Benvenuti al corso

Linguaggi Formali e Compilatori

Prof. Arturo Carpi

2021/22

Obiettivi formativi

Conoscenze

- teoria dei linguaggi formali
- aspetti formali sintattico/semantici dei linguaggi di programmazione
- funzionamento di interpreti e compilatori.

Abilità

- progettare analizzatori lessicali e analizzatori sintattici per semplici linguaggi di programmazione

Abilità comunicative

- terminologia relativa a grammatiche, automi, compilatori

Prerequisiti

Matematica Discreta

Le nozioni di funzione, relazione, equivalenza, partizione, insieme ordinato, semigruppo, grafo, albero

Algoritmi e strutture dati

Risulterà utile la conoscenza dei più semplici algoritmi su grafi diretti (accessibilità, verifica della connessione, ecc.)

Esame

Prova scritta

Oggetto quattro problemi di carattere algoritmico/computazionale

Durata 90'

Obiettivo verificare la capacità di applicare le conoscenze apprese alla soluzione di problemi pratici

Validità 12 mesi

Prova orale

Durata circa 30'

Obiettivo verificare

- comprensione dei contenuti teorici
- conoscenza degli argomenti in programma
- capacità espositiva (linguaggio tecnico)

Lingua Italiano, Inglese o Francese.

Materiale didattico

Su Unistudium:

- dispense
- slides
- esercizi proposti
- avvisi
- programma del corso
- elenco dei testi consigliati

Ricevimento

Lunedì ore 13–15, Martedì ore 10–11 (salvo modifiche)
oppure su appuntamento (anche su Teams)

Per comunicare con me usare esclusivamente

- e-mail istituzionale (nome.cognome@studenti.unipg.it) con un soggetto significativo
- messaggistica Unistudium

Il corso

Durata del corso

6 CFU = 42 ore

Per la miglior riuscita del corso ...

Intervenite in presenza, in audio o con messaggi per

- chiarimenti
- curiosità
- segnalare errori
- soprattutto se faccio riferimento a qualche nozione che non conoscete

Per chi è a distanza: quando non volete intervenire tenete camera e microfono spenti

DOMANDE ?

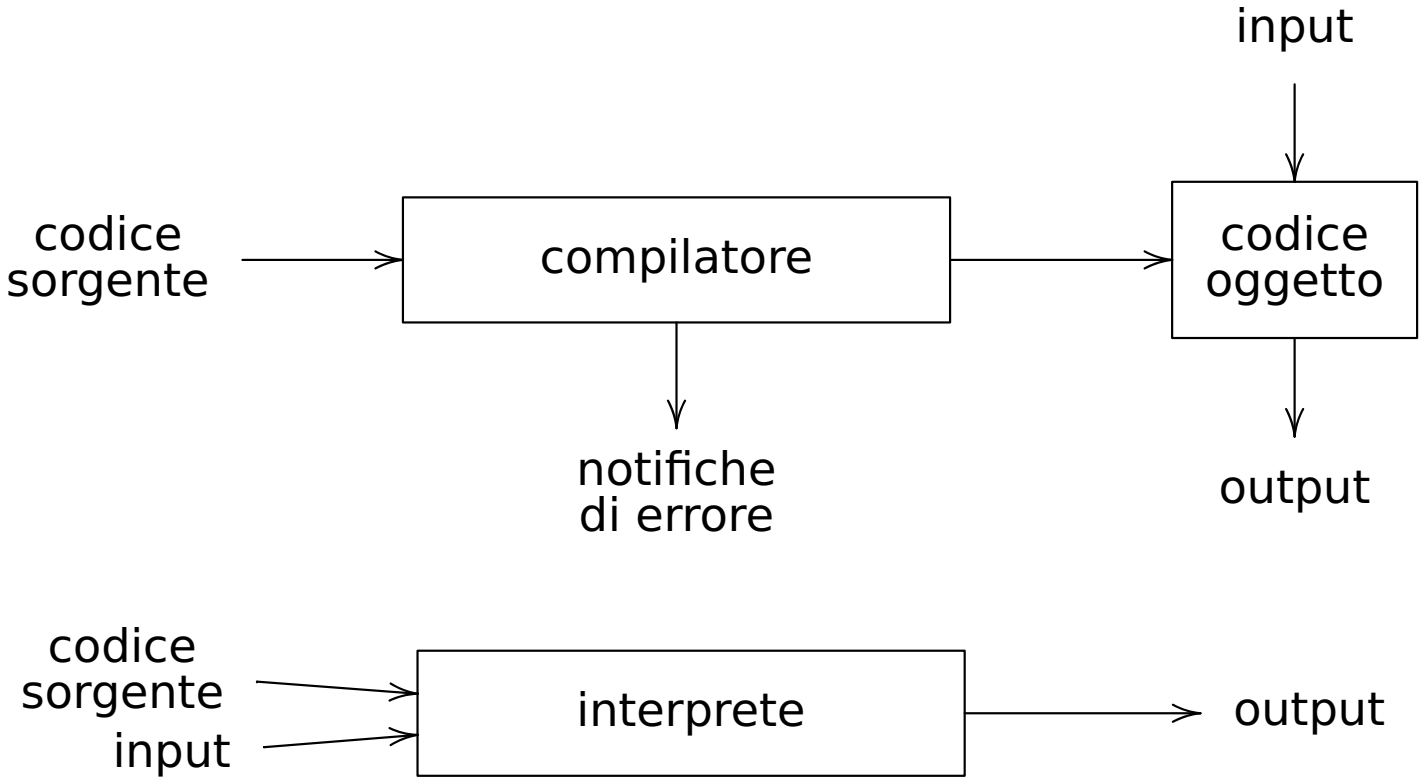
Le fasi della compilazione

Arturo Carpi

Dipartimento di Matematica e Informatica
Università di Perugia

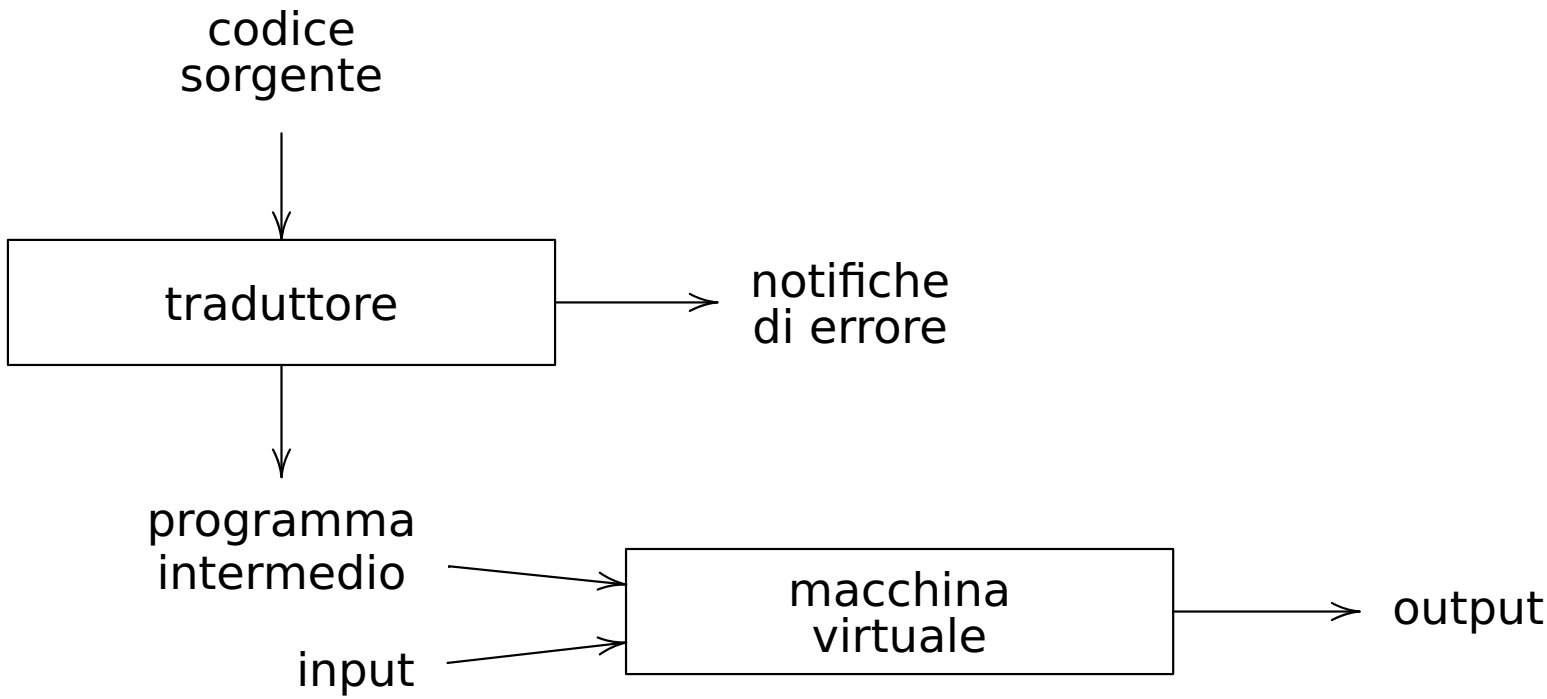
Corso di Linguaggi Formali e Compilatori - a.a. 2021/22

Compilatori e interpreti



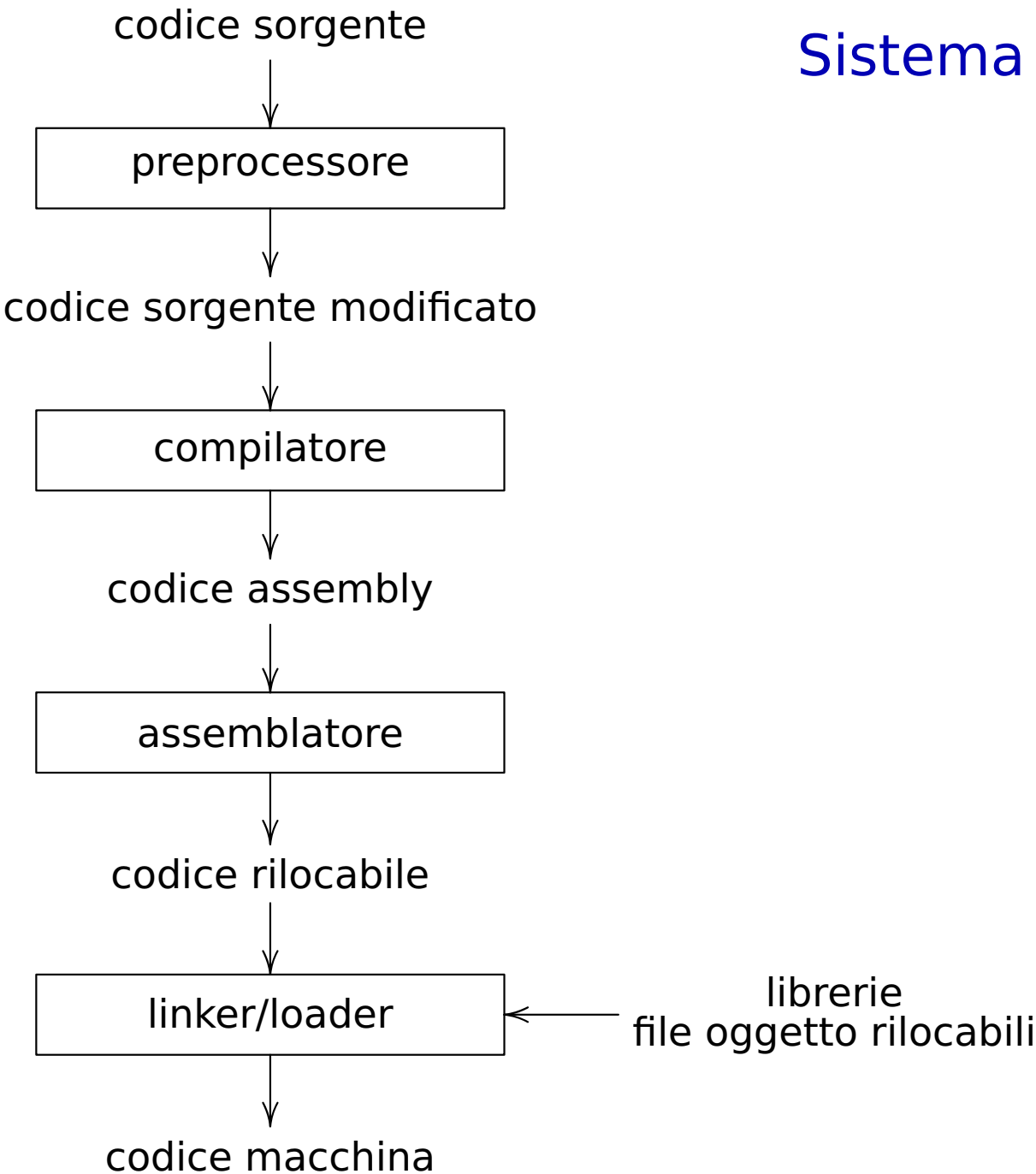
- Il compilatore esegue la traduzione prima dell'esecuzione
- È anche possibile la traduzione da un linguaggio di programmazione a un altro linguaggio di programmazione
- L'interprete esegue la traduzione istruzione per istruzione simultaneamente all'esecuzione

Compilatori ibridi



- Il traduttore esegue la traduzione in un linguaggio intermedio (e.g., Java bytecode)
- La macchina virtuale (e.g., JVM) interpreta il codice intermedio

Sistema di compilazione



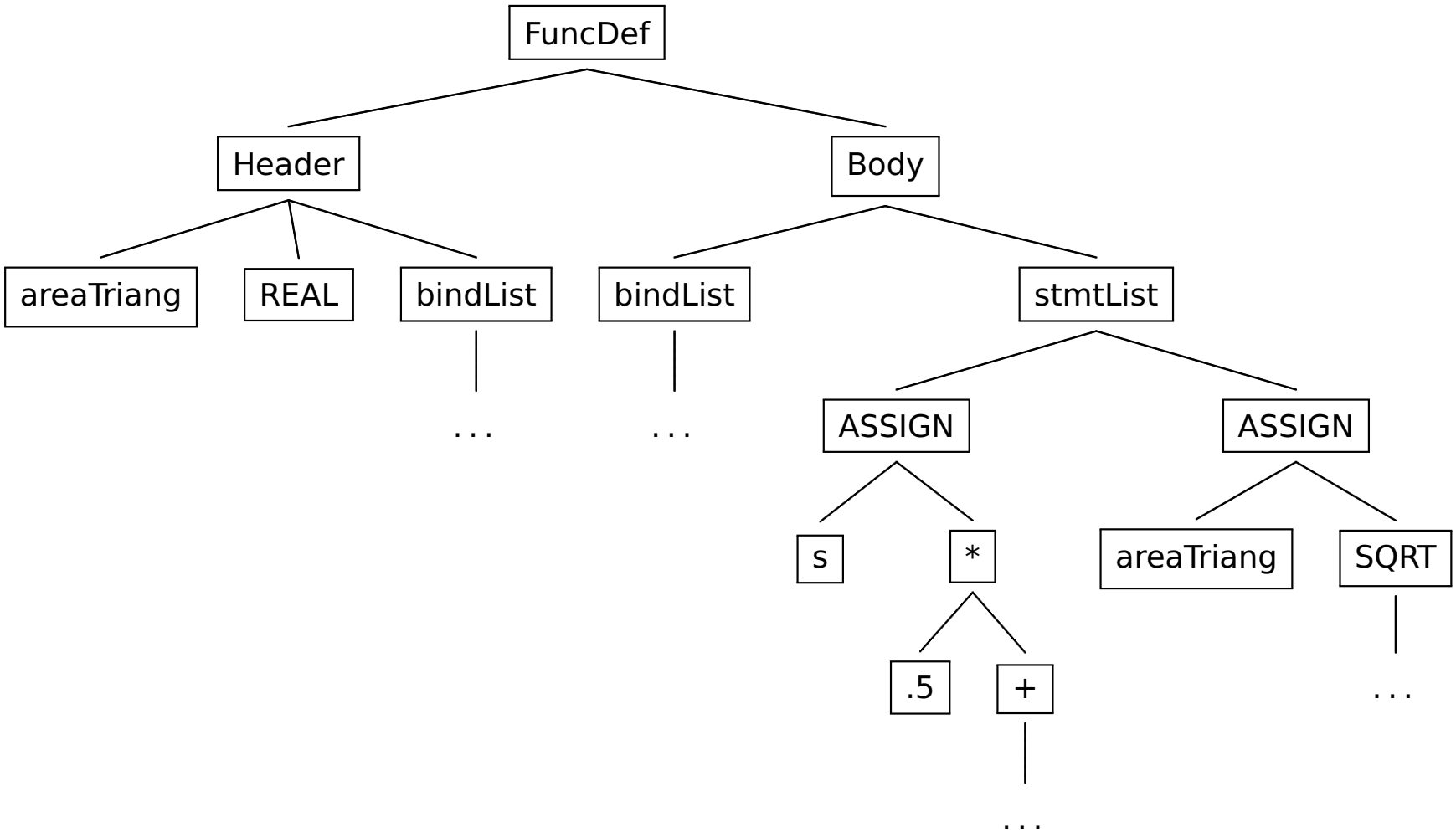
$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

```
FUNCTION areaTriang(a,b,c: real): real;                    PASCAL
VAR s: real;
    BEGIN
        s := (a+b+c)*0.5;
        areaTriang := sqrt(s*(s-a)*(s-b)*(s-c))
    END;

fun areaTriang(a: real, b: real, c: real) =                ML
    let
        val s = (a+b+c)*0.5
    in
       sqrt(s*(s-a)*(s-b)*(s-c))
    end

double areaTriang(double a, double b, double c) {          JAVA
    double s = (a+b+c)*0.5;
    return Math.sqrt(s*(s-a)*(s-b)*(s-c));
}
```

Albero sintattico astratto



Analisi e sintesi

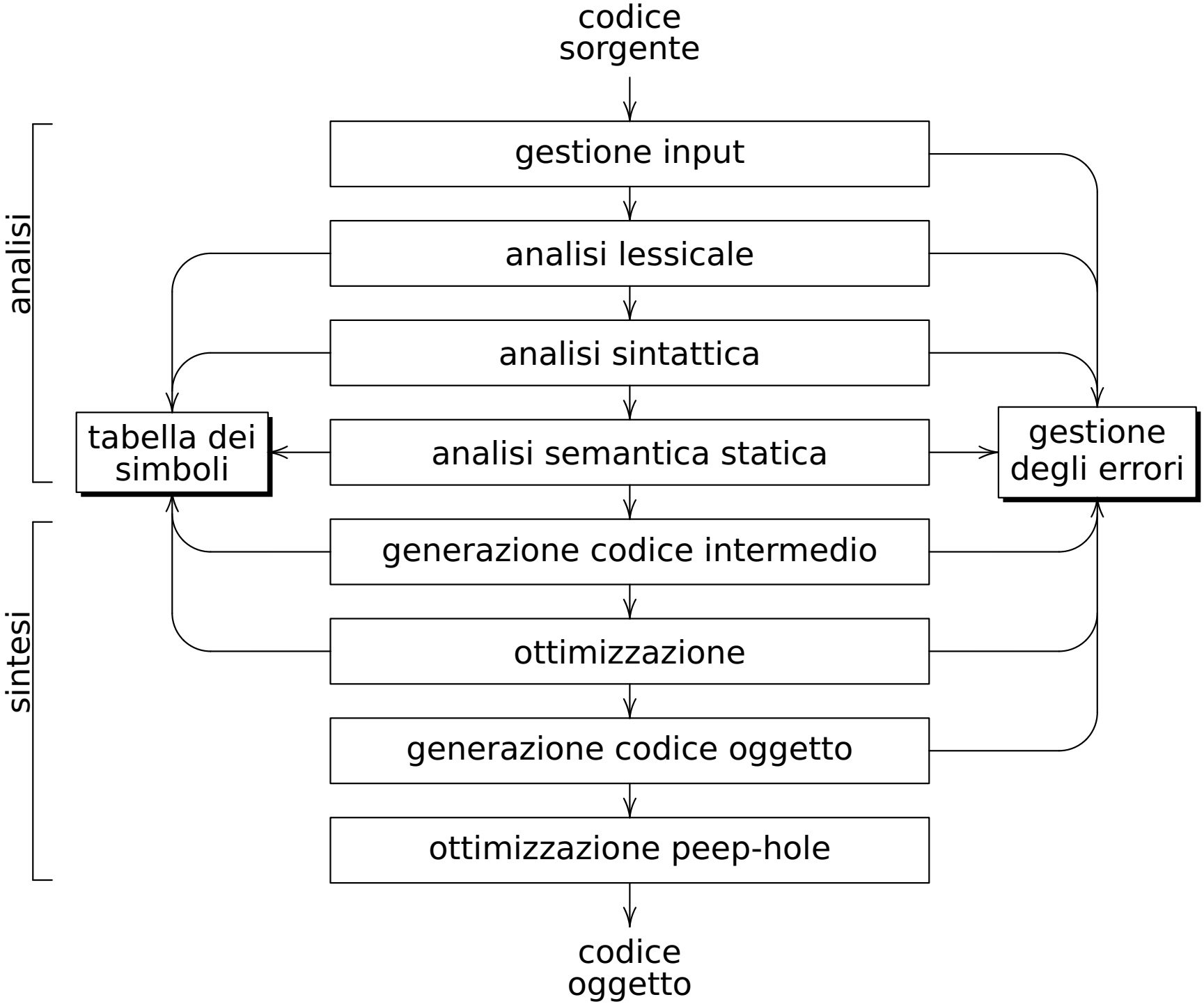
La compilazione si divide in:

Fase analitica

La costruzione dell'albero sintattico astratto dal codice sorgente

Fase sintetica

La produzione del codice oggetto a partire dall'albero sintattico astratto


codice
sorgente
gestione input
analisi lessicale
analisi sintattica
analisi semantica statica
generazione codice intermedio
ottimizzazione
generazione codice oggetto
ottimizzazione peep-hole
codice
oggetto
tabella dei
simboli
gestione
degli errori
analisi
sintesi

Lessico, sintassi, semantica

- To be or not to be
- Popolo del è sovranità la
- Il mattone mangia una canzone verde

Analisi lessicale

L'analizzatore lessicale (scanner)

- raggruppa i caratteri in sequenze elementari (lessemi)
- associa a ogni lessema
  - una classe lessicale (token)
  - un attributo, che viene registrato nella tabella dei simboli

Esempio

```
position = initial + rate * 60
```

Lessema	Token	attributo
position	id	1
=	assign	-
initial	id	2
+	sum	-
rate	id	3
*	prod	-
60	number	4

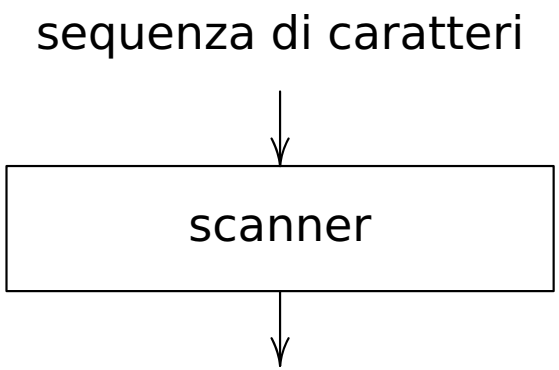
tabella dei simboli	
1	position
2	initial
3	rate
4	60

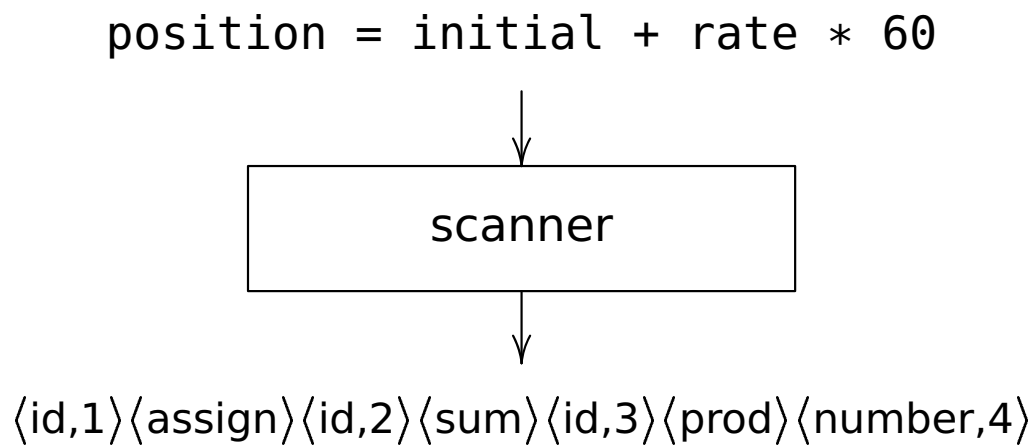
oppure . . .

Lessema	Token	attributo
position	id	1
=	op	2
initial	id	3
+	op	4
rate	id	5
*	op	6
60	number	7

tabella dei simboli	
1	position
2	assign
3	initial
4	sum
5	rate
6	prod
7	60

Analisi lessicale



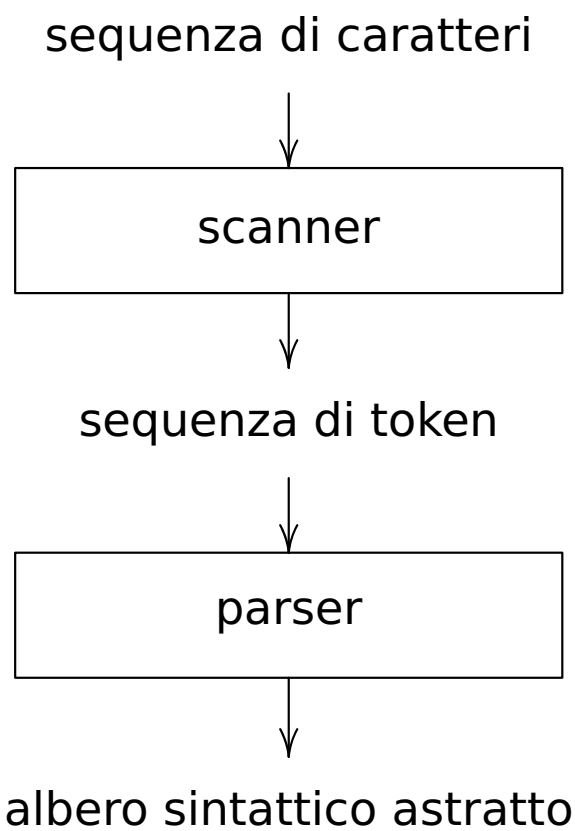


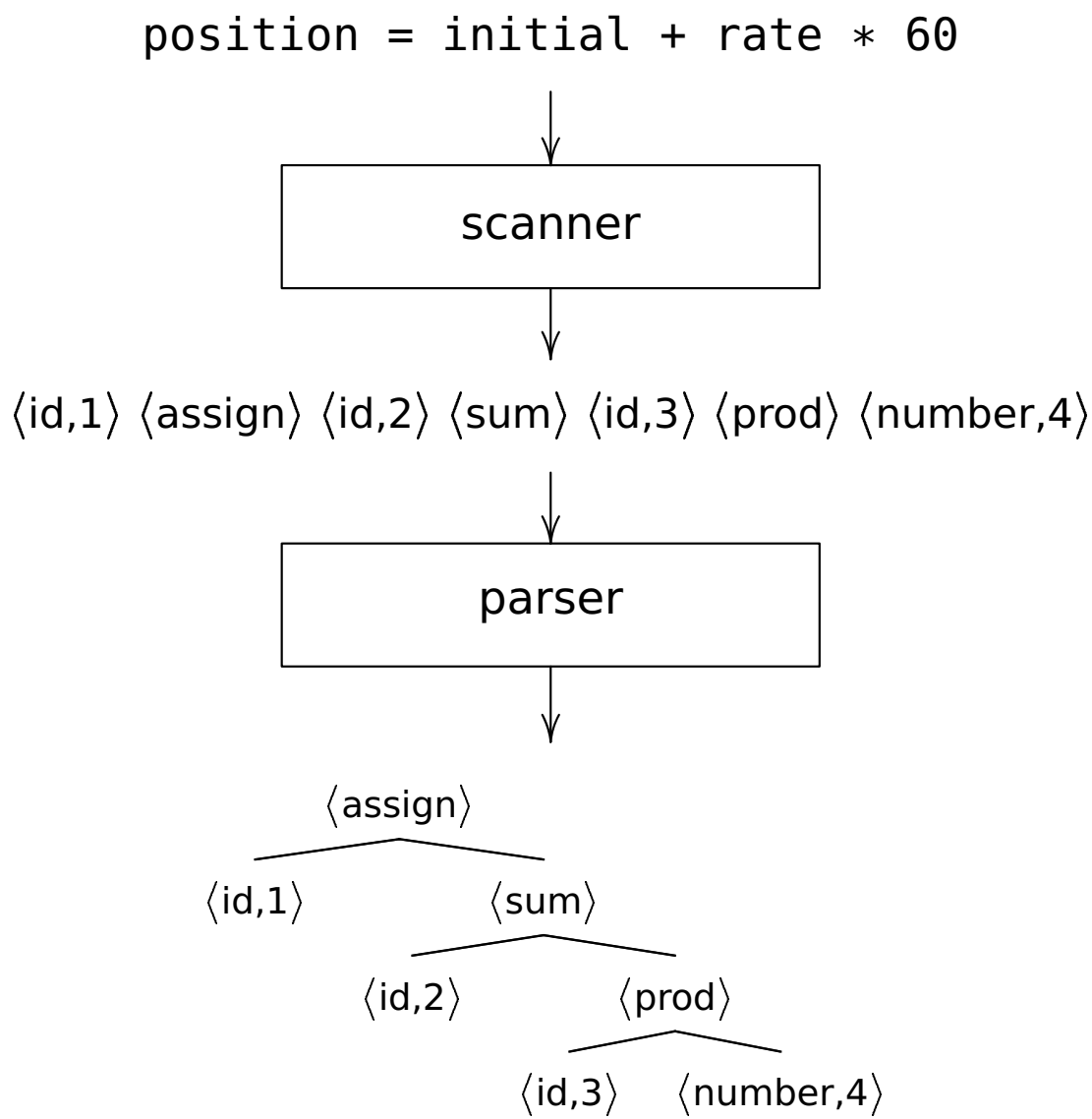
Analisi sintattica

L’analizzatore sintattico (parser) organizza i tokens prodotti dallo scanner in un albero (albero sintattico astratto)

- i nodi interni rappresentano operazioni
- i figli del nodo rappresentano gli argomenti dell’operazione

Analisi sintattica





Analisi semantica statica

L'analisi semantica utilizza l'albero sintattico astratto e la tabella dei simboli per verificare che il programma sorgente sia semanticamente coerente con la definizione del linguaggio.

Type checking

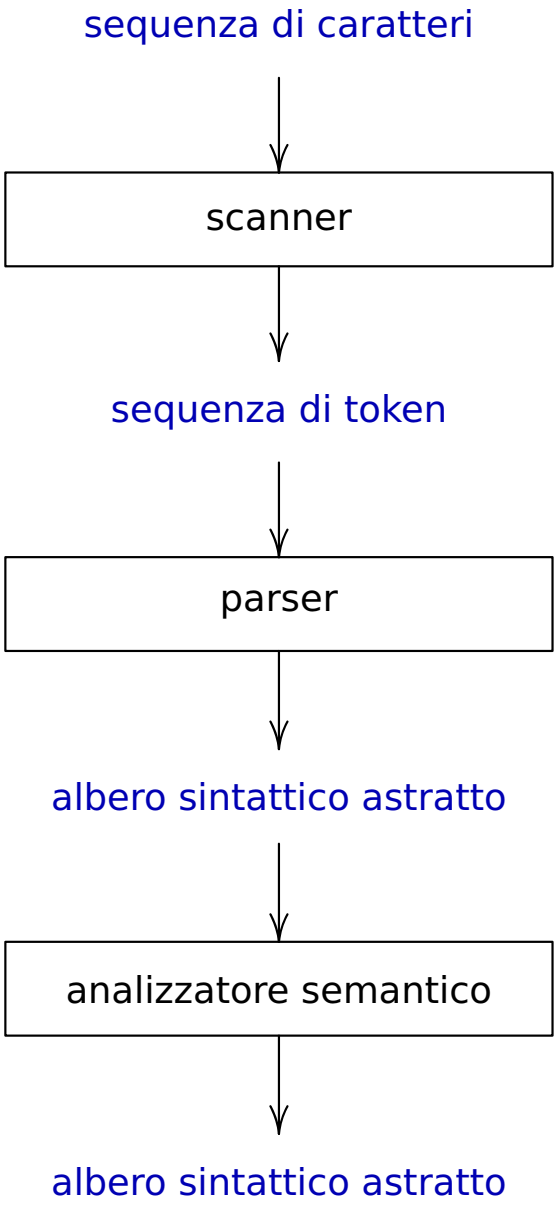
Verifica che ogni operatore abbia tutti gli operandi del tipo corretto. Può eseguire la coercizione

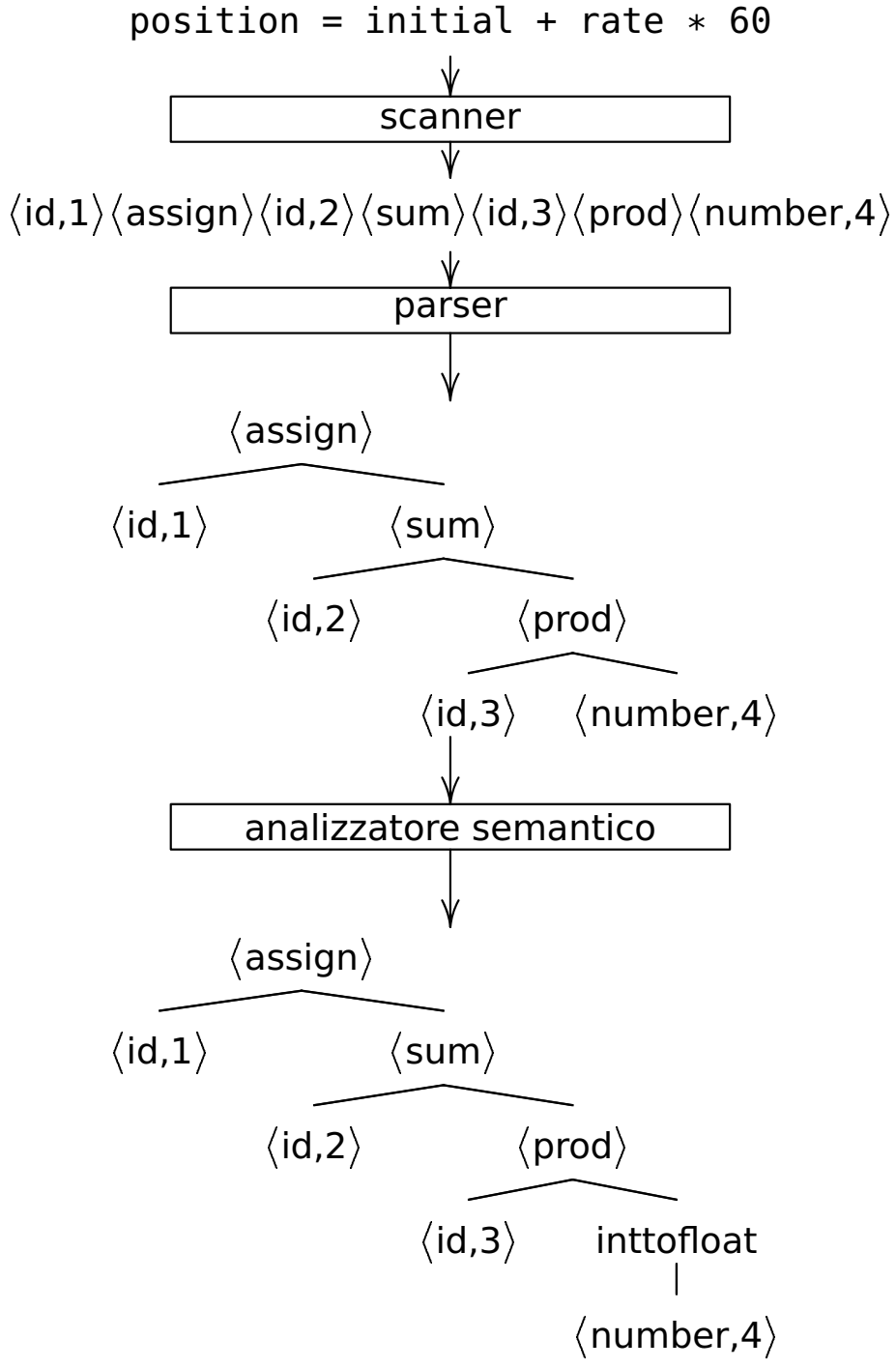
Esempi

In C è permesso moltiplicare un intero per un reale; è quindi necessario che il compilatore converta silenziosamente l'intero in un reale.

In Pascal, il costrutto `WHILE` richiede un valore di tipo `BOOLEAN`. Pertanto, l'analizzatore semantico rileverà un errore nell'espressione

```
WHILE X DO X := X - 1 ;
```


sequenza di caratteri
scanner
sequenza di token
parser
albero sintattico astratto
analizzatore semantico
albero sintattico astratto

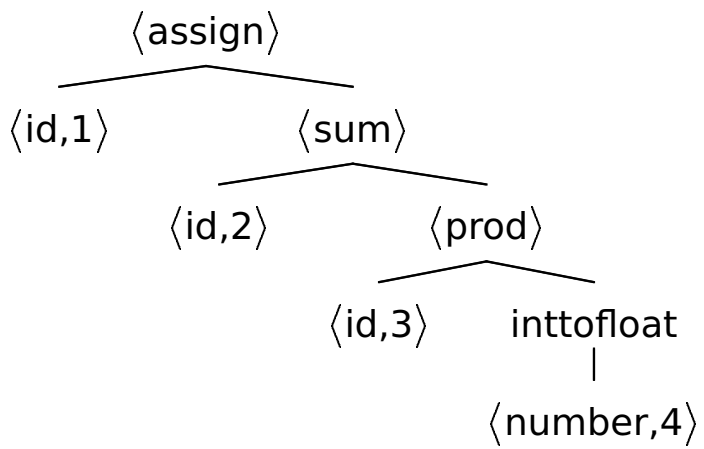

position = initial + rate * 60
scanner
⟨id,1⟩⟨assign⟩⟨id,2⟩⟨sum⟩⟨id,3⟩⟨prod⟩⟨number,4⟩
parser
⟨assign⟩
⟨id,1⟩
⟨sum⟩
⟨id,2⟩
⟨prod⟩
⟨id,3⟩
⟨number,4⟩
analizzatore semantico
⟨assign⟩
⟨id,1⟩
⟨sum⟩
⟨id,2⟩
⟨prod⟩
⟨id,3⟩
inttofloat
⟨number,4⟩

Generazione del codice intermedio

Dall'albero sintattico astratto (e dalla tabella dei simboli) si ottiene un codice in un linguaggio con

- istruzioni elementari e facili da tradurre in codice macchina
- indipendente dall'architettura (non specifica gli indirizzi in memoria nè i registri)

Esempio



$\Longrightarrow$

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

Ottimizzazione

L'ottimizzatore cerca di ridurre il tempo o lo spazio necessario all'esecuzione del codice intermedio.

Esempio

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

$\Longrightarrow$

```
t2 = id3 * 60.0
id1 = id2 + t2
```

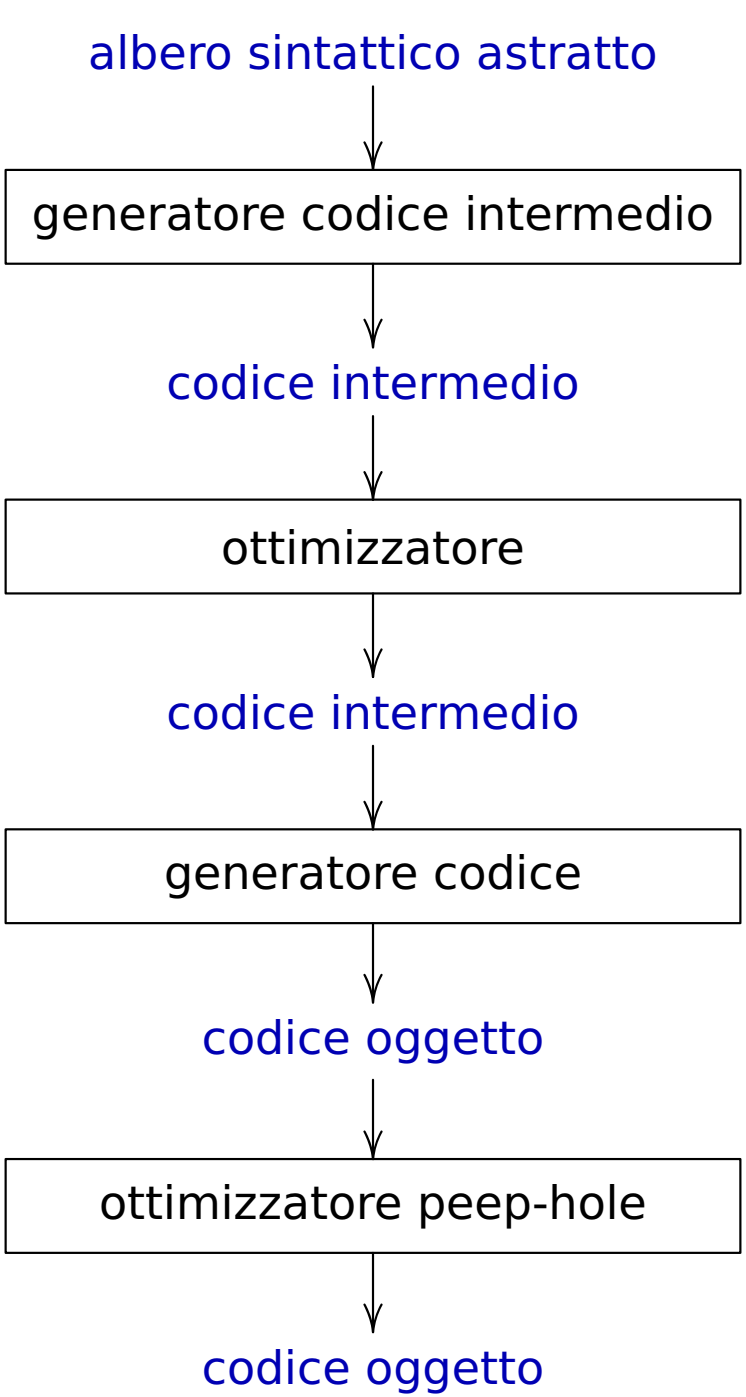
Generazione del codice oggetto

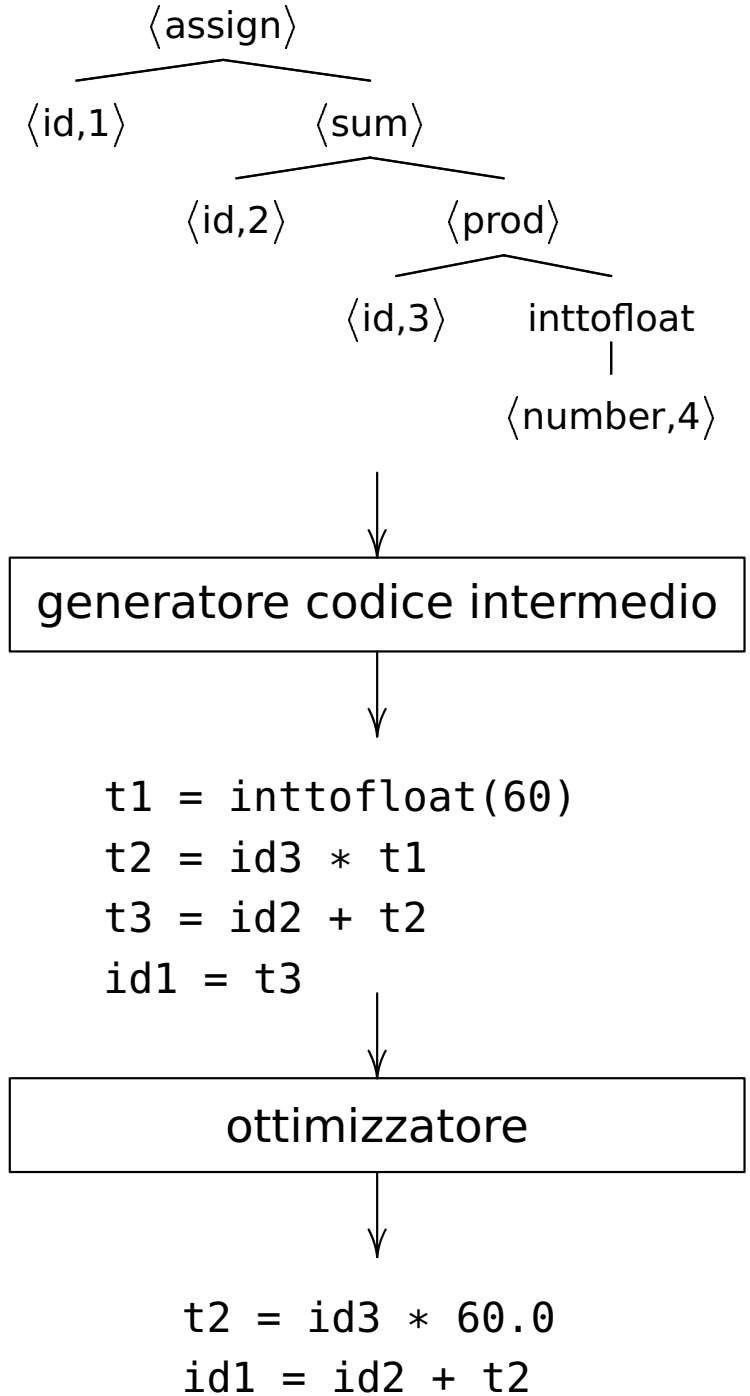
È necessario

- fissare le locazioni di memoria dei dati
- generare il codice per accedere a tali dati
- selezionare i registri per i calcoli intermedi
- ...

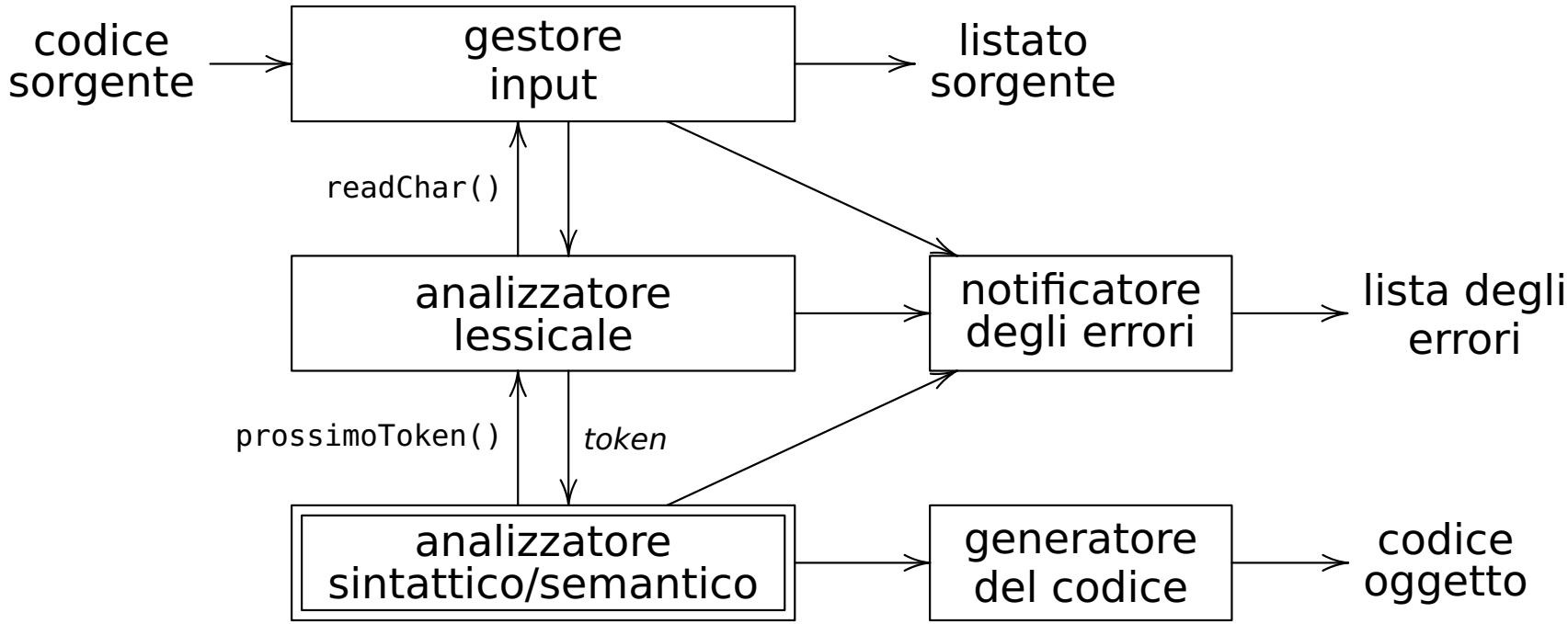
È la parte più complessa della compilazione e dipende strettamente dall'architettura della piattaforma (ma è riutilizzabile)

Infine è possibile un'ulteriore ottimizzazione, legata alle specificità dell'architettura (ottimizzazione peep-hole)


albero sintattico astratto
generatore codice intermedio
codice intermedio
ottimizzatore
codice intermedio
generatore codice
codice oggetto
ottimizzatore peep-hole
codice oggetto


⟨assign⟩
⟨id,1⟩
⟨sum⟩
⟨id,2⟩
⟨prod⟩
⟨id,3⟩
inttofloat
⟨number,4⟩
generatore codice intermedio
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
ottimizzatore
t2 = id3 * 60.0
id1 = id2 + t2

Organizzazione del compilatore



- Il nucleo è l'analizzatore sintattico (che realizza anche l'analisi semantica statica)
- l'analizzatore sintattico invoca la funzione `prossimoToken`, implementata nell'analizzatore lessicale
- l'analizzatore sintattico attiva, mediante la chiamata di funzioni opportune, la generazione del codice