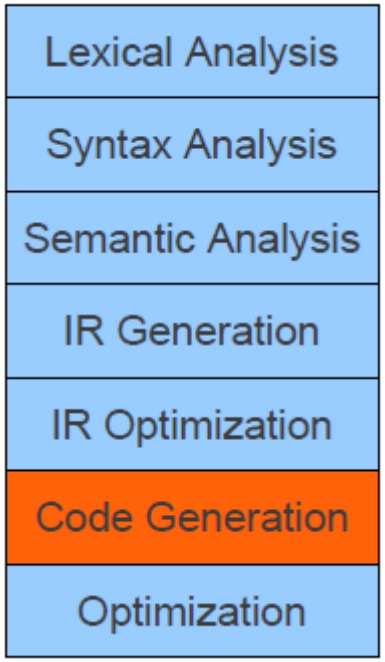


N.B. Il problema di generare il codice target ottimale è indecidibile.

Generazione del codice oggetto

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

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

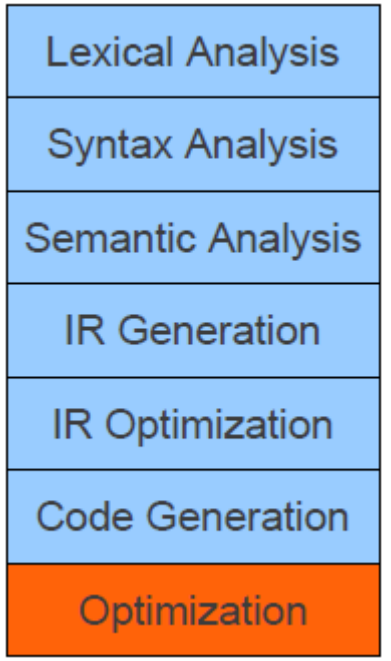


L'output è solitamente codice macchina o codice assembly

Ottimizzazione

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

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



Generazione del codice target o oggetto

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

Gestione degli errori

- Durante tutte le fasi della compilazione possono venir riscontrati 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.
- 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.

Quante passate?

- Un compilatore ad una sola passat
- Alcuni linguaggi sono stati progett… n una sola passata (Pascal).
- In altri casi le possibilità offerte da… impediscono la compilazione in un…
- Uno svantaggio della compilazione… possibile effettuare delle ottimizza… generare codice di alta qualità.
- Può essere difficile, infatti, valutar… e deve essere analizzata per produrr…

1. Nel caso di linguaggi che prevedono l'uso di mutua ricorsione, per es.

P(x)=1 se x<=1 e Q(x+2) se x>1

Q(x)=P(x-3)+5

2. In JAVA la dichiarazione delle variabili e dei metodi non devono seguire in genere un particolare ordine:

```
Class Prova {

  void inc () {n=n+1}
  int n;
  void use() {n=0; inc();}
}
```

Realizzare un compilatore

- Il primo compilatore fu scritto in linguaggio assembler (e non c'erano altre alternative)
- Se si dispone di un compilatore C_1 per un determinato linguaggio L_1 è possibile scrivere un compilatore C_2 per un altro linguaggio L_2.
- E' 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 il compilatore gira su una macchina diversa da quella su cui girerà il codice target (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.