

# Linguaggi di programmazione

Riassunto dei principali argomenti

Autore:

**Davide Bianchi**

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
<b>2</b>	<b>Esempio di linguaggio basilare</b>	<b>2</b>
2.1	Semantica big-step . . . . .	2
2.1.1	Esempio . . . . .	2
2.2	Semantica small-step . . . . .	3
<b>3</b>	<b>Linguaggio imperativo</b>	<b>3</b>
3.1	Memoria . . . . .	4
3.2	Sistemi di transizione . . . . .	4
3.3	Semantica small-step su un linguaggio imperativo . . . . .	5
3.4	Esecuzione di programmi e proprietà . . . . .	5
3.5	Funzione di valutazione della semantica . . . . .	5
3.6	Possibili varianti del linguaggio . . . . .	6
3.6.1	Inversione dell'ordine di valutazione . . . . .	6
3.6.2	Regole di assegnamento . . . . .	6
3.6.3	Inizializzazione della memoria . . . . .	6
3.6.4	Valori memorizzabili . . . . .	6
3.7	Type systems . . . . .	6
3.7.1	Regole di tipaggio . . . . .	7
3.7.2	Proprietà di tipaggio . . . . .	7
<b>4</b>	<b>Forme di induzione</b>	<b>8</b>
4.1	Induzione matematica . . . . .	8
4.2	Induzione strutturale . . . . .	8
4.2.1	Induzione strutturale su numeri naturali . . . . .	8
4.2.2	Induzione strutturale su strutture complesse . . . . .	8
4.3	Rule induction . . . . .	9
<b>5</b>	<b>Aspetti funzionali</b>	<b>9</b>
<b>6</b>	<b>Dati e memoria variabile</b>	<b>9</b>
<b>7</b>	<b>Sotto-tipaggio</b>	<b>9</b>

## 1 Introduzione

Un linguaggio di programmazione è composto da:

- *Sintassi*: insieme di regole di scrittura del linguaggio;
- *Semantica*: descrizione del comportamento del programma a tempo di esecuzione;
- *Pragmatica*: descrizione delle caratteristiche del linguaggio, delle sue funzionalità ecc.

Gli stili per dare la semantica di un linguaggio sono 3:

- *Operazionale*: la semantica è data con sistemi di transizione, fornendo i passi della computazione passo passo;
- *Denotazionale*: il significato di un programma è dato dalla struttura di un insieme;
- *Assiomatica*: il significato è dato attraverso regole assiomatiche o qualche tipo di logica.

## 2 Esempio di linguaggio basilare

La semantica operativa di un linguaggio è data attraverso un sistema di regole di inferenza, date come segue:

$$(Assioma) \frac{}{(Conclusione)} \quad (Regola) \frac{(Hyp_1) (Hyp_2) \dots (Hyp_n)}{(Conclusione)}$$

Introduciamo la sintassi di un linguaggio basato solo su espressioni aritmetiche:

$$E := n \mid E \mid E + E \mid E * E \dots$$

La valutazione di programmi generati con questa sintassi può procedere in due modi:

- *Small step*: la semantica fornisce il sistema per procedere nell'esecuzione, passo dopo passo;
- *Big step*: la semantica va subito al risultato finale.

### 2.1 Semantica big-step

Forniamo la semantica big-step per il linguaggio dato sopra:

$$\text{B-Num} \frac{}{n \Downarrow n} \quad \text{B-Add} \frac{E_1 \Downarrow n_1 \quad E_2 \Downarrow n_2}{E_1 + E_2 \Downarrow n_3} \quad n_3 = \text{add}(n_1, n_2)$$

La semantica big-step fornisce immediatamente il risultato, dando subito il valore finale dell'espressione che si sta valutando.

#### 2.1.1 Esempio

$$\text{B-Add} \frac{\text{B-Num} \frac{}{3 \Downarrow 3} \quad \text{B-Add} \frac{\text{B-Num} \frac{}{2 \Downarrow 2} \quad \text{B-Num} \frac{}{1 \Downarrow 1}}{2 + 1 \Downarrow 3}}{3 + (2 + 1) \Downarrow 6}$$

**Teorema 2.1** (Determinatezza per semantica big-step).  $E \Downarrow m$  e  $E \Downarrow n$  implica  $m = n$ .

## 2.2 Semantica small-step

Indichiamo con  $E_1 \rightarrow E_2$  lo svolgimento di un solo passo di semantica.

$$\begin{array}{c} \text{S-Left} \frac{E_1 \rightarrow E'_1}{E_1 + E_2 \rightarrow E'_1 + E_2} \\ \text{S-N.Right} \frac{E_2 \rightarrow E'_2}{n_1 + E_2 \rightarrow n_1 + E'_2} \\ \text{S-Add} \frac{-}{n_1 + n_2 \rightarrow n_3} n_3 = \text{add}(n_1, n_2) \end{array}$$

Con queste regole l'ordine di valutazione degli statement è fisso, procede sempre da sinistra verso destra. Diamo un'alternativa:

$$\begin{array}{c} \text{S-Left} \frac{E_1 \rightarrow_{ch} E_2}{E_1 + E_2 \rightarrow_{ch} E'_1 + E_2} \\ \text{S-Right} \frac{E_2 \rightarrow_{ch} E'_2}{E_1 + E_2 \rightarrow_{ch} E_1 + E'_2} \\ \text{S-Add} \frac{-}{n_1 + n_2 \rightarrow_{ch} n_3} n_3 = \text{add}(n_1, n_2) \end{array}$$

In questo caso l'ordine di valutazione è arbitrario. La notazione utilizzata in generale è la seguente:

- la relazione  $\rightarrow^k$ , con  $k \in \mathbb{N}$ , indica una sequenza di  $n$  passi applicando la semantica small-step;
- la relazione  $\rightarrow^*$ , indica una sequenza di derivazione lunga un certo numero di passi. Questa relazione è riflessiva ed è la chiusura transitiva di  $\rightarrow$ .

**Teorema 2.2** (Determinatezza per semantica small-step). *Definiamo:*

- **strong determinacy:**  $E \rightarrow F$  e  $E \rightarrow G$  implica  $F = G$ ;
- **weak determinacy:**  $E \rightarrow^* m$  e  $E \rightarrow^* n$  implica  $m = n$ ;

## 3 Linguaggio imperativo

Definiamo la sintassi di un semplice linguaggio imperativo:

$$\begin{array}{l} b := \text{true} \mid \text{false} \\ n := \{\dots - 1, 0, 1, 2, \dots\} \\ l := \{l_0, l_1, \dots\} \\ op := + \mid \geq \\ e := n \mid b \mid e \text{ op } e \mid \text{if } e \text{ then } e \text{ else } e \mid l := e \mid !l \mid \text{skip} \mid e; e \mid \text{while } e \text{ do } e \end{array}$$

**Nota:** lo statement  $!l$  indica l'intero memorizzato al momento alla locazione  $l$ . Inoltre il linguaggio non è tipato, quindi sono ammesse le sintassi come  $2 \geq \text{true}$ .

### 3.1 Memoria

La memoria è necessaria per poter valutare gli statement di lettura da una locazione. In particolare definiamo

$$\begin{aligned} \text{dom}(f) &= \{a \in A \mid \exists b \in B \text{ s.t. } f(a) = b\} \\ \text{ran}(f) &= \{b \in B \mid \exists a \in A \text{ s.t. } f(a) = b\} \end{aligned}$$

Lo store del linguaggio imperativo in questione è un insieme di funzioni parziali che vanno dalle locazioni di memoria nei numeri interi:

$$s : \mathbb{L} \rightarrow \mathbb{Z}$$

L'aggiornamento della memoria funziona come segue:

$$s[l \rightarrow n](l') = \begin{cases} n & \text{if } l = l' \\ s(l') & \text{altrimenti} \end{cases}$$

### 3.2 Sistemi di transizione

Le semantiche operazionali sono date attraverso sistemi di transizione, ovvero strutture composte da:

- un insieme *Config* di configurazioni;
- una relazione binaria  $\rightarrow \subseteq \text{Config} \times \text{Config}$ ;

Per indicare un generale passo di semantica si usa la notazione

$$\langle e, s \rangle \rightarrow \langle e', s' \rangle$$

che rappresenta una trasformazione di un programma  $e$  con una memoria  $s$  in un programma  $e'$  con memoria associata  $s'$ . I singoli passi di computazione sono singole applicazioni di regole della semantica.

### 3.3 Semantica small-step su un linguaggio imperativo

$$\begin{array}{c}
\text{(op+)} \frac{}{\langle n_1 + n_2, s \rangle \rightarrow \langle n, s \rangle} \quad n = \text{add}(n_1, n_2) \qquad \text{(op-geq)} \frac{}{\langle n_1 \geq n_2, s \rangle \rightarrow \langle b, s \rangle} \quad b = \text{geq}(n_1, n_2) \\
\\
\text{(op1)} \frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \rightarrow \langle e'_1 \text{ op } e_2, s' \rangle} \qquad \text{(op2)} \frac{\langle e_2, s \rangle \rightarrow \langle e'_2, s' \rangle}{\langle v \text{ op } e_2, s \rangle \rightarrow \langle v \text{ op } e'_2, s' \rangle} \\
\\
\text{(deref)} \frac{}{\langle !l, s \rangle \rightarrow \langle n, s \rangle} \quad \text{if } l \in \text{dom}(s) \text{ and } s(l) = n \quad \text{(assign1)} \frac{}{\langle l := n, s \rangle \rightarrow \langle \text{skip}, s[l \rightarrow n] \rangle} \quad \text{if } l \in \text{dom}(s) \\
\\
\text{(assign2)} \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle l := e, s \rangle \rightarrow \langle l := e', s' \rangle} \qquad \text{(if-tt)} \frac{}{\langle \text{if true then } e_1 \text{ else } e_2, s \rangle \rightarrow \langle e_1, s \rangle} \\
\\
\text{(if-ff)} \frac{}{\langle \text{if false then } e_1 \text{ else } e_2, s \rangle \rightarrow \langle e_2, s \rangle} \qquad \text{(if)} \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle \text{if } e \text{ then } e_1 \text{ else } e_2, s \rangle \rightarrow \langle \text{if } e' \text{ then } e_1 \text{ else } e_2, s' \rangle} \\
\\
\text{(seq)} \frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle e_1; e_2, s \rangle \rightarrow \langle e'_1; e_2, s' \rangle} \qquad \text{(seq.skip)} \frac{}{\langle \text{skip}; e_2, s \rangle \rightarrow \langle e_2, s \rangle} \\
\\
\text{(while)} \frac{}{\langle \text{while } e \text{ do } e_1, s \rangle \rightarrow \langle \text{if } e \text{ then } (e_1; \text{while } e \text{ do } e_1) \text{ else skip}, s \rangle}
\end{array}$$

### 3.4 Esecuzione di programmi e proprietà

L'esecuzione di programmi con questa semantica consiste nel trovare una memoria  $s'$  tale per cui valga che

$$\langle P, s \rangle \rightarrow^* \langle v, s' \rangle$$

ovvero che si raggiunga una configurazione terminale in un certo numero di passi.

Illustriamo inoltre due importanti proprietà:

**Teorema 3.1** (Strong normalization). *Per ogni memoria  $s$  e ogni programma  $P$  esiste una qualche memoria  $s'$  tale che*

$$\langle P, s \rangle \rightarrow^* \langle v, s' \rangle$$

**Teorema 3.2** (Determinatezza). *Se  $\langle e, s \rangle \rightarrow \langle e_1, s_1 \rangle$  e  $\langle e, s \rangle \rightarrow \langle e_2, s_2 \rangle$  allora  $\langle e_1, s_1 \rangle = \langle e_2, s_2 \rangle$ .*

### 3.5 Funzione di valutazione della semantica

Date le regole nella sezione 3.3, possiamo dire che in generale, per valutare una porzione di programma, viene applicata la regola

$$\llbracket - \rrbracket : \text{Exp} \rightarrow (\text{Store} \rightarrow \text{Store})$$

dove, data una generica espressione  $e$ , la funzione  $\llbracket \cdot \rrbracket$  prende una memoria e ne ritorna una aggiornata dopo la valutazione di  $e$ .

$$\llbracket e \rrbracket(s) = \begin{cases} s' & \text{se } \langle e, s \rangle \rightarrow \langle e', s' \rangle \\ \text{undefined} & \text{altrimenti} \end{cases}$$

### 3.6 Possibili varianti del linguaggio

Nel linguaggio illustrato possono essere introdotte anche diverse varianti.

#### 3.6.1 Inversione dell'ordine di valutazione

È possibile ad esempio introdurre un ordine di valutazione *right-to-left*, ossia:

$$\text{(op1b)} \frac{\langle e_2, s \rangle \rightarrow \langle e'_2, s' \rangle}{\langle e_1 + e_2, s \rangle \rightarrow \langle e_1 + e'_2, s' \rangle} \quad \text{(op2b)} \frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle e_1 + v, s \rangle \rightarrow \langle e'_1 + v, s' \rangle}$$

Aggiungendo queste due regole alla semantica ovviamente salta la regola della determinatezza.

#### 3.6.2 Regole di assegnamento

Una piccola variante alla regola dell'assegnamento:

$$\text{(assign1b)} \frac{}{\langle l := n, s \rangle \rightarrow \langle n, s[l \rightarrow n] \rangle} \text{ if } l \in \text{dom}(s) \quad \text{(seq.skip.b)} \frac{}{\langle v; e_2, s \rangle \rightarrow \langle e_2, s \rangle}$$

#### 3.6.3 Inizializzazione della memoria

Possibili varianti a livello di inizializzazione della memoria potrebbero essere:

- inizializzare implicitamente tutte le locazioni a 0;
- permettere assegnamenti ad una locazione  $l$  tale che  $l \notin \text{dom}(s)$  per inizializzare quella locazione.

#### 3.6.4 Valori memorizzabili

Altre estensioni relative alla memoria (qui definita staticamente, ovvero l'insieme delle locazioni possibili è fisso) possono includere:

- la possibilità di memorizzare anche altri tipi di dato (non solo interi come in questo caso);
- la possibilità di avere una memoria definita dinamicamente, quindi dare la possibilità di avere sempre nuove locazioni disponibili oltre a quelle già in uso.

### 3.7 Type systems

Un type system è una struttura i cui usi principali sono:

- descrivere quando i programmi sono sensati;
- prevenire certi tipi di errore;
- strutturare i programmi;
- dare delle linee guida per la progettazione del linguaggio;
- dare informazioni utili per la fase di ottimizzazione da parte del compilatore;

- rinforzare alcune proprietà di *sicurezza* del programma.

Definiamo la funzione

$$\Gamma \vdash e : T$$

che sostanzialmente assegna il tipo  $T$  all'espressione  $e$ , per qualche tipo  $T$  del linguaggio.

Aggiungiamo al linguaggio i tipi delle espressioni  $T$  e i tipi delle locazioni  $T_{loc}$ :

$$T ::= int \mid bool \mid unit$$

$$T_{loc} ::= intref$$

### 3.7.1 Regole di tipaggio

$$\begin{array}{ll}
\text{(int)} \frac{}{\Gamma \vdash n : int} \text{ per } n \in \mathbb{Z} & \text{(bool)} \frac{}{\Gamma \vdash b : bool} \text{ per } b \in \{true, false\} \\
\text{(op+)} \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int} & \text{(op-geq)} \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \geq e_2 : bool} \\
\text{(if)} \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} & \text{(assign)} \frac{\Gamma \vdash e : int}{\Gamma \vdash l := e : unit} \text{ se } \Gamma(l) = intref \\
\text{(deref)} \frac{}{\Gamma \vdash !l : int} \text{ se } \Gamma(l) = intref & \text{(skip)} \frac{}{\Gamma \vdash skip : unit} \\
\text{(seq)} \frac{\Gamma \vdash e_1 : unit \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1; e_2 : T} & \text{(while)} \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : unit}{\Gamma \vdash \text{while } e_1 \text{ do } e_2 : unit}
\end{array}$$

**Nota:** le regole di tipaggio sono *syntax-directed*, ovvero per ogni regola della sintassi astratta si ha una regola di tipaggio.

### 3.7.2 Proprietà di tipaggio

**Teorema 3.3** (Progress). *Se  $\Gamma \vdash e : T$  e  $dom(\Gamma) \subseteq dom(s)$  allora  $e$  è un valore oppure esiste una coppia  $\langle e', s' \rangle$  tale che*

$$\langle e, s \rangle \rightarrow \langle e', s' \rangle$$

**Teorema 3.4** (Type preservation). *Se  $\Gamma \vdash e : T$  e  $dom(\Gamma) \subseteq dom(s)$  e  $\langle e, s \rangle \rightarrow \langle e', s' \rangle$  allora si ha che  $\Gamma \vdash e' : T$  e  $dom(\Gamma) \subseteq dom(s')$*

Mettendo insieme le due proprietà sopra, si ottiene una nuova proprietà, esplicativa del fatto che programmi ben tipati non vanno mai in deadlock.

**Teorema 3.5** (Safety). *Se  $\Gamma \vdash e : T$ ,  $dom(\Gamma) \subseteq dom(s)$  e  $\langle e, s \rangle \rightarrow^* \langle e', s' \rangle$  allora  $e'$  è un valore oppure esiste una coppia  $\langle e'', s'' \rangle$  tale che  $\langle e', s' \rangle \rightarrow \langle e'', s'' \rangle$*

**Teorema 3.6** (Type inference). *Dati  $\Gamma, e$ , può essere trovato il tipo  $T$  tale che  $\Gamma \vdash e : T$  oppure può essere provato che  $T$  non esiste.*

**Teorema 3.7** (Decidibilità del type-checking). *Dati  $\Gamma, e, T$ , è decidibile  $\Gamma \vdash e : T$*

**Teorema 3.8** (Unicità del tipaggio). *Se vale che  $\Gamma \vdash e : T$  e  $\Gamma \vdash e : T'$  allora  $T = T'$ .*



## 4 Forme di induzione

L'induzione è una tecnica formale che consente di provare delle proprietà su determinate categorie di oggetti, sfruttando la natura di questi oggetti. Esistono 3 tipi di induzione:

- matematica;
- strutturale;
- rule induction<sup>2</sup>.

### 4.1 Induzione matematica

È la forma di induzione più semplice, consiste infatti nel dimostrare una proprietà  $P(-)$  su numeri naturali procedendo nel modo seguente:

1. **Caso base:** provare che  $P(0)$  è vera, usando qualche procedimento matematico;
2. **Caso induttivo:**
  - (a) assumere che l'ipotesi induttiva valga, ovvero che valga  $P(k)$ ;
  - (b) dall'ipotesi induttiva dimostrare che vale  $P(k + 1)$ , usando qualche procedimento matematico.

Se i punti precedenti sono veri, allora  $P(n)$  è vera per ogni numero naturale.

### 4.2 Induzione strutturale

#### 4.2.1 Induzione strutturale su numeri naturali

Per dimostrare una proprietà  $P$  su numeri naturali basta applicare il seguente metodo:

- **Caso base:** dimostrare che vale  $P(0)$ ;
- **Caso induttivo:** dimostrare che è vera  $P(\text{succ}(K))$  assumendo come ipotesi induttiva che valga  $P(K)$  per qualche  $K \in \mathbb{N}$ .

L'induzione strutturale consiste quindi nell'assumere che l'ipotesi induttiva valga per la *sottostruttura* di  $\text{succ}(K)$ .

#### 4.2.2 Induzione strutturale su strutture complesse

Prendiamo come esempio la costruzione di alberi binari. Diamo la seguente grammatica per costruire gli alberi:

$$T ::= \text{leaf} \mid \text{tree}(T, T)$$

In tal caso partiamo col presupposto che:

- **Caso base:** una foglia sia un albero binario;
- **Caso induttivo:** se  $L$  e  $R$  sono alberi binari, allora lo è anche  $\text{tree}(L, R)$ .

### 4.3 Rule induction

L'idea di base della rule induction consiste nell'ignorare la struttura di ciò che si deriva per fare induzione sulla dimensione dell'albero di derivazione.

---

<sup>2</sup>Appena avrò una traduzione valida la metterò.

**5 Aspetti funzionali**

**6 Dati e memoria variabile**

**7 Sotto-tipaggio**