Elementi di Semantica Operazionale

Appunti per gli studenti di Fondamenti di Programmazione
(corsi A-B-C)

Corso di Laurea in Informatica
Università di Pisa
A.A. 2004/05

R. Barbuti, P. Mancarella e F. Turini

Indice

Premessa

Gli studenti sono invitati a leggere queste note con criticità ed attenzione, per la potenziale presenza di refusi ed imprecisioni che possono essere rimasti nel testo. Si ringraziano tutti coloro che vorranno segnalarli; commenti, suggerimenti e critiche riguardanti il materiale o la sua presentazione sono particolarmente benvenuti.

Il materiale di queste note nasce dallo sforzo congiunto di docenti, assistenti e studenti dei corsi di Teoria ed Applicazione delle Macchine Calcolatrici (TAMC) e di Programmazione I dei Corsi di Laurea in Scienze dell'Informazione e in Informatica, per aggiornare e migliorare il corso. Per il loro contributo in tal senso, nonché per le molte osservazioni sul testo, ringraziamo Antonio Brogi, Paola Cappanera, Andrea Corradini, Gianluigi Ferrari, Paola Inverardi, Carlo Montangero, Monica Nesi, Dino Pedreschi, Alessandra Raffaetà e gli studenti dei corsi di TAMC e Programmazione I degli Anni Accademici precedenti.

1 Introduzione

I linguaggi di programmazione sono linguaggi artificiali che servono per esprimere algoritmi, ovvero procedimenti risolutivi di problemi, in modo che un elaboratore automatico sia poi in grado di eseguirli. Un *programma* non è altro che la traduzione di un algoritmo in un particolare linguaggio di programmazione.

La definizione formale dei linguaggi, artificiali e non, prevede essenzialmente due aspetti: una descrizione della *sintassi* del linguaggio, ovvero delle frasi legali esprimibili nel linguaggio stesso; e una descrizione della *semantica* del linguaggio, ovvero del significato di frasi sintatticamente corrette. Nel caso dei linguaggi di programmazione, la sintassi descrive la struttura di un programma, ovvero in che modo frasi e simboli di base possono essere composti al fine di ottenere un programma legale nel linguaggio. A differenza dei linguaggi naturali, la sintassi dei linguaggi di programmazione è relativamente semplice e può essere descritta in modo rigoroso utilizzando i formalismi messi a disposizione dalla teoria dei linguaggi formali. Non è scopo di queste note, né del corso di Fondamenti di Programmazione, affrontare lo studio di tale teoria: nel seguito daremo per scontati i pochi concetti di base sulla sintassi dei linguaggi introdotti nella prima parte del corso.

Nella storia, peraltro recente, dei linguaggi di programmazione vi è un evidente contrasto tra il modo in cui ne viene descritta la sintassi e quello in cui ne viene descritta la semantica. Di solito una descrizione rigorosa e formale della sintassi è affiancata da una descrizione poco rigorosa e del tutto informale della semantica: i manuali di riferimento dei linguaggi di programmazione più comuni testimoniano questa situazione. Le descrizioni informali sono spesso incomplete e poco precise, dando luogo ad incomprensioni da parte di chi ne fa uso, vuoi per capire come utilizzare un dato linguaggio nella risoluzione di problemi, vuoi per affrontare il problema della realizzazione (o implementazione) di un dato linguaggio, vuoi per ragionare in modo astratto sulla correttezza dei programmi scritti in un dato linguaggio.

A tali carenze si è cercato di sopperire introducendo metodi e tecniche formali per la definizione della semantica dei linguaggi di programmazione. Scopo di queste note è introdurre il lettore ad una di esse che va sotto il nome di *semantica operazionale*. L'idea che sta alla base di tale approccio è di dare la semantica di un linguaggio mediante la definizione di un *sistema* e del suo comportamento in corrispondenza di programmi scritti nel linguaggio.

Una prima importante osservazione da fare è che nella descrizione della semantica di un linguaggio si fa riferimento alla sua *sintassi astratta*, intendendo con essa una descrizione sintattica che pone in rilievo la struttura sintattica essenziale dei vari costrutti del linguaggio, prescindendo da dettagli che non contribuiscono a chiarire il significato delle frasi, anche se rilevanti sotto altri punti di vista. Un modo per descrivere la sintassi astratta dei vari costrutti è attraverso i cosiddetti *alberi di derivazione*, in cui è specificata proprio la struttura, sotto forma di albero, dei costrutti stessi. Tuttavia, per non appesantire la notazione, presenteremo sia le produzioni della grammatica utilizzata sia i vari costrutti sintattici sotto forma di stringhe di simboli, ma dovrà essere sempre chiaro al lettore che esse rappresentano in realtà l'albero di derivazione per la stringa stessa.

2 I sistemi di transizioni

In questo paragrafo introduciamo i concetti che sono alla base dell'approccio *operazionale* alla semantica dei linguaggi e diamo la semantica in questo stile di un semplice linguaggio per la descrizione di espressioni.

Definizione 2.1 Un sistema di transizioni $\mathtt{S}$ è una tripla $\langle \Gamma, \mathtt{T}, \rightarrow \rangle$ dove

- Γ è un insieme i cui elementi sono detti *configurazioni*;
- $\mathtt{T} \subseteq \Gamma$ è un sottoinsieme di Γ i cui elementi sono detti *configurazioni terminali*;
- $\rightarrow$ è un insieme di coppie $\langle \gamma, \gamma' \rangle$ di configurazioni e viene detta *relazione di transizione*. Denoteremo con $\gamma \rightarrow \gamma'$ l'appartenenza della coppia $\langle \gamma, \gamma' \rangle$ alla relazione $\rightarrow$ e chiameremo *transizioni* tali elementi. $\square$

In prima approssimazione possiamo pensare alle configurazioni come agli *stati* in cui il sistema si può trovare durante il suo funzionamento: in particolare le configurazioni terminali corrispondono a quegli stati in cui il sistema ha terminato le sue operazioni. La relazione di transizione descrive il comportamento del sistema: il significato intuitivo di una transizione $\gamma \rightarrow \gamma'$ è che il sistema dallo stato γ è in grado di portarsi nello stato γ'. In quest'ottica, il comportamento del sistema a partire da una certa configurazione è formalizzato nella seguente definizione.

Definizione 2.2 Dato un sistema di transizioni $\mathtt{S} = \langle \Gamma, \mathtt{T}, \rightarrow \rangle$, una *derivazione* in $\mathtt{S}$ è una sequenza (eventualmente infinita) di configurazioni $\gamma_0, \gamma_1, \ldots, \gamma_{i-1}, \gamma_i, \ldots$ tale che, per ogni $k \geq 1$, $\gamma_{k-1} \rightarrow \gamma_k$. Nel seguito, una derivazione in $\mathtt{S}$ verrà indicata con $\gamma_0 \rightarrow \gamma_1 \rightarrow \ldots \rightarrow \gamma_i$. Inoltre indicheremo con $\gamma \stackrel{*}{\rightarrow} \gamma'$ l'esistenza di una derivazione $\gamma_0 \rightarrow \gamma_1 \rightarrow \ldots \rightarrow \gamma_i$ in cui $\gamma_0 = \gamma$ e $\gamma_i = \gamma'$. □

Dato un sistema di transizioni $\langle \Gamma, \mathtt{T}, \rightarrow \rangle$, diremo che una configurazione $\gamma \in \Gamma \setminus \mathtt{T}$ è *bloccata* se e solo se non esiste alcuna configurazione γ' tale che $\gamma \rightarrow \gamma'$.

Come primi esempi, consideriamo grammatiche ed automi a stati finiti e costruiamo dei sistemi di transizioni ad essi corrispondenti.

Esempio 2.3 Sia $G = \langle \Lambda, V, S, P \rangle$ una grammatica libera da contesto. Definiamo un sistema di transizioni S_G associato alla grammatica nel seguente modo.

- le configurazioni di S_G sono stringhe in $(\Lambda \cup V)^*$;
- le configurazioni terminali di S_G sono le stringhe in Λ^*;
- dette α e β due configurazioni (stringhe), esiste una transizione $\alpha \rightarrow_G \beta$ se e solo se
  - α è del tipo $\delta A \gamma$, con $\delta, \gamma \in (\Lambda \cup V)^*$ e $A \in V$;
  - β è del tipo $\delta \eta \gamma$;
  - $A ::= \eta$ è una produzione di P.

È evidente, per costruzione di S_G, che $\alpha \in \mathcal{L}(G)$ se e solo se $S \stackrel{*}{\rightarrow}_G \alpha$, dove $\mathcal{L}(G)$ indica il linguaggio generato da G. ■

Nel seguito utilizzeremo sistemi di transizioni in cui la relazione di transizione è spesso espressa mediante regole *condizionali*, ovvero regole del tipo:

$$\frac{\mathtt{C}_1 \quad \mathtt{C}_2 \ldots \mathtt{C}_n}{\gamma \rightarrow \gamma'}$$

dove $\mathtt{C}_1 \quad \mathtt{C}_2 \ldots \mathtt{C}_n$ (dette *precondizioni* o *premesse* della regola) sono i prerequisiti alla transizione $\gamma \rightarrow \gamma'$. Così, ad esempio, la definizione di $\rightarrow_G$ dell'esempio 2.3 può essere data come segue. Per ogni produzione $(A ::= \eta) \in P$ si definisce una regola condizionale:

$$\frac{\alpha = \delta A \gamma \qquad \delta, \gamma \in (\Lambda \cup V)^* \qquad \beta = \delta \eta \gamma}{\alpha \rightarrow_G \beta}$$

Tale regola può essere data in modo equivalente come:

$$\frac{\delta, \gamma \in (\Lambda \cup V)^*}{\delta A \gamma \rightarrow_G \delta \eta \gamma}$$

Si consideri ad esempio la grammatica definita dalle seguenti produzioni:

```
S ::= ab | aSb
```

che genera il linguaggio $\{\mathtt{a}^n\mathtt{b}^n \mid n > 0\}$. Le regole del corrispondente sistema di transizioni sono:

$$\frac{\delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^*}{\delta\mathtt{S}\gamma \to \delta\mathtt{ab}\gamma}$$

$$\frac{\delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^*}{\delta\mathtt{S}\gamma \to \delta\mathtt{aSb}\gamma}$$

Nell'esempio precedente, la relazione di transizione del sistema S_G corrisponde esattamente alla nozione di *derivazione* nella grammatica, nel senso che $\alpha \to_G \beta$ se e solo se β è ottenibile da α riscrivendo un simbolo non terminale di α con la parte destra di una produzione per quel simbolo. Nell'esempio che segue, vediamo come definire un nuovo sistema di transizioni a partire da una grammatica, in cui però la relazione di transizione corrisponde a riscrivere sempre il simbolo non terminale più a sinistra.

Esempio 2.4 Sia $G = \langle \Lambda, V, S, P \rangle$ una grammatica libera da contesto e sia S'_G il sistema di transizioni così definito.

- le configurazioni di S'_G sono stringhe in $(\Lambda \cup V)^*$;
- le configurazioni terminali di S'_G sono le stringhe in Λ^*;
- la relazione di transizione $\rightsquigarrow_G$ è definita dalle seguenti regole condizionali, una per ogni produzione $(A ::= \eta) \in P$:

$$\boxed{\frac{\delta \in \Lambda^* \qquad \gamma \in (\Lambda \cup V)^*}{\delta A \gamma \rightsquigarrow_G \delta\eta\gamma}}$$

Anche in questo caso è facile convincersi che $\alpha \in \mathcal{L}(G)$ se e solo se $S \rightsquigarrow^*_G \alpha$. ■

Gli automi a stati finiti, utilizzati per la descrizione di linguaggi regolari, possono essere visti come un caso particolare di sistemi di transizioni. L'esempio che segue mostra come, a partire da un automa a stati finiti, è possibile costruire un sistema di transizioni che ne simula il comportamento.

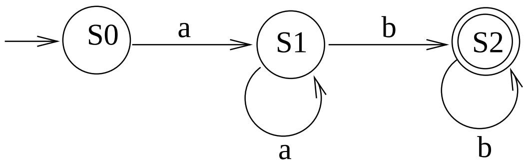
Esempio 2.5 Sia Λ un alfabeto e sia $\mathcal{A}$ un automa a stati finiti su Λ. Siano inoltre $\mathcal{S}$ l'insieme degli stati di $\mathcal{A}$ e $\mathcal{F} \subseteq \mathcal{S}$ l'insieme degli stati di riconoscimento di $\mathcal{A}$. Indichiamo con $s \stackrel{a}{\Longrightarrow} t$ la presenza in $\mathcal{A}$ di un arco, etichettato con a, uscente dallo stato s ed entrante nello stato t. Costruiamo un sistema di transizioni $S_\mathcal{A}$ che corrisponde all'automa, nel seguente modo:

- le configurazioni di $S_\mathcal{A}$ sono coppie del tipo $\langle s, \alpha \rangle$ con $s \in \mathcal{S}$ e $\alpha \in \Lambda^*$
- le configurazioni terminali di $S_\mathcal{A}$ sono le coppie $\langle s, \epsilon \rangle$, con $s \in \mathcal{F}$ (ricordiamo che ϵ denota la stringa vuota).
- la relazione di transizione $\rightsquigarrow_\mathcal{A}$ è data dalle seguenti regole condizionali, una per ogni arco $s \stackrel{a}{\Longrightarrow} t$ dell'automa:

$$\boxed{\frac{\alpha \in \Lambda^*}{\langle s, a\alpha \rangle \rightsquigarrow_\mathcal{A} \langle t, \alpha \rangle}}$$

Anche in questo caso è facile convincersi che una stringa α appartiene al linguaggio riconosciuto da $\mathcal{A}$ se e solo se, detto s_0 lo stato iniziale dell'automa, $\langle s_0, \alpha \rangle \rightsquigarrow^*_\mathcal{A} \langle t, \epsilon \rangle$, con $\langle t, \epsilon \rangle$ configurazione terminale.

Consideriamo come esempio concreto l'automa rappresentato dalla seguente figura:



che riconosce il linguaggio $\{a^n b^m \mid n, m > 0\}$. Le regole del corrispondente sistema di transizioni sono:

$$\frac{\alpha \in \Lambda^*}{\langle s_0, \mathtt{a}\alpha \rangle \rightarrow \langle s_1, \alpha \rangle}$$

$$\frac{\alpha \in \Lambda^*}{\langle s_1, \mathtt{a}\alpha \rangle \rightarrow \langle s_1, \alpha \rangle}$$

$$\frac{\alpha \in \Lambda^*}{\langle s_1, \mathtt{b}\alpha \rangle \rightarrow \langle s_2, \alpha \rangle}$$

$$\frac{\alpha \in \Lambda^*}{\langle s_2, \mathtt{b}\alpha \rangle \rightarrow \langle s_2, \alpha \rangle}$$

■

2.1 Un sistema di transizioni per la somma di numeri naturali

Vediamo un esempio di sistema di transizioni per il calcolo della somma tra numeri naturali. Questi ultimi sono rappresentati dalla seguente grammatica:

```
Num ::= Cif | Num Cif
Cif ::= 0 | 1 | 2 | ...  | 9
```

Nella definizione del sistema di transizioni utilizziamo un approccio di tipo gerarchico: definiamo cioè, dapprima, le transizioni di un *sottosistema* che definisce la somma tra cifre decimali con riporto e successivamente quelle del sistema per la somma di due numeri arbitrari. Quest'ultimo farà uso del sottosistema già definito laddove ciò risulti utile. Definiamo quindi due sistemi di transizioni:

- $\mathtt{S}_{cr}$, che definisce la somma tra cifre decimali con riporto; la relazione di transizione di questo sottosistema è rappresentata dal simbolo $\rightarrow_{cr}$;
- $\mathtt{S}_{n+}$, che definisce la somma tra numeri decimali; la relazione di transizione di questo sottosistema è rappresentata dal simbolo $\rightarrow_{n+}$.

Nel seguito useremo le seguenti convenzioni:

- $\mathtt{c}$, $\mathtt{c}'$, $\mathtt{c}''$, ... stanno per generiche cifre decimali, ovvero elementi della categoria sintattica `Cif`;
- $\mathtt{r}$, $\mathtt{r}'$, $\mathtt{r}''$, ... stanno per elementi dell'insieme $\{\mathtt{0},\mathtt{1}\}$, i riporti derivanti dalla somma tra due cifre;
- $\mathtt{n}$, $\mathtt{n}'$, $\mathtt{m}$, ... stanno per generici numeri decimali, ovvero elementi della categoria sintattica `Num`.

Per rendere più snella la trattazione, indicheremo con $\mathtt{x} \in \mathtt{T}$ il fatto che la stringa `x` appartiene al linguaggio associato alla categoria sintattica `T`: scriveremo, ad esempio, $\mathtt{0} \in \mathtt{Cif}$, $\mathtt{n} \in \mathtt{Num}$ e così via.

2.1.1 $\mathtt{S}_{cr}$: somma tra cifre decimali con riporto

Le configurazioni del sistema di transizioni $\mathtt{S}_{cr}$ per la somma di due cifre sono di due tipi:

- $\langle \mathtt{c}, \mathtt{c}', \mathtt{r} \rangle$, che rappresentano lo stato in cui il sistema si appresta a sommare le due cifre $\mathtt{c}$ e $\mathtt{c}'$ e il riporto $\mathtt{r}$;
- $\langle \mathtt{c}, \mathtt{r} \rangle$, che rappresentano la cifra ed il riporto risultante dalla somma di due cifre. Sono queste, dunque, tutte e sole le configurazioni terminali.

Più formalmente:

$$\begin{array}{rcl} \Gamma_{cr} & = & \{\langle \mathtt{c}, \mathtt{c}', \mathtt{r} \rangle \mid \mathtt{r} \in \{\mathtt{0}, \mathtt{1}\} \text{ e } \mathtt{c}, \mathtt{c}' \in \mathtt{Cif}\} \cup \\ & & \{\langle \mathtt{c}, \mathtt{r} \rangle \mid \mathtt{r} \in \{\mathtt{0}, \mathtt{1}\} \text{ e } \mathtt{c} \in \mathtt{Cif}\} \\ \\ \mathtt{T}_{cr} & = & \{\langle \mathtt{c}, \mathtt{r} \rangle \mid \mathtt{r} \in \{\mathtt{0}, \mathtt{1}\} \text{ e } \mathtt{c} \in \mathtt{Cif}\}. \end{array}$$

La relazione di transizione è data dalle seguenti regole:

$$\boxed{\begin{array}{ll} \langle \mathtt{0}, \mathtt{0}, \mathtt{0} \rangle \to_{cr} \langle \mathtt{0}, \mathtt{0} \rangle & \langle \mathtt{0}, \mathtt{0}, \mathtt{1} \rangle \to_{cr} \langle \mathtt{1}, \mathtt{0} \rangle \\ \langle \mathtt{0}, \mathtt{1}, \mathtt{0} \rangle \to_{cr} \langle \mathtt{1}, \mathtt{0} \rangle & \langle \mathtt{0}, \mathtt{1}, \mathtt{1} \rangle \to_{cr} \langle \mathtt{2}, \mathtt{0} \rangle \\ \langle \mathtt{0}, \mathtt{2}, \mathtt{0} \rangle \to_{cr} \langle \mathtt{2}, \mathtt{0} \rangle & \langle \mathtt{0}, \mathtt{2}, \mathtt{1} \rangle \to_{cr} \langle \mathtt{3}, \mathtt{0} \rangle \\ \quad \dots \qquad \dots & \quad \dots \qquad \dots \\ \langle \mathtt{9}, \mathtt{8}, \mathtt{0} \rangle \to_{cr} \langle \mathtt{7}, \mathtt{1} \rangle & \langle \mathtt{9}, \mathtt{8}, \mathtt{1} \rangle \to_{cr} \langle \mathtt{8}, \mathtt{1} \rangle \\ \langle \mathtt{9}, \mathtt{9}, \mathtt{0} \rangle \to_{cr} \langle \mathtt{8}, \mathtt{1} \rangle & \langle \mathtt{9}, \mathtt{9}, \mathtt{1} \rangle \to_{cr} \langle \mathtt{9}, \mathtt{1} \rangle \end{array}}$$

Il "calcolo", mediante il sistema appena definito, della somma di due cifre $\mathtt{c}$ e $\mathtt{c}'$ e di un riporto $\mathtt{r}$ corrisponde ad individuare una derivazione che porta la configurazione *iniziale* $\langle \mathtt{c}, \mathtt{c}', \mathtt{r} \rangle$ in una configurazione terminale $\langle \mathtt{c}'', \mathtt{r}' \rangle$. Si noti che, per come è definita $\to_{cr}$, tali derivazioni sono di lunghezza unitaria.

2.1.2 $\mathtt{S}_{n+}$: somma tra numeri decimali

Le configurazioni del sistema di transizioni $\mathtt{S}_{n+}$ per la somma di due numeri decimali sono di tre tipi:

- $\langle \mathtt{n}, \mathtt{n}' \rangle$, che rappresentano lo stato in cui il sistema si appresta a calcolare la somma tra $\mathtt{n}$ e $\mathtt{n}'$ e che chiameremo configurazioni *iniziali*;
- $\langle \mathtt{n}, \mathtt{n}', \mathtt{m}, \mathtt{r} \rangle$ che rappresentano uno stadio intermedio del calcolo in cui il sistema si appresta a sommare i numeri $\mathtt{n}$ e $\mathtt{n}'$ ed il riporto $\mathtt{r}$, avendo calcolato il risultato parziale $\mathtt{m}$;
- $\mathtt{n}$, che rappresentano il risultato finale del calcolo; queste sono dunque tutte e sole le configurazioni terminali.

Più formalmente:

$$\begin{array}{rcl} \Gamma_{n+} & = & \{\langle \mathtt{n}, \mathtt{n}' \rangle \mid \mathtt{n}, \mathtt{n}' \in \mathtt{Num}\} \cup \\ & & \{\langle \mathtt{n}, \mathtt{n}', \mathtt{m}, \mathtt{r} \rangle \mid \mathtt{r} \in \{\mathtt{0}, \mathtt{1}\} \text{ e } \mathtt{n}, \mathtt{n}', \mathtt{m} \in \mathtt{Num}\} \cup \\ & & \{\mathtt{n} \mid \mathtt{n} \in \mathtt{Num}\} \\ \mathtt{T}_{n+} & = & \{\mathtt{n} \mid \mathtt{n} \in \mathtt{Num}\} \end{array}$$

Il sistema di transizioni $\mathtt{S}_{n+}$ utilizza $\mathtt{S}_{cr}$ come sottosistema, e ciò è reso esplicito dal fatto che alcune transizioni in $\to_{n+}$ hanno come prerequisito una derivazione in $\to_{cr}$. In effetti, le transizioni di $\to_{n+}$ corrispondono al calcolo della somma tra numeri che abbiamo imparato fin dalle scuole elementari, che prevede la somma di due cifre e di un riporto come operazione "elementare". Consideriamo come esempio il procedimento che un alunno delle scuole elementari segue per effettuare la somma di $\mathtt{54}$ e $\mathtt{38}$:

- stadio iniziale del calcolo

```
 54
 38
```

A questa situazione corrisponde la configurazione iniziale $\langle \texttt{54}, \texttt{38} \rangle$.

- stadio intermedio

  ```
  1
  54
  38
 ----
   2
  ```

  poiché `4+8=2` con riporto `1`.
  In questa situazione intermedia il calcolo ha prodotto un risultato parziale, `2`, ed un riporto, `1`, sommando le prime due cifre dei numeri di partenza; queste ultime, di qui in poi, possono essere semplicemente ignorate. A questa situazione corrisponde la transizione $\langle \texttt{54}, \texttt{38} \rangle \rightarrow_{n+} \langle \texttt{5}, \texttt{3}, \texttt{2}, \texttt{1} \rangle$, dove `5` e `3` rappresentano i numeri ancora da sommare, `2` il risultato intermedio e `1` il riporto.

- stadio intermedio

  ```
  0
   54
   38
  ----
   92
  ```

  poiché `5+3+1=9` con riporto `0`.
  A questa situazione corrisponde la transizione $\langle \texttt{5}, \texttt{3}, \texttt{2}, \texttt{1} \rangle \rightarrow_{n+} \langle \texttt{0}, \texttt{0}, \texttt{92}, \texttt{0} \rangle$.

- risultato della somma: `92`.
  Non essendoci più cifre da sommare, il risultato è `92`. A questa situazione corrisponde la transizione $\langle \texttt{0}, \texttt{0}, \texttt{92}, \texttt{0} \rangle \rightarrow_{n+} \texttt{92}$.

Si noti come, nei due stadi intermedi del calcolo, abbiamo assunto che la somma di due cifre e di un riporto sia nota all'alunno. Nel caso del sistema di transizioni, ciò corrisponde ad assumere che derivazioni in $\rightarrow_{cr}$ possano essere prerequisiti per le transizioni di $\rightarrow_{n+}$. Formalmente ciò può essere espresso mediante regole condizionali. Ad esempio, la transizione corrispondente al primo dei due stadi intermedi, si ottiene grazie alla regola condizionale:

$$\boxed{\dfrac{\langle \texttt{4}, \texttt{8}, \texttt{0} \rangle \rightarrow_{cr} \langle \texttt{2}, \texttt{1} \rangle}{\langle \texttt{54}, \texttt{38} \rangle \rightarrow_{n+} \langle \texttt{5}, \texttt{3}, \texttt{2}, \texttt{1} \rangle}}$$

Si noti come la premessa sia definita in termini del (sotto-)sistema di transizioni $\texttt{S}_{cr}$.

Le transizioni di $\rightarrow_{n+}$ sono date qui di seguito:

$$\dfrac{\langle \texttt{c}, \texttt{c}', \texttt{0} \rangle \rightarrow_{cr} \langle \texttt{c}'', \texttt{r} \rangle}{\langle \texttt{c}, \texttt{c}' \rangle \rightarrow_{n+} \langle \texttt{0}, \texttt{0}, \texttt{c}'', \texttt{r} \rangle} \qquad \text{(i)}$$

$$\dfrac{\langle \texttt{c}, \texttt{c}', \texttt{0} \rangle \rightarrow_{cr} \langle \texttt{c}'', \texttt{r} \rangle}{\langle \texttt{nc}, \texttt{c}' \rangle \rightarrow_{n+} \langle \texttt{n}, \texttt{0}, \texttt{c}'', \texttt{r} \rangle} \qquad \text{(ii)}$$

$$\dfrac{\langle \texttt{c}', \texttt{c}, \texttt{0} \rangle \rightarrow_{cr} \langle \texttt{c}'', \texttt{r} \rangle}{\langle \texttt{c}', \texttt{nc} \rangle \rightarrow_{n+} \langle \texttt{0}, \texttt{n}, \texttt{c}'', \texttt{r} \rangle} \qquad \text{(iii)}$$

$$\frac{\langle \mathtt{c}, \mathtt{c}', \mathtt{0}\rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r}\rangle}{\langle \mathtt{nc}, \mathtt{n}'\mathtt{c}'\rangle \rightarrow_{n+} \langle \mathtt{n}, \mathtt{n}', \mathtt{c}'', \mathtt{r}\rangle} \tag{iv}$$

$$\langle \mathtt{0}, \mathtt{0}, \mathtt{m}, \mathtt{0}\rangle \rightarrow_{n+} \mathtt{m} \tag{v}$$

$$\frac{\langle \mathtt{c}, \mathtt{c}', \mathtt{r}\rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r}'\rangle}{\langle \mathtt{c}, \mathtt{c}', \mathtt{m}, \mathtt{r}\rangle \rightarrow_{n+} \langle \mathtt{0}, \mathtt{0}, \mathtt{c}''\mathtt{m}, \mathtt{r}'\rangle} \quad \text{se } \langle \mathtt{c}, \mathtt{c}', \mathtt{r}\rangle \neq \langle \mathtt{0}, \mathtt{0}, \mathtt{0}\rangle \tag{vi}$$

$$\frac{\langle \mathtt{c}, \mathtt{c}', \mathtt{r}\rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r}'\rangle}{\langle \mathtt{nc}, \mathtt{c}', \mathtt{m}, \mathtt{r}\rangle \rightarrow_{n+} \langle \mathtt{n}, \mathtt{0}, \mathtt{c}''\mathtt{m}, \mathtt{r}'\rangle} \tag{vii}$$

$$\frac{\langle \mathtt{c}', \mathtt{c}, \mathtt{r}\rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r}'\rangle}{\langle \mathtt{c}', \mathtt{nc}, \mathtt{m}, \mathtt{r}\rangle \rightarrow_{n+} \langle \mathtt{0}, \mathtt{n}, \mathtt{c}''\mathtt{m}, \mathtt{r}'\rangle} \tag{viii}$$

$$\frac{\langle \mathtt{c}, \mathtt{c}', \mathtt{r}\rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r}'\rangle}{\langle \mathtt{nc}, \mathtt{n}'\mathtt{c}', \mathtt{m}, \mathtt{r}\rangle \rightarrow_{n+} \langle \mathtt{n}, \mathtt{n}', \mathtt{c}''\mathtt{m}, \mathtt{r}'\rangle} \tag{ix}$$

Osserviamo come le precedenti regole sono in realtà *schemi di regole*: ogni schema rappresenta una collezione di regole (che chiameremo *istanze* della regola), ciascuna ottenuta rimpiazzando i simboli dello schema con particolari valori. Ad esempio, (ii) rappresenta, tra le altre, le regole:

$$\frac{\langle 9, 5, 0\rangle \rightarrow_{cr} \langle 4, 1\rangle}{\langle 379, 5\rangle \rightarrow_{n+} \langle 37, 0, 4, 1\rangle}$$

$$\frac{\langle 2, 4, 0\rangle \rightarrow_{cr} \langle 6, 0\rangle}{\langle 82, 4\rangle \rightarrow_{n+} \langle 8, 0, 6, 0\rangle}$$

Le transizioni (i) $\div$ (iv) corrispondono ai passi iniziali del calcolo in cui i numeri da sommare sono costituiti: entrambi da una sola cifra, il primo da più cifre e il secondo da una sola cifra, il primo da una sola cifra ed il secondo da più cifre ed entrambi da più cifre, rispettivamente. La transizione (v) corrisponde invece al passo terminale del calcolo: la configurazione $\langle \mathtt{0}, \mathtt{0}, \mathtt{m}, \mathtt{0}\rangle$ rappresenta la situazione in cui non ci sono più cifre da sommare, non c'è riporto ed il risultato intermedio è $\mathtt{m}$. Le regole (vi) $\div$ (ix) corrispondono ai passi intermedi del calcolo in cui il risultato intermedio è $\mathtt{m}$ ed i numeri ancora da sommare sono costituiti: entrambi da una sola cifra, il primo da più cifre e il secondo da una sola cifra, il primo da una sola cifra ed il secondo da più cifre ed entrambi da più cifre, rispettivamente. La condizione a margine della regola (vi) inibisce l'uso della transizione stessa a partire dalla configurazione $\langle \mathtt{0}, \mathtt{0}, \mathtt{m}, \mathtt{0}\rangle$, per la quale è applicabile la regola (v).

Esempio 2.6 Vediamo un esempio di derivazione nel sistema che abbiamo appena definito, che corrisponde al "calcolo" della somma tra 927 e 346. Ad ogni passo della derivazione associamo (tra parentesi graffe) la giustificazione del passo stesso, costituita dall'etichetta che identifica la regola utilizzata e dall'istanza della sua eventuale premessa:

$$\begin{array}{ll} & \langle 927, 346\rangle \\ \rightarrow_{n+} & \quad \{\text{(iv)}, \langle 7, 6, 0\rangle \rightarrow_{cr} \langle 3, 1\rangle\ \} \\ & \langle 92, 34, 3, 1\rangle \end{array}$$

$\rightarrow_{n+}$ $\{(\text{ix}),\ \langle \mathtt{2,4,1}\rangle \rightarrow_{cr} \langle \mathtt{7,0}\rangle\ \}$

$\langle \mathtt{9,3,73,0}\rangle$

$\rightarrow_{n+}$ $\{(\text{vi}),\ \langle \mathtt{9,3,0}\rangle \rightarrow_{cr} \langle \mathtt{2,1}\rangle\ \}$

$\langle \mathtt{0,0,273,1}\rangle$

$\rightarrow_{n+}$ $\{(\text{vi}),\ \langle \mathtt{0,0,1}\rangle \rightarrow_{cr} \langle \mathtt{1,0}\rangle\ \}$

$\langle \mathtt{0,0,1273,0}\rangle$

$\rightarrow_{n+}$ $\{(\text{v})\}$

$\mathtt{1273}$

■

2.2 Rappresentazione e semantica dei numeri

Nel paragrafo precedente abbiamo usato la comune rappresentazione dei numeri naturali mediante sequenze di cifre decimali. Per capire in che senso il sistema $\mathtt{S}_{n+}$ *definisce* la somma tra numeri naturali, dobbiamo introdurre la distinzione tra un numero naturale e la sua rappresentazione. Sia allora $\mathbf{N} = \{\underline{0}, \underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}, \ldots\}$ l'insieme dei numeri naturali, sul quale sono definite le usuali operazioni di somma ($+$), prodotto ($\times$), quoziente e resto della divisione intera (*div* e *mod*): assumeremo anche di avere le operazioni di "uguaglianza" ($=$), "disuguaglianza" ($\neq$), "minore"($\prec$), "maggiore" ($\succ$), "maggiore o uguale" ($\succeq$), "minore o uguale" ($\preceq$) su coppie di valori in $\mathbf{N}$[1]. Ad ogni elemento `n` di `Num` corrisponde il *valore* in $\mathbf{N}$ di cui `n` è la *rappresentazione*. Tale associazione è data dalla seguente *funzione di valutazione* $\eta : \mathtt{Num} \rightarrow \mathbf{N}$, definita per casi nel modo seguente:

$$
\begin{array}{rcl}
\eta(\mathtt{0}) & = & \underline{0} \\
\eta(\mathtt{1}) & = & \underline{1} \\
 & \ldots & \\
\eta(\mathtt{9}) & = & \underline{9} \\
\eta(\mathtt{nc}) & = & (\eta(\mathtt{n}) \ \times \ \underline{10}) \ + \ \eta(\mathtt{c})
\end{array}
$$

Si noti come i casi nella definizione di η corrispondano proprio ai casi della definizione sintattica di `Num`. Analogamente, è possibile definire la *funzione di rappresentazione* $\nu : \mathbf{N} \rightarrow \mathtt{Num}$, definita come segue:

$$
\begin{array}{rcll}
\nu(\underline{0}) & = & \mathtt{0} & \\
\nu(\underline{1}) & = & \mathtt{1} & \\
 & \ldots & & \\
\nu(\underline{9}) & = & \mathtt{9} & \\
\nu(\underline{\mathtt{n}}) & = & \nu(\underline{\mathtt{n}}\ div\ \underline{10})\nu(\underline{\mathtt{n}}\ mod\ \underline{10}) & \text{se } \underline{\mathtt{n}} \ \succ \ \underline{9}
\end{array}
$$

La scelta di denotare i *valori* in $\mathbf{N}$ con la notazione $\underline{\mathtt{n}}$ non deve indurre a pensare che la funzione di valutazione (risp. rappresentazione) possa essere definita semplicemente come $\eta(\mathtt{n}) = \underline{\mathtt{n}}$ (risp. $\nu(\underline{\mathtt{n}}) = \mathtt{n}$). La notazione $\underline{\mathtt{n}}$ è anch'essa una *rappresentazione*, diversa da quella sintattica, da noi scelta per denotare un *valore* in $\mathbf{N}$. La funzione di valutazione definita per casi fornisce un modo per calcolare in modo *costruttivo* il valore corrispondente ad una stringa della particolare sintassi (linguaggio di rappresentazione) utilizzata (nel nostro caso, di una stringa di `Num`).

[1]Si noti la necessità di distinguere tra i simboli utilizzati per denotare le operazioni su $\mathbf{N}$ ed i simboli utilizzati per rappresentare sintatticamente gli stessi. Così, ad esempio, `+` è il simbolo *sintattico* utilizzato per rappresentare l'operazione di somma $+$ tra numeri naturali, ovvero tra elementi di $\mathbf{N}$. Analogamente, $\times, -, div, mod, =, \neq, \prec, \succ, \succeq, \preceq$ sono operazioni su coppie di naturali e non vanno confuse con i corrispondenti simboli sintattici `*`, `-`, `/`, `%`, `==`, `!=`, <, >, >= e <=.

Per convincerci ulteriormente della necessità di distinguere tra *valori* e *rappresentazioni*, utilizziamo una sintassi diversa per la descrizione di numeri naturali, che corrisponde alla rappresentazione *binaria* dei numeri. Utilizziamo volutamente i simboli `z` e `u`, anziché gli usuali simboli `0` e `1`, per le cifre binarie (*bit*) proprio per evidenziarne il carattere puramente *simbolico*.

```
Binary ::= Bit | Binary Bit
Bit ::= z | u
```

Qual è il numero naturale rappresentato dalla stringa `zuzz`? Definiamo per casi una nuova funzione di valutazione $\eta' : \texttt{Binary} \to \mathbf{N}$ per il nuovo linguaggio. Nella definizione di η' utilizziamo la convenzione di denotare con x una generica stringa in `Binary` e con b un generico bit (ovvero b sta per `z` oppure `u`).

$$\begin{array}{rcl}\eta'(\texttt{z}) & = & \underline{0}\\ \eta'(\texttt{u}) & = & \underline{1}\\ \eta'(xb) & = & (\eta'(x)\ \times\ \underline{2})\ +\ \eta'(b)\end{array}$$

Applicando la funzione η' alla stringa `zuzz` otteniamo:

$$\begin{array}{rcl}\eta'(\texttt{zuzz}) & = & (\eta'(\texttt{zuz}) \times \underline{2}) + \eta'(\texttt{z})\\ & = & (\eta'(\texttt{zuz}) \times \underline{2}) + \underline{0}\\ & = & (((\eta'(\texttt{zu}) \times \underline{2}) + \eta'(\texttt{z})) \times \underline{2}) + \underline{0}\\ & = & (((\eta'(\texttt{zu}) \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0}\\ & = & (((((\eta'(\texttt{z}) \times \underline{2}) + \eta'(\texttt{u})) \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0}\\ & = & (((((\underline{0} \times \underline{2}) + \underline{1}) \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0}\\ & = & ((((\underline{0} + \underline{1}) \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0}\\ & = & (((\underline{1} \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0}\\ & = & ((\underline{2} + \underline{0}) \times \underline{2}) + \underline{0}\\ & = & (\underline{2} \times \underline{2}) + \underline{0}\\ & = & \underline{4} + \underline{0}\\ & = & \underline{4}\end{array}$$

Il sistema di transizioni $\texttt{S}_{n+}$ definisce la somma tra numeri naturali nel senso che, data la rappresentazione di due valori naturali, fornisce la rappresentazione della loro somma. Ciò può essere espresso formalmente dal seguente asserto:

$$\textbf{se } \langle \texttt{n}, \texttt{m} \rangle \to^{*}_{n+} \texttt{k} \qquad \textbf{allora} \qquad \eta(\texttt{n})\ +\ \eta(\texttt{m}) = \eta(\texttt{k})$$

È facile convincersi che vale anche il viceversa, ovvero:

$$\textbf{se } \underline{\texttt{n}}\ +\ \underline{\texttt{m}} = \underline{\texttt{k}} \qquad \textbf{allora} \qquad \langle \nu(\underline{\texttt{n}}), \nu(\underline{\texttt{m}}) \rangle \to^{*}_{n+} \nu(\underline{\texttt{k}})$$

Sebbene la distinzione tra valori/operazioni e loro rappresentazione sintattica sia formalmente necessaria, nel seguito la ometteremo quando ciò non crei confusione o ambiguità. Utilizzeremo ad esempio `n` sia per indicare il valore $\underline{\texttt{n}}$ sia per indicare la rappresentazione di $\underline{\texttt{n}}$, così come scriveremo $\underline{\texttt{n}} > \underline{\texttt{m}}$ o `n>m` anziché $\underline{\texttt{n}} \succ \underline{\texttt{m}}$.

3 La definizione operazionale della semantica

In questo paragrafo vediamo come i sistemi di transizioni possano essere utilizzati per dare la semantica in stile operazionale di un semplice linguaggio: quello delle espressioni aritmetiche. L'approccio operazionale alla semantica dei linguaggi di programmazione prevede, come accennato in precedenza, la definizione del comportamento di un sistema (di transizioni) in corrispondenza dei costrutti del linguaggio. Nel caso del linguaggio `Exp` che stiamo considerando, le configurazioni rappresentano gli stati in cui il sistema si trova ad operare durante il calcolo di una espressione mentre le transizioni rappresentano il procedere del calcolo stesso. Le configurazioni terminali corrispondono poi agli stati finali, che rappresentano in qualche modo il *risultato* del calcolo. A seconda del livello di dettaglio che si sceglie per descrivere l'evolvere del calcolo si ottiene un sistema più o meno *astratto*.

Tradizionalmente, nella descrizione dei linguaggi di programmazione si hanno due tipi di sistema, che si differenziano proprio per il livello di dettaglio scelto nella descrizione dei singoli passi di calcolo. La semantica che si ottiene nei due casi viene detta, rispettivamente, *semantica di valutazione* (*small-step*) e *semantica naturale* (*big-step*). È importante notare che, al di là della loro definizione, le due semantiche sono equivalenti, nel senso che conducono a risultati equivalenti a meno di semplici trasformazioni.

Nella semantica naturale, l'idea è di descrivere direttamente in che modo si ottiene il risultato di un calcolo, astraendo quanto più possibile dai passi intermedi. Più precisamente, ciò che la relazione di transizione descrive è direttamente la corrispondenza tra le configurazioni iniziali e le configurazioni finali.

Nella semantica di valutazione lo scopo è di descrivere i singoli passi del calcolo. Ad esempio, nel sistema del paragrafo precedente, le configurazioni non terminali rappresentano gli stadi intermedi della valutazione di una addizione e quelle terminali il risultato. Il valore di una somma (o meglio la rappresentazione di tale valore) si ottiene attraverso una derivazione che, a partire da una configurazione iniziale corrispondente all'addizione stessa e passando attraverso una serie di stadi intermedi, determina la configurazione terminale che rappresenta il valore del risultato. In queste note ci concentreremo sulla semantica naturale dei linguaggi.

3.1 Espressioni semplificate

Consideriamo dapprima il linguaggio delle espressioni semplificate:

```
Exp ::= Num | Exp + Exp
```

Una descrizione informale, ma sicuramente vicina alla nostra intuizione, di come si calcola il valore di un'espressione in tale linguaggio, può essere data come segue:

(r1) il valore di un'espressione costituita da un numero `n` è il valore rappresentato da `n`, ovvero $\underline{\mathtt{n}}$;

(r2) il valore di un'espressione del tipo `E+E'` si ottiene sommando i valori $\underline{\mathtt{n}}$ e $\underline{\mathtt{m}}$, dove $\underline{\mathtt{n}}$ è il valore di `E` e $\underline{\mathtt{m}}$ è il valore di `E'`.

Osserviamo innanzitutto che le due regole corrispondono alle due alternative nella definizione sintattica di `Exp`. La seconda regola suggerisce anche un modo per *calcolare* il valore di un'espressione complessa `E+E'`, che consiste nel calcolare il valore delle due sottoespressioni `E` ed `E'` per poi calcolarne la somma (assumendo, naturalmente, di saper calcolare la somma tra due numeri naturali).

Vediamo come queste due regole consentano di determinare, ad esempio, il valore dell'espressione `2+3`. Data la struttura sintattica di `2+3`, non possiamo che appellarci inizialmente alla regola (r2)

(a) il valore di `2+3` è $\underline{\mathtt{n}} + \underline{\mathtt{m}}$, dove $\underline{\mathtt{n}}$ è il valore di `2` e $\underline{\mathtt{m}}$ è il valore di `3`.

Applicando tale regola abbiamo dunque ridotto il problema alla determinazione dei due valori $\underline{\mathtt{n}}$ e $\underline{\mathtt{m}}$ e a questo scopo possiamo utilizzare la regola (r1), guidati ancora dalla struttura sintattica delle espressioni in gioco.

(b1) il valore di `2` è $\underline{\mathtt{2}}$

(b2) il valore di `3` è $\underline{\mathtt{3}}$.

Sappiamo inoltre che

(b3) la somma di $\underline{\mathtt{2}}$ e $\underline{\mathtt{3}}$ è $\underline{\mathtt{5}}$

e mettendo insieme (a), (b1), (b2), (b3), possiamo concludere che il valore di 2+3 è $\underline{5}$.

Come possiamo riflettere l'intuizione espressa dalle regole precedenti nella descrizione di un sistema di transizioni? Immaginiamo, in prima approssimazione, di utilizzare un sistema le cui configurazioni sono semplicemente espressioni e di voler definire una relazione di transizione i cui elementi siano transizioni del tipo $\mathtt{E} \rightarrow \mathtt{n}$, dove E è un'espressione e n è la rappresentazione del suo valore. Una possibilità consiste nell'elencare un numero (infinito!) di transizioni del tipo

$$\mathtt{0+1} \rightarrow \mathtt{1} \qquad \mathtt{1+1} \rightarrow \mathtt{2} \qquad \mathtt{1+2+2} \rightarrow \mathtt{5} \qquad \ldots$$

Un sistema descritto in questo modo è troppo *astratto*, nel senso che non suggerisce alcun modo per "calcolare" un'espressione e dunque ottenerne il valore.

Riconsideriamo dapprima la regola (r2): nel sistema che stiamo cercando di definire, una transizione $\mathtt{E} \rightarrow \mathtt{n}$ esprime il fatto che il valore di E è il numero naturale rappresentato da n. Dunque, (r2) può essere vista come regola per determinare la transizione dalla configurazione del tipo $\mathtt{E+E}'$ alla configurazione che rappresenta il valore di $\mathtt{E+E}'$. Detto altrimenti, la regola (r2) può essere riformulata in termini del sistema di transizioni che stiamo definendo, nel seguente modo:

$$\textbf{se} \quad \mathtt{E} \rightarrow \mathtt{n} \quad \textbf{e} \quad \mathtt{E}' \rightarrow \mathtt{m} \qquad \textbf{allora} \qquad \mathtt{E} + \mathtt{E}' \rightarrow \mathtt{k}, \text{ dove } \mathtt{k=n+m} \qquad (\dagger)$$

Vediamo ora come si traduce in termini di transizioni la regola (r1). Il valore dell'espressione n è rappresentato da n stesso e dunque per le configurazioni del tipo n non abbiamo bisogno di introdurre alcuna transizione: esse sono proprio le configurazioni terminali. Questa scelta, apparentemente corretta, è in contrasto proprio con la regola (†) appena introdotta: come calcoliamo, ad esempio, il valore di 2+3? La regola (†) ci impone di determinare due transizioni del tipo $\mathtt{2} \rightarrow \mathtt{n}$ e $\mathtt{3} \rightarrow \mathtt{m}$, ma abbiamo appena stabilito che non vi sono transizioni la cui parte sinistra è la rappresentazione di un numero!

Un modo per ovviare a questo inconveniente ci è di nuovo suggerito dalla descrizione informale delle regole (r1) e (r2): in esse, infatti, la distinzione tra rappresentazioni e valori è esplicita e ciò consente di parlare del valore $\underline{\mathtt{n}}$ di un'espressione semplice n. Nel sistema di transizioni che stiamo definendo, ciò si riflette nel trattare esplicitamente i valori: a questo scopo, basta considerare un nuovo insieme di configurazioni che, oltre alle espressioni, contiene anche i valori, ovvero gli elementi di **N**.

Riassumendo, abbiamo le seguenti configurazioni:

$$\begin{array}{rcl} \Gamma & = & \{\mathtt{E} \mid \mathtt{E} \in \mathtt{Exp}\} \cup \\ & & \{\underline{\mathtt{n}} \mid \underline{\mathtt{n}} \in \mathbf{N}\} \\ \mathtt{T} & = & \{\underline{\mathtt{n}} \mid \underline{\mathtt{n}} \in \mathbf{N}\} \end{array}$$

Siamo in grado di descrivere le regole del sistema di transizioni per le espressioni semplificate. In esse, utilizziamo la notazione introdotta in precedenza per le regole condizionali, nonché la convenzione che n rappresenta un elemento della categoria sintattica Num, mentre E ed E′ rappresentano elementi della categoria sintattica Exp.

$$\mathtt{n} \rightarrow \underline{\mathtt{n}} \qquad (r1)$$

$$\frac{\mathtt{E} \rightarrow \underline{\mathtt{n}} \qquad \mathtt{E}' \rightarrow \underline{\mathtt{m}} \qquad \underline{\mathtt{k}} = \underline{\mathtt{n}} + \underline{\mathtt{m}}}{\mathtt{E+E}' \rightarrow \underline{\mathtt{k}}} \qquad (r2)$$

Si noti come la definizione delle regole sia *guidata* dalla sintassi del linguaggio: la prima regola definisce le transizioni per configurazioni costituite da un elemento della categoria sintattica Num, mentre la seconda definisce transizioni per espressioni in cui compare l'operatore +. La premessa $\underline{\mathtt{k}} = \underline{\mathtt{n}} + \underline{\mathtt{m}}$ della seconda regola corrisponde all'ipotesi di saper calcolare la somma tra numeri naturali: questo calcolo può essere visto come una derivazione in un sottosistema simile a $\mathtt{S}_{n+}$ del Paragrafo 2.1.2. Le altre due premesse della regola (r2) sono infine transizioni nel sistema che stiamo definendo: esse corrispondono al calcolo delle sottoespressioni dell'espressione di partenza.

Esempio 3.1 Vediamo un esempio di derivazione in questo sistema che corrisponde a valutare l'espressione 2+3+8, ovvero a determinare una transizione del tipo 2+3+8 $\rightarrow \underline{\mathtt{m}}$ per qualche $\underline{\mathtt{m}}$, applicando opportunamente le regole (r1) e (r2).

$$
\begin{array}{ll}
 & \mathtt{2+3+8} \\
\rightarrow & \quad \{\ (\mathrm{r2}), \\
 & \quad\ \ (\mathrm{r1}):\ \mathtt{2} \rightarrow \underline{\mathtt{2}} \\
 & \quad\ \ (\mathrm{d1}):\ \mathtt{3+8} \rightarrow \underline{\mathtt{11}}, \quad \underline{\mathtt{2}} + \underline{\mathtt{11}} = \underline{\mathtt{13}}\} \\
 & \underline{\mathtt{13}}
\end{array}
$$

La derivazione corrisponde ad una applicazione della regola (r2). Le premesse della regola, opportunamente istanziate, sono riportate come parti della giustificazione: la prima, un'istanza della regola (r1), e la terza, il calcolo della somma di due numeri, sono sufficientemente semplici da non richiedere ulteriori giustificazioni. La seconda, invece, è la conclusione che si ottiene dalla sottoderivazione (d1) seguente:

(d1):

$$
\begin{array}{ll}
 & \mathtt{3+8} \\
\rightarrow & \quad \{\ (\mathrm{r2}), \\
 & \quad\ \ (\mathrm{r1}):\ \mathtt{3} \rightarrow \underline{\mathtt{3}} \\
 & \quad\ \ (\mathrm{r1}):\ \mathtt{8} \rightarrow \underline{\mathtt{8}}, \quad \underline{\mathtt{3}} + \underline{\mathtt{8}} = \underline{\mathtt{11}}\} \\
 & \underline{\mathtt{11}}
\end{array}
$$

■

In generale, una derivazione è dunque costituita da una derivazione *principale* e da tante *sottoderivazioni* quante sono le transizioni non banali del tipo $\gamma \rightarrow \gamma'$ che compaiono nelle sue giustificazioni. Anche le sottoderivazioni possono contenere transizioni di questo tipo nelle proprie giustificazioni, che andranno eventualmente giustificate mediante ulteriori sottoderivazioni.

Un modo compatto per rappresentare l'intera derivazione è di costruire un albero in cui un nodo e i suoi figli rappresentano, rispettivamente, la conclusione dell'applicazione di una regola e le istanze delle premesse di tale regola. La derivazione precedente, in questa rappresentazione grafica, risulta essere la seguente.

$$
\dfrac{\dfrac{}{\mathtt{2} \rightarrow \underline{\mathtt{2}}}\,(r1) \quad \dfrac{\dfrac{}{\mathtt{3} \rightarrow \underline{\mathtt{3}}}\,(r1) \quad \dfrac{}{\mathtt{8} \rightarrow \underline{\mathtt{8}}}\,(r1) \quad \underline{\mathtt{11}} = \underline{\mathtt{3}} + \underline{\mathtt{8}}}{\mathtt{3+8} \rightarrow \underline{\mathtt{11}}}\,(r2) \quad \underline{\mathtt{13}} = \underline{\mathtt{2}} + \underline{\mathtt{11}}}{\mathtt{2+3+8} \rightarrow \underline{\mathtt{13}}}\,(r2)
$$

L'albero è stato ottenuto partendo dalla espressione 2+3+8 da valutare ed applicando la regola (r2) ottenendo

$$
\dfrac{\mathtt{2} \rightarrow \underline{\mathtt{n}} \quad \mathtt{3+8} \rightarrow \underline{\mathtt{m}} \quad \underline{\mathtt{k}} = \underline{\mathtt{n}} + \underline{\mathtt{m}}}{\mathtt{2+3+8} \rightarrow \underline{\mathtt{k}}}\,(r2)
$$

Le ulteriori applicazioni delle regole alle premesse hanno poi consentito di espandere l'albero verso l'alto e di determinare i valori $\underline{\mathtt{n}}$, $\underline{\mathtt{m}}$ e $\underline{\mathtt{k}}$, fino ad ottenere l'albero finale rappresentato nella figura precedente. Un ulteriore passo consente ad esempio di determinare il valore $\underline{\mathtt{n}}$ come segue:

$$
\dfrac{\dfrac{}{\mathtt{2} \rightarrow \underline{\mathtt{2}}}\,(r1) \quad \mathtt{3+8} \rightarrow \underline{\mathtt{m}} \quad \underline{\mathtt{k}} = \underline{\mathtt{2}} + \underline{\mathtt{m}}}{\mathtt{2+3+8} \rightarrow \underline{\mathtt{k}}}\,(r2)
$$

Si noti che l'applicazione di regole senza premesse determina un nodo foglia (cioè senza figli) dell'albero.

In queste note, tuttavia, continueremo ad utilizzare prevalentemente la rappresentazione che rimanda il calcolo delle premesse a sottoderivazioni presentate successivamente.

Un'osservazione importante sull'esempio precedente riguarda la scelta delle sottoespressioni ridotte nelle giustificazioni. Per quale motivo abbiamo scelto di ridurre, nel primo passo della derivazione, la sottoespressione `3+8` e non, ad esempio, la sottoespressione `2+3`? Ricordiamo, a questo proposito, che la stringa `2+3+8` è per noi una rappresentazione lineare del suo albero di derivazione in una grammatica, non ambigua, che definisce lo stesso linguaggio delle espressioni associato alla grammatica, ambigua, data in precedenza. Tutte le scelte fatte nella derivazione precedente corrispondono all'assunzione che l'albero di derivazione della stringa `2+3+8` sia quello rappresentato (ancora in modo lineare, ma questa volta mettendo in evidenza la struttura dell'espressione) da `2+[3+8]`[2].

Esempio 3.2 Vediamo come esempio la derivazione che si ottiene assumendo che l'albero sintattico sia ora `[2+3]+8`.

`2+3+8`

$\to$ { (r2),
(d1): $\mathtt{2} + \mathtt{3} \to \underline{5}$
(r1): $\mathtt{8} \to \underline{8}$
$\underline{5} + \underline{8} = \underline{13}$ }

$\underline{13}$

con la sottoderivazione (d1) ottenuta come segue:

(d1):

`2+3`

$\to$ { (r2),
(r1): $\mathtt{2} \to \underline{2}$
(r1): $\mathtt{3} \to \underline{3}$
$\underline{2} + \underline{3} = \underline{5}$ }

$\underline{5}$

Le proprietà dell'operatore di somma tra numeri naturali garantiscono che il risultato, nei due casi, è il medesimo, ma ciò chiaramente non vale per un linguaggio più espressivo come quello che stiamo per introdurre. ■

3.2 Le espressioni a valori naturali

La grammatica del linguaggio che definisce le espressioni a valori naturali è data dalle seguenti produzioni:

```
Exp ::= Num | (Exp) | Exp Op Exp.
Op  ::= + | - | * | / | %
Num ::= ...
```

Osserviamo che questa grammatica non solo non assegna nessuna precedenza agli operatori, ma è anche ambigua: ciò ha solamente lo scopo di semplificare la definizione semantica. Infatti, la semantica operazionale (o meglio le regole che definiscono la relazione di transizione) è definita per casi in accordo alla sintassi del linguaggio. Poiché il processo di disambiguazione e di assegnazione di precedenze tra gli operatori richiede l'introduzione di ulteriori simboli non terminali nella grammatica, questo significherebbe complicare anche la definizione delle regole a scapito della chiarezza. La possibilità di scrivere espressioni non ambigue è data dalla presenza della produzione `(Exp)`. È da notare inoltre che, concettualmente,

[2]Le parentesi `[` e `]` non devono essere intese come simboli sintattici, ma solo come ausilio per rappresentare linearmente l'albero di derivazione.

la definizione semantica non risulterebbe diversa: per queste ragioni abbiamo scelto di mantenere la definizione più compatta possibile a scapito della completezza.

Denotiamo con $\rightarrow_{exp}$ la relazione di transizione del sistema $\mathtt{S}_{exp}$ che definisce la semantica del linguaggio per le espressioni a valori naturali, le cui configurazioni sono, come nel caso delle espressioni del paragrafo precedente:

$$
\begin{array}{rcl}
\Gamma_{exp} & = & \{\mathtt{E} \mid \mathtt{E} \in \mathtt{Exp}\} \cup \\
 & & \{\underline{\mathtt{n}} \mid \underline{\mathtt{n}} \in \mathbf{N}\} \\
\mathtt{T}_{exp} & = & \{\underline{\mathtt{n}} \mid \underline{\mathtt{n}} \in \mathbf{N}\}
\end{array}
$$

Le regole che definiscono $\rightarrow_{exp}$ sono elencate qui di seguito. Alcune delle premesse utilizzate nelle regole corrispondono ad operazioni "elementari" tra numeri che si suppone di saper calcolare mediante un sottosistema opportuno (come nel caso della premessa $\underline{\mathtt{n}}+\underline{\mathtt{m}} = \underline{\mathtt{k}}$ della regola (r2) del paragrafo precedente). Per semplicità, tali premesse verranno rappresentate mediante uguaglianze del tipo $\underline{\mathtt{n}} + \underline{\mathtt{m}} = \underline{\mathtt{k}}$, $\underline{\mathtt{n}} \times \underline{\mathtt{m}} = \underline{\mathtt{k}}$, ecc.

$$\mathtt{n} \rightarrow_{exp} \underline{\mathtt{n}} \qquad (exp_n)$$

$$\frac{\mathtt{E} \rightarrow_{exp} \underline{\mathtt{n}} \qquad \mathtt{E}' \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} + \underline{\mathtt{n}}'}{\mathtt{E+E}' \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_+)$$

$$\frac{\mathtt{E} \rightarrow_{exp} \underline{\mathtt{n}} \qquad \mathtt{E}' \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} \times \underline{\mathtt{n}}'}{\mathtt{E*E}' \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_*)$$

$$\frac{\mathtt{E} \rightarrow_{exp} \underline{\mathtt{n}} \qquad \mathtt{E}' \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}} \geq \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} - \underline{\mathtt{n}}'}{\mathtt{E\text{-}E}' \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_-)$$

$$\frac{\mathtt{E} \rightarrow_{exp} \underline{\mathtt{n}} \qquad \mathtt{E}' \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}}' \neq \underline{\mathtt{0}} \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}}\ div\ \underline{\mathtt{n}}'}{\mathtt{E}\ /\ \mathtt{E}' \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_{div})$$

$$\frac{\mathtt{E} \rightarrow_{exp} \underline{\mathtt{n}} \qquad \mathtt{E}' \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}}' \neq \underline{\mathtt{0}} \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}}\ mod\ \underline{\mathtt{n}}'}{\mathtt{E}\ \%\ \mathtt{E}' \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_{mod})$$

$$\frac{\mathtt{E} \rightarrow_{exp} \underline{\mathtt{n}}}{\mathtt{(E)} \rightarrow_{exp} \underline{\mathtt{n}}} \qquad (exp_{()})$$

Queste regole sono semplici estensioni delle regole date nel paragrafo precedente. Si osservi che le premesse delle regole (exp_{div}) e (exp_{mod}), che corrispondono al calcolo di quoziente e resto della divisione intera, contengono, tra le premesse, la condizione $\underline{\mathtt{n}}' \neq \underline{\mathtt{0}}$, caso in cui l'operazione di divisione non è definita. Analogamente, la premessa $\underline{\mathtt{n}} \geq \underline{\mathtt{n}}'$ della regola (exp_-) è necessaria per garantire che il risultato della differenza sia ancora un valore naturale.

Un'ultima osservazione riguarda il significato della regola ($exp_{()}$): se letta distrattamente, essa può apparire scorretta, dal momento che la transizione consiste esclusivamente nella "semplificazione" di `(E)` in `E`. Tale semplificazione, se vista nel contesto di una espressione complessa, non sarebbe lecita: si pensi ad esempio alla semplificazione da `(2+3)*5` in `2+3*5` e alla usuale precedenza tra gli operatori `+` e `*` . Ricordiamo ancora che, in realtà, `E` sta ad indicare l'*albero sintattico* dell'espressione `E`: dunque la semplificazione precedente sta ad indicare che, nel contesto in cui si trova, l'albero sintattico di `(E)` viene rimpiazzato dall'albero sintattico di `E`, e dunque la regola è corretta. Utilizzando ancora le parentesi `[` e `]` per mettere in evidenza la struttura dell'albero di derivazione, la regola $(exp_{()})$ consente di ottenere la transizione $[(2+3)] * 5 \rightarrow_{exp} [2+3] * 5$.

In che senso questo sistema definisce la semantica del linguaggio `Exp`? L'idea è quella di vedere le configurazioni finali come i *valori* delle espressioni di partenza. Denotando con $\mathcal{E}[\![\mathtt{E}]\!]$ la semantica di un'espressione, abbiamo formalmente:

$$\boxed{\mathcal{E}[\![\mathtt{E}]\!] = \underline{\mathtt{k}} \quad \textbf{se e solo se} \quad \mathtt{E} \rightarrow_{exp} \underline{\mathtt{k}}.}$$

Si noti che nel sistema $\mathtt{S}_{exp}$ esistono configurazioni *bloccate*, quali ad esempio `2-15`, `5 / (3-3)`, e così via. Intuitivamente, queste configurazioni corrispondono ad espressioni che, seppure sintatticamente corrette, non hanno un valore naturale.

Esempio 3.3 Vediamo quale è la derivazione in $\mathtt{S}_{exp}$ che consente di stabilire che il valore dell'espressione `25 - 15 / 2` è $\underline{18}$.

$$\begin{array}{ll}
 & \mathtt{25\ -\ 15\ /\ 2} \\
\rightarrow_{exp} & \{\ (exp_{-}), \\
 & \quad (exp_n)\text{: } 25 \rightarrow_{exp} \underline{25}, \\
 & \quad (\text{d1})\text{: } \mathtt{15\ /\ 2} \rightarrow_{exp} \underline{7}, \\
 & \quad \underline{25} \geq \underline{7}, \quad \underline{25} - \underline{7} = \underline{18}\ \} \\
 & \underline{18}
\end{array}$$

dove (d1) è data da:

(d1):

$$\begin{array}{ll}
 & \mathtt{15\ /\ 2} \\
\rightarrow_{exp} & \{\ (exp_{div}), \\
 & \quad (exp_n)\text{: } 15 \rightarrow_{exp} \underline{15}, \\
 & \quad (exp_n)\text{: } 2 \rightarrow_{exp} \underline{2}, \\
 & \quad \underline{2} \neq \underline{0}, \quad \underline{15}\ div\ \underline{2} = \underline{7}\ \} \\
 & \underline{7}
\end{array}$$

Osserviamo ancora che le sottoderivazioni "semplici" (che corrispondono ad istanze della regola (exp_n)) sono state direttamente riportate nelle giustificazioni delle regole in cui vengono utilizzate. La sottoderivazione che corrisponde alla valutazione di `15 / 2` è stata invece indicata con (d1) e riportata esplicitamente. Osserviamo infine che anche in questo caso è stata fatta una precisa assunzione sull'albero sintattico dell'espressione in gioco (rappresentato da `25 - [15 /2]`). ■

Esercizi

1. Valutare le seguenti espressioni, indicando per ciascuna l'albero sintattico sottinteso nella derivazione:
   (a) `15+(20 / 6) - 15`
   (b) `100 % 7 + 11`
   (c) `35 - (10 % 4) + 7`
2. Valutare l'espressione `12 / 2 - 2` supponendo che il suo albero sintattico sia:
   (a) `[12 / 2] - 2`
   (b) `12 / [2 - 2]`

4 Lo stato

Immaginiamo di estendere il linguaggio delle espressioni del Paragrafo 3.2 aggiungendo la nuova produzione `Exp ::= Ide`, che consente di scrivere espressioni in cui compaiono dei nomi (identificatori)[3]. È evidente che il significato di un'espressione come `x+2` dipende dal valore associato al nome `x`: in generale, il valore di un'espressione in cui compaiono i nomi $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ dipende dal valore associato a tali nomi[4]. Per descrivere la semantica di espressioni siffatte, aggiungiamo un ulteriore ingrediente al nostro sistema di transizioni, il cosiddetto *stato*. In questa sezione introduciamo dapprima lo stato nella sua forma più semplice, ovvero come tabella (un insieme di associazioni tra nomi e valori), e successivamente in una forma più strutturata, ovvero come sequenza di tabelle, necessaria per trattare alcuni meccanismi di Java, quali l'annidamento di blocchi.

4.1 Il concetto di stato

Nella sua forma più semplice, uno stato non è altro che un insieme di associazioni del tipo $\langle nome, val \rangle$ dove *val* è un valore. Indichiamo genericamente con `Val` l'insieme dei valori associati ai nomi nello stato: dunque, per quanto visto sino ad ora `Val` è l'insieme $\mathbf{N}$. Uno stato φ può essere visto come una funzione

$$\varphi : \mathtt{Ide} \mapsto \mathtt{Val}$$

che, dato un nome `x` restituisce il valore associato ad `x` in φ. In generale, uno stato φ è una funzione *parziale* sul dominio `Ide`, ovvero una funzione definita solo su alcuni elementi di `Ide` e non su altri. Addirittura, ci troveremo sempre a trattare con stati che contengono associazioni per un numero molto limitato di identificatori (tipicamente quelli che compaiono in un programma) e dunque per un sottoinsieme molto ristretto dell'insieme (infinito!) `Ide`.

Nel seguito, utilizzeremo $\varphi, \varphi_0, \varphi', \varphi'', \ldots$ per denotare generici stati. Inoltre, adotteremo l'usuale notazione $\varphi(\mathtt{x})$ per denotare il valore *associato ad* `x` *nello stato* φ. Dato uno stato φ che non contiene alcuna associazione per l'identificatore `x`, diremo che $\varphi(\mathtt{x}) = \perp$, dove $\perp$ indica appunto che la funzione φ è indefinita sull'argomento `x`.

A volte avremo bisogno di indicare esplicitamente *tutti e soli* gli identificatori per i quali è prevista una associazione nello stato φ. Siano $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_k$ tali identificatori e siano $\underline{\mathtt{v}}_1, \underline{\mathtt{v}}_2, \ldots, \underline{\mathtt{v}}_k$ i valori ad essi associati nello stato φ; un modo per rappresentare tale stato consiste nel definire φ come la funzione:

$$\varphi(x) = \begin{cases} \underline{\mathtt{v}}_1 & \text{se } x = \mathtt{x}_1 \\ \underline{\mathtt{v}}_2 & \text{se } x = \mathtt{x}_2 \\ \ldots & \\ \underline{\mathtt{v}}_k & \text{se } x = \mathtt{x}_k \\ \perp & \text{altrimenti} \end{cases}$$

Un modo alternativo, ma più conciso, consiste nell'elencare le associazioni presenti in φ come segue:

$$\varphi = \{\mathtt{x}_1 \mapsto \underline{\mathtt{v}}_1, \mathtt{x}_2 \mapsto \underline{\mathtt{v}}_2, \ldots, \mathtt{x}_k \mapsto \underline{\mathtt{v}}_k\}$$

In particolare, indicheremo con $\varphi = \omega$ uno stato *vuoto*, ovvero tale che $\varphi(\mathtt{x}) = \perp$ qualunque sia `x`. Si noti che, in questa notazione, è essenziale che gli identificatori $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_k$ siano tutti distinti tra loro, mentre non è importante l'ordine in cui vengono elencate le associazioni $\mathtt{x}_i \mapsto \underline{\mathtt{v}}_i$ all'interno delle parentesi graffe.

4.1.1 Espressioni con stato

La dipendenza delle espressioni dallo stato (ovvero dai valori associati ai nomi che compaiono nell'espressione stessa), si riflette, nei sistemi di transizioni, nel fatto che le configurazioni non terminali del sistema stesso sono ora coppie del tipo $\langle \mathtt{E}, \varphi \rangle$. Nella nuova semantica, il significato di una derivazione $\langle \mathtt{E}, \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}}$ è che il valore di `E`, date le associazioni in φ, è $\underline{\mathtt{n}}$.

[3] `Ide` è la categoria sintattica a partire dalla quale è possibile generare identificatori.

[4] Esistono tuttavia espressioni il cui valore è indipendente dal valore associato ai nomi che vi compaiono, come ad esempio `x-x`.

$$\langle \mathtt{n}, \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}} \qquad (exp_n)$$

$$\frac{\varphi(\mathtt{x}) = \underline{\mathtt{n}}}{\langle \mathtt{x}, \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}}} \qquad (exp_{ide})$$

$$\frac{\langle \mathtt{E}, \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} + \underline{\mathtt{n}}'}{\langle \mathtt{E+E}', \varphi \rangle \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_+)$$

$$\frac{\langle \mathtt{E}, \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} \times \underline{\mathtt{n}}'}{\langle \mathtt{E*E}', \varphi \rangle \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_*)$$

$$\frac{\langle \mathtt{E}, \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}} \geq \underline{\mathtt{n}}' \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} - \underline{\mathtt{n}}'}{\langle \mathtt{E\text{-}E}', \varphi \rangle \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_-)$$

$$\frac{\langle \mathtt{E}, \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}}' \neq \underline{0} \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} \; div \; \underline{\mathtt{n}}'}{\langle \mathtt{E\ /\ E}', \varphi \rangle \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_{div})$$

$$\frac{\langle \mathtt{E}, \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}} \qquad \langle \mathtt{E}', \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}}' \qquad \underline{\mathtt{n}}' \neq \underline{0} \qquad \underline{\mathtt{m}} = \underline{\mathtt{n}} \; mod \; \underline{\mathtt{n}}'}{\langle \mathtt{E\ \%\ E}', \varphi \rangle \rightarrow_{exp} \underline{\mathtt{m}}} \qquad (exp_{mod})$$

$$\frac{\langle \mathtt{E}, \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}}}{\langle \mathtt{(E)}, \varphi \rangle \rightarrow_{exp} \underline{\mathtt{n}}} \qquad (exp_{()})$$

Abbiamo dovuto introdurre una nuova regola corrispondente alla nuova alternativa sintattica per le espressioni: (exp_{ide}) esprime il fatto che il valore dell'espressione costituita dal solo identificatore `x` nello stato φ non è altro che il valore associato ad `x` in tale stato, ovvero $\varphi(\mathtt{x})$.

La semantica di un'espressione è ora parametrica rispetto allo stato ed è quindi definita come:

$$\boxed{\mathcal{E}[\![\mathtt{E}]\!]\varphi = \underline{\mathtt{k}} \quad \textbf{se e solo se} \quad \langle \mathtt{E}, \varphi \rangle \rightarrow_{exp} \underline{\mathtt{k}}.}$$

Esempio 4.1 Vediamo come esempio la semantica dell'espressione `(x+8) / y` a partire da uno stato φ in cui al nome `x` è associato il valore $\underline{5}$ ed al nome `y` il valore $\underline{2}$, ovvero da uno stato φ tale che $\varphi(\mathtt{x}) = \underline{5}$ e $\varphi(\mathtt{y}) = \underline{2}$.

$\langle \mathtt{(x+8)\ /\ y}, \varphi \rangle$

$\rightarrow_{exp}$ { (exp_{div}),
(d1): $\langle \mathtt{(x+8)}, \varphi \rangle \rightarrow_{exp} \underline{13}$,
(exp_{ide}): $\langle \mathtt{y}, \varphi \rangle \rightarrow_{exp} \underline{2}$
$\underline{2} \neq \underline{0}$, $\underline{13} \; div \; \underline{2} = \underline{6}$ }

$\underline{6}$

(d1):

$\langle \mathtt{(x+8)}, \varphi \rangle$

$\rightarrow_{exp}$ { $(exp_{()})$,
(d2): $\langle \mathtt{x+8}, \varphi \rangle \rightarrow_{exp} \underline{13}$ }

$\underline{13}$

(d2):

$$
\begin{array}{ll}
 & \langle \mathtt{x+8}, \varphi \rangle \\
\rightarrow_{exp} & \{\ (exp_{+}), \\
 & \quad (exp_{ide})\text{: } \langle \mathtt{x}, \varphi \rangle \rightarrow_{exp} \underline{\mathtt{5}}, \\
 & \quad (exp_{n})\ \langle \mathtt{8}, \varphi \rangle \rightarrow_{exp} \underline{\mathtt{8}}, \\
 & \quad \underline{\mathtt{13}} = \underline{\mathtt{5}} + \underline{\mathtt{8}}\ \} \\
 & \underline{\mathtt{13}}
\end{array}
$$

■

4.2 Strutturazione dello stato

Per poter trattare correttamente caratteristiche del linguaggio Java più complesse quali l'annidamento dei blocchi e le chiamate ai metodi, è necessario dare allo stato una strutturazione più sofisticata rispetto a quella introdotta nella Sezione 4.1. Intuitivamente, invece di trattare lo stato come una tabella di associazioni tra identificatori e valori, lo vogliamo trattare come una *pila*, ovvero come una sequenza di tabelle, ciascuna delle quali è un insieme di associazioni tra identificatori e valori. Useremo il termine *frame* per fare riferimento ad una tabella[5]. Il nuovo stato è ora una pila di frame. Nel seguito useremo φ, φ', ecc. per denotare frame e $\sigma, \sigma', ecc.$ per denotare stati, ovvero sequenze di frame.

Si consideri la rappresentazione grafica di uno stato σ costituito da due frame φ_1 e φ_2.

$$
\sigma \left\{
\begin{array}{ll}
\varphi_2 & \begin{array}{|c|c|}\hline \mathrm{x} & \underline{\mathtt{31}} \\ \mathrm{y} & \underline{\mathtt{12}} \\ \hline \end{array} \\
\varphi_1 & \begin{array}{|c|c|}\hline \mathrm{z} & \underline{\mathtt{8}} \\ \mathrm{x} & \underline{\mathtt{25}} \\ \hline \end{array}
\end{array}
\right.
$$

La questione cruciale è che in uno stato possono essere presenti contemporaneamente più associazioni per uno stesso identificatore: nell'esempio, per l'identificatore `x` esistono due associazioni nello stato σ, una nel frame φ_1 e l'altra nel frame φ_2. Ma quale è allora il valore di un identificatore in uno stato? Nell'esempio, quale tra i valori $\underline{\mathtt{25}}$ e $\underline{\mathtt{31}}$ è il valore di `x` nello stato σ? È il valore che si trova ricercando un'associazione per `x` a partire dal frame più "recente".

Intuitivamente, essendo uno stato una *pila* di frame (una struttura cioè in cui un frame viene sempre "impilato" sull'ultimo creato), il valore di un identificatore è quello che si trova ricercandolo nei frame dello stato dall'alto verso il basso. Nell'esempio, il valore dell'identificatore `x` è $\underline{\mathtt{31}}$: il frame φ_2 è stato infatti impilato sul frame φ_1.

Vediamo come si può formalizzare la nuova strutturazione dello stato come pila di frame. Osserviamo innanzitutto che il concetto di frame corrisponde essenzialmente al concetto di stato così come definito nella sezione 4.1: in altre parole un frame è una funzione che associa un valore ad un numero finito di identificatori in `Ide`. Seguendo le notazioni già introdotte, indicheremo con $\varphi(\mathtt{x})$ il valore associato all'identificatore `x` nel frame φ, e rappresenteremo con $\varphi(\mathtt{x}) = \bot$ il fatto che il frame φ non associa alcun valore all'identificatore `x`. Infine, indicheremo con ω il frame vuoto, ovvero il frame in cui, per ogni identificatore `x`, $\omega(\mathtt{x}) = \bot$. Nel seguito, indicheremo con Φ l'insieme dei possibili frame.

Uno *stato* è ora una sequenza di frame. Se σ è uno stato e φ un frame, indichiamo con $\varphi.\sigma$ il nuovo stato ottenuto aggiungendo il frame φ alla sequenza σ: nel nuovo stato così ottenuto, φ è il frame *più recente*. Indicando con Ω lo stato vuoto (ovvero la sequenza vuota di frame), l'insieme Σ degli stati può allora essere definito induttivamente come segue:

$$\Sigma = \{\Omega\} \cup \{\varphi.\sigma | \varphi \in \Phi, \sigma \in \Sigma\}.$$

Una formalizzazione dello stato σ dell'esempio precedente è la sequenza:

$$\varphi_2.\varphi_1.\Omega$$

[5]Dunque un frame corrisponde allo stato della sezione 4.1.

dove φ_1 e φ_2 sono le funzioni:

$$\varphi_1(v) = \begin{cases} \underline{8} & \text{se } v{=}\mathtt{z} \\ \underline{25} & \text{se } v{=}\mathtt{x} \\ \perp & \text{altrimenti} \end{cases} \qquad \varphi_2(v) = \begin{cases} \underline{12} & \text{se } v{=}\mathtt{y} \\ \underline{31} & \text{se } v{=}\mathtt{x} \\ \perp & \text{altrimenti} \end{cases}$$

Dato uno stato σ ed un identificatore x, continuiamo ad indicare con $\sigma(\mathtt{x})$ il valore associato ad x in σ[6]. Avremo allora:

$$\sigma(\mathtt{x}) = \begin{cases} \perp & \text{se } \sigma = \Omega \\ \varphi(\mathtt{x}) & \text{se } \sigma = \varphi.\sigma' \text{ e } \varphi(\mathtt{x}) \neq \perp \\ \sigma'(\mathtt{x}) & \text{se } \sigma = \varphi.\sigma' \text{ e } \varphi(\mathtt{x}) = \perp \end{cases}$$

Consideriamo ancora l'esempio precedente, con $\sigma = \varphi_2.\varphi_1.\Omega$. Abbiamo: $\sigma(\mathtt{x}) = \underline{31}$ poiché $\varphi_2(\mathtt{x}) = \underline{31}$. Inoltre, $\sigma(\mathtt{z}) = \underline{8}$ poiché $\varphi_2(\mathtt{z}) = \perp$ e $\varphi_1(\mathtt{z}) = \underline{8}$.

Osserviamo infine che, con la nuova definizione di stato strutturato, le regole date nel paragrafo 4.1.1 rimangono immutate, con l'unica accortezza che, nelle configurazioni del sistema di transizione $\langle \mathtt{E}, \varphi \rangle$ ivi riportate la seconda componente è uno stato strutturato. Riportiamo come esempio la regola (exp_{ide}):

$$\frac{\sigma(\mathtt{x}) = \underline{\mathtt{n}}}{\langle \mathtt{x}, \sigma \rangle \rightarrow_{exp} \underline{\mathtt{n}}} \qquad (exp_{ide})$$

dove, secondo le convenzioni appena adottate, abbiamo indicato con σ la seconda componente delle configurazioni, trattandosi di uno stato strutturato.

4.3 Modifica dello stato

Come vedremo in seguito, il significato di un programma è quello di eseguire transizioni di stato, cioè modificare lo stato. A questo proposito, introduciamo una notazione compatta per esprimere le modifiche di stato: l'effetto di queste ultime è di cambiare il valore associato ad uno o più identificatori. Come abbiamo visto nella precedente sezione, un medesimo stato può contenere più associazioni per lo stesso identificatore e dunque bisogna fare attenzione a quali associazioni sono coinvolte nella modifica.

Introduciamo dapprima una notazione per la modifica di un frame, o meglio del valore associato ad uno o più identificatori in un frame. Siano allora φ un frame, x un identificatore e $\underline{\mathtt{v}}$ un elemento di Val. Denotiamo con $\varphi[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ un frame in cui al nome x è associato $\underline{\mathtt{v}}$, mentre ad ogni altro identificatore y (diverso da x) è associato lo stesso valore associato ad y in φ. Ricordando che un frame φ non è altro che una funzione $\varphi : \mathtt{Ide} \mapsto \mathtt{Val}$, quanto appena detto può essere espresso formalmente in questo modo:

$$\varphi[^{\underline{\mathtt{v}}}/_{\mathtt{x}}](y) = \begin{cases} \varphi(y) & \text{se } y \neq \mathtt{x} \\ \underline{\mathtt{v}} & \text{se } y = \mathtt{x} \end{cases}$$

Si noti che $\varphi[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ è ancora un frame, ovvero una funzione da Ide in Val. La precedente notazione può essere estesa per trattare la modifica di un frame in corrispondenza di più identificatori. Siano dunque φ un frame, $\mathtt{x}_1, \ldots \mathtt{x}_k$ una sequenza di k identificatori *distinti* e $\underline{\mathtt{v}}_1, \ldots \underline{\mathtt{v}}_k$ una sequenza di k elementi in Val. La notazione

$$\varphi[^{\underline{\mathtt{v}}_1}/_{\mathtt{x}_1}, \ldots, ^{\underline{\mathtt{v}}_k}/_{\mathtt{x}_k}]$$

denota il frame:

$$\varphi[^{\underline{\mathtt{v}}_1}/_{\mathtt{x}_1}, \ldots, ^{\underline{\mathtt{v}}_k}/_{\mathtt{x}_k}](y) = \begin{cases} \underline{\mathtt{v}}_1 & \text{se } y = \mathtt{x}_1 \\ \underline{\mathtt{v}}_2 & \text{se } y = \mathtt{x}_2 \\ & \cdots \\ \underline{\mathtt{v}}_k & \text{se } y = \mathtt{x}_k \\ \varphi(y) & \text{altrimenti} \end{cases}$$

[6]Si tratta di un piccolo abuso di notazione, essendo σ una sequenza e non una funzione come φ nella Sezione 4.1.

Come esempio, dato il frame $\varphi = \{\mathtt{x} \mapsto \underline{5}, \mathtt{y} \mapsto \underline{2}\}$ e detto φ_1 il frame $\varphi[^{\underline{4}}/_{\mathtt{x}}, ^{\underline{3}}/_{\mathtt{y}}]$, avremo che $\varphi_1(\mathtt{x}) = \underline{4}$, $\varphi_1(\mathtt{y}) = \underline{3}$ e $\varphi_1(\mathtt{z}) = \perp$ per ogni altro identificatore $\mathtt{z}$.

È importante rimarcare che la condizione che gli identificatori $\mathtt{x}_1, \ldots, \mathtt{x}_k$ siano tutti *distinti* in

$$\varphi[^{\underline{\mathtt{v}}_1}/_{\mathtt{x}_1}, \ldots, ^{\underline{\mathtt{v}}_k}/_{\mathtt{x}_k}]$$

garantisce che quanto ottenuto sia ancora una funzione da $\mathtt{Ide}$ in $\mathtt{Val}$. Laddove fosse necessario rappresentare modifiche successive per lo stesso identificatore, non potremo utilizzare la notazione appena introdotta per modifiche multiple. Così, ad esempio, saremo autorizzati a scrivere $\varphi' = (\varphi[^{\underline{10}}/_{\mathtt{x}}])[^{\underline{20}}/_{\mathtt{x}}]$, ma non $\varphi'' = \varphi[^{\underline{10}}/_{\mathtt{x}}, ^{\underline{20}}/_{\mathtt{x}}]$. Infatti, mentre dalle definizioni precedenti è chiaro che φ' è una funzione e che $\varphi'(\mathtt{x}) = \underline{20}$, è anche chiaro che φ'' non è una *funzione* da $\mathtt{Ide}$ in $\mathtt{Val}$ (quale sarebbe, infatti, l'*unico* valore associato ad $\mathtt{x}$ in φ''?).

Come ulteriore esempio, dato il frame $\varphi = \{\mathtt{x} \mapsto \underline{5}, \mathtt{y} \mapsto \underline{2}\}$, il frame $\varphi[^{\underline{4}}/_{\mathtt{x}}, ^{\underline{3}}/_{\mathtt{z}}]$ è uguale a $\varphi' = \{\mathtt{x} \mapsto \underline{4}, \mathtt{y} \mapsto \underline{2}, \mathtt{z} \mapsto \underline{3}\}$ (verificarlo per esercizio).

Veniamo ora alla modifica di stato, riservando per essa la stessa notazione usata per la modifica di frame: $\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ indica la modifica del valore associato ad $\mathtt{x}$ nello stato σ. Intuitivamente, così come il valore associato ad un identificatore $\mathtt{x}$ in uno stato è quello che si determina trovando una associazione per $\mathtt{x}$ a partire dal frame più recente, anche la modifica $\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ riguarda l'associazione più recente per $\mathtt{x}$. Detto altrimenti, nello stato $\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ viene modificato il frame più recente che contiene una associazione per $\mathtt{x}$. Consideriamo ad esempio uno stato σ del tipo

$$\varphi_1.\varphi_2 \ldots \varphi_i \ldots \varphi_k.\Omega$$

Se φ_i è il primo frame nella sequenza che contiene una associazione per $\mathtt{x}$ (o detto altrimenti $\varphi_1(\mathtt{x}) = \varphi_2(\mathtt{x}) = \ldots = \varphi_{i-1}(\mathtt{x}) = \perp$ e $\varphi_i(\mathtt{x}) \neq \perp$), lo stato $\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ non è altro che la sequenza modificata

$$\varphi_1.\varphi_2 \ldots \varphi_i[^{\underline{\mathtt{v}}}/_{\mathtt{x}}] \ldots \varphi_k.\Omega$$

Si noti che, se nessun frame nello stato contiene una associazione per $\mathtt{x}$, la modifica non ha alcun effetto. Nell'esempio, se $\varphi_1(\mathtt{x}) = \varphi_2(\mathtt{x}) = \ldots = \varphi_k(\mathtt{x}) = \perp$, allora $\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}] = \sigma$.

Una definizione formale di $\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ può essere data nel modo seguente.

$$\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}] = \begin{cases} \varphi[^{\underline{\mathtt{v}}}/_{\mathtt{x}}].\sigma' & \text{se } \sigma = \varphi.\sigma' \text{ e } \varphi(\mathtt{x}) \neq \perp \\ \varphi.\sigma'[^{\underline{\mathtt{v}}}/_{\mathtt{x}}] & \text{se } \sigma = \varphi.\sigma' \text{ e } \varphi(\mathtt{x}) = \perp \\ \Omega & \text{se } \sigma = \Omega \end{cases}$$

È chiaro che la notazione può essere estesa facilmente per trattare la modifica di una sequenza di identificatori *distinti*, che indicheremo con

$$\sigma[^{\underline{\mathtt{v}}_1}/_{\mathtt{x}_1}, \ldots, ^{\underline{\mathtt{v}}_k}/_{\mathtt{x}_k}].$$

Consideriamo come esempio lo stato $\sigma = \varphi_2.\varphi_1.\Omega$, dove φ_1 e φ_2 sono i frame seguenti:

$$\varphi_1(v) = \begin{cases} \underline{8} & \text{se } v = \mathtt{z} \\ \underline{25} & \text{se } v = \mathtt{x} \\ \perp & \text{altrimenti} \end{cases} \qquad \varphi_2(v) = \begin{cases} \underline{12} & \text{se } v = \mathtt{y} \\ \underline{31} & \text{se } v = \mathtt{x} \\ \perp & \text{altrimenti} \end{cases}$$

Lo stato $\sigma[^{\underline{50}}/_{\mathtt{z}}, ^{\underline{40}}/_{\mathtt{x}}]$ è la sequenza $\varphi_2'.\varphi_1'.\Omega$, con $\varphi_1' = \varphi_1[^{\underline{50}}/_{\mathtt{z}}]$ e $\varphi_2' = \varphi_2[^{\underline{40}}/_{\mathtt{x}}]$. D'altra parte, lo stato $\sigma[^{\underline{10}}/_{\mathtt{w}}] = \sigma$ dal momento che φ_2 e φ_1 non contengono una associazione per $\mathtt{w}$.

Concludiamo questa sezione osservando alcune differenze tra le possibili notazioni introdotte sin qui riguardo a stati e frame. Innanzitutto, è importante osservare che la scrittura $\varphi(\mathtt{x}) = \underline{\mathtt{v}}$ (risp. $\sigma(\mathtt{x}) = \underline{\mathtt{v}}$) stabilisce semplicemente che il valore associato ad $\mathtt{x}$ nel frame φ (risp. nello stato σ) è $\underline{\mathtt{v}}$, mentre è *ignoto* (e non necessariamente *indefinito*) il valore associato ad ogni altro identificatore che non sia $\mathtt{x}$. La scrittura $\varphi(\mathtt{x}) = \underline{\mathtt{v}}$ non equivale a dire che il frame φ contiene una sola associazione (quella per l'identificatore $\mathtt{x}$): non corrisponde cioè a stabilire che $\varphi = \{\mathtt{x} \mapsto \underline{\mathtt{v}}\}$. Infatti, la scrittura $\{\mathtt{x} \mapsto \underline{\mathtt{v}}\}$ indica un frame in cui il valore associato ad $\mathtt{x}$ è $\underline{\mathtt{v}}$, mentre è *indefinito* ($\perp$) il valore associato ad ogni altro identificatore. Inoltre, entrambe le scritture precedenti non equivalgono a $\varphi[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$, in quanto quest'ultima sta ad indicare un frame che associa ad $\mathtt{x}$ il valore $\underline{\mathtt{v}}$ e si comporta come φ, qualunque esso sia, in corrispondenza di ogni altro identificatore. Concludiamo infine notando che mentre $\varphi[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ rappresenta comunque un frame diverso da φ (con la particolarissima eccezione in cui $\varphi(\mathtt{x}) = \underline{\mathtt{v}}$), $\sigma[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ può essere una sequenza di frame identica a σ.

Esercizi

1. Dire quali sono i valori associati agli identificatori $\mathtt{x}$ ed $\mathtt{y}$ nel frame φ in tutti i casi seguenti:
   (a) $\varphi = \{\mathtt{x} \mapsto \underline{10}, \mathtt{y} \mapsto \underline{10}\}$
   (b) $\varphi = \{\mathtt{x} \mapsto \underline{10}, \mathtt{z} \mapsto \underline{10}\}$
   (c) $\varphi = \varphi'$
   (d) $\varphi = \varphi'[^{\underline{10}}/_{\mathtt{y}}]$
   (e) $\varphi = \varphi'[^{\underline{10}}/_{\mathtt{y}}, ^{\underline{20}}/_{\mathtt{x}}]$
   (f) $\varphi = (\varphi'[^{\underline{10}}/_{\mathtt{y}}, ^{\underline{20}}/_{\mathtt{x}}])[^{\underline{10}}/_{\mathtt{x}}]$

2. Siano $\varphi = \{\mathtt{x} \mapsto \underline{10}, \mathtt{y} \mapsto \underline{10}\}$ e $\varphi' = \{\mathtt{y} \mapsto \underline{10}, \mathtt{x} \mapsto \underline{10}, \mathtt{z} \mapsto \underline{10}\}$. Mostrare che φ' e $\varphi[^{\underline{10}}/_{\mathtt{z}}]$ denotano lo stesso frame (ovvero la stessa funzione).
3. Mostrare che $(\varphi[^{\underline{\mathtt{v}}'}/_{\mathtt{x}}])[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ e $\varphi[^{\underline{\mathtt{v}}}/_{\mathtt{x}}]$ denotano lo stesso frame, qualunque siano $\varphi, \underline{\mathtt{v}}, \underline{\mathtt{v}}'$ e $\mathtt{x}$.
4. Siano
   $\varphi_1 = \{\mathtt{x} \mapsto \underline{10}, \mathtt{y} \mapsto \underline{10}\}$
   e
   $\varphi_2 = \{\mathtt{x} \mapsto \underline{9}, \mathtt{z} \mapsto \underline{10}\}$.
   Dire quali sono i valori associati agli identificatori $\mathtt{x}$, $\mathtt{y}$ e $\mathtt{z}$ nello stato σ in tutti i casi seguenti:
   (a) $\sigma = \varphi_1.\varphi_2.\Omega$
   (b) $\sigma = \varphi_2.\varphi_1.\Omega$
   (c) $\sigma = \varphi_1.\varphi_2[^{\underline{8}}/_{\mathtt{x}}].\Omega$
   (d) $\sigma = \varphi_1[^{\underline{8}}/_{\mathtt{x}}].\varphi_2.\Omega$

5 La semantica operazionale del nucleo di Java

A partire da questo paragrafo affronteremo lo studio della semantica in stile operazionale di un frammento di linguaggio di programmazione: il nucleo imperativo di Java. Ci limiteremo a considerare i tipi di dato elementari `int` e `boolean`, le dichiarazioni di variabili limitate ai tipi suddetti ed i costrutti imperativi di base, nelle loro forme più semplici: assegnamento, blocco, condizionale, comando iterativo `while`. Per la sua eccessiva semplicità il frammento da noi considerato non si presenta come un linguaggio di programmazione reale, ma solo come uno strumento per lo studio dei concetti essenziali nella semantica dei linguaggi.

5.1 Le espressioni e la loro semantica

Il primo aspetto da affrontare nello studio di un linguaggio riguarda i meccanismi che esso mette a disposizione per rappresentare e manipolare dati. Nel nostro caso, i tipi di dato presenti nel linguaggio sono solamente valori interi e logici (booleani) ed i costrutti per manipolarli consentono di definire espressioni sia a valori interi che a valori booleani. Questi ultimi sono i valori di verità nell'insieme $\mathbf{B} = \{tt, ff\}$ che verranno rappresentati sintatticamente con i simboli `true` e `false` rispettivamente: il linguaggio mette a disposizione una serie di operatori per definire espressioni (le *proposizioni*) il cui valore è un valore di verità (ad esempio `5>3`) e per combinare tali espressioni. Il significato di tali operatori viene dato formalmente in questo paragrafo. La sintassi delle espressioni è la seguente:

```
Exp ::= Const_val | Ide | (Exp) | Exp Op Exp | !Exp
Op ::= + | - | * | / | % | == | != | < | <= | > | >= | && | ||
Const_val ::= Num | Bool
Bool ::= true | false
Num ::= ...
```

Nel seguito, ometteremo la distinzione esplicita tra *valori* e *rappresentazioni* e utilizzeremo le seguenti convenzioni:

- $\mathtt{n}, \mathtt{m}, \mathtt{n}', \mathtt{m}', \ldots$ denotano valori in $\mathbf{N}$, o loro rappresentazioni ;
- $\mathtt{b}, \mathtt{b}', \mathtt{b}_1, \ldots$ denotano valori in $\mathbf{B}$, o loro rappresentazioni;
- $\mathtt{E}, \mathtt{E}', \ldots$ denotano generiche espressioni
- $\mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots$ denotano identificatori.

Abbiamo già discusso nel paragrafo 3.2 la semantica delle espressioni a valori naturali. Per quanto concerne le espressioni a valori booleani, osserviamo che gli operatori `&&`, `||` e `!` corrispondono agli operatori logici $\wedge, \vee$ e $\neg$. Come nel caso delle espressioni a valori naturali, le regole che definiscono la semantica delle espressioni booleane sono guidate dalla sintassi e, in esse, utilizziamo delle premesse che corrispondono ad operazioni "elementari" tra valori di verità che si suppone di saper calcolare.

$$\langle \mathtt{true}, \sigma \rangle \rightarrow_{exp} tt \qquad (exp_{true})$$

$$\langle \mathtt{false}, \sigma \rangle \rightarrow_{exp} ff \qquad (exp_{false})$$

Le due regole precedenti corrispondono allo schema di regola (exp_n) per le costanti naturali.

$$\frac{\langle \mathtt{E}, \sigma \rangle \rightarrow_{exp} \mathtt{b} \qquad \langle \mathtt{E}', \sigma \rangle \rightarrow_{exp} \mathtt{b}' \qquad \mathtt{b}'' = \mathtt{b} \ \vee \ \mathtt{b}'}{\langle \mathtt{E} \mid\mid \mathtt{E}', \sigma \rangle \rightarrow_{exp} \mathtt{b}''} \qquad (exp_{or})$$

$$\frac{\langle \mathtt{E}, \sigma \rangle \rightarrow_{exp} \mathtt{b} \qquad \langle \mathtt{E}', \sigma \rangle \rightarrow_{exp} \mathtt{b}' \qquad \mathtt{b}'' = \mathtt{b} \ \wedge \ \mathtt{b}'}{\langle \mathtt{E} \ \&\& \ \mathtt{E}', \sigma \rangle \rightarrow_{exp} \mathtt{b}''} \qquad (exp_{and})$$

$$\frac{\langle \mathtt{E}, \sigma \rangle \rightarrow_{exp} \mathtt{b} \qquad \mathtt{b}' = \neg \mathtt{b}}{\langle !\mathtt{E}, \sigma \rangle \rightarrow_{exp} \mathtt{b}'} \qquad (exp_{not})$$

Queste regole definiscono gli operatori del linguaggio `&&`, `||` e `!` in termini dei corrispondenti operatori logici $\wedge, \vee$ e $\neg$ [7].

$$\frac{\langle \mathtt{E}, \sigma\rangle \rightarrow_{exp} \mathtt{n} \qquad \langle \mathtt{E}', \sigma\rangle \rightarrow_{exp} \mathtt{n}' \qquad \mathtt{b} = \mathtt{n}\ rel\ \mathtt{n}'}{\langle \mathtt{E\ rel\ E}', \sigma\rangle \rightarrow_{exp} \mathtt{b}} \qquad (exp_{rel})$$

Con questa regola abbiamo in realtà espresso tutte le regole che riguardano gli operatori di relazione (confronto) previsti nella sintassi del linguaggio. Il simbolo `rel` sta per uno qualunque dei sei simboli sintattici `==`, `!=`, $<$, $>$, $>=$ e $<=$ previsti dal linguaggio, mentre il simbolo *rel* sta per la corrispondente funzione semantica ($=, \neq, \prec, \succ, \succeq, \preceq$ rispettivamente). Dunque (exp_{rel}) esprime contemporaneamente sei regole.

5.2 I costrutti di controllo e la loro semantica

La porzione di linguaggio vista fino ad ora (le espressioni) fornisce i meccanismi di base a nostra disposizione per rappresentare dati e per manipolarli. Un linguaggio che si limitasse a ciò consentirebbe di risolvere problemi la cui soluzione si limita semplicemente al calcolo di espressioni. Tuttavia i problemi che si affrontano nella pratica sono quasi sempre di natura più complessa e la loro soluzione in termini di puro calcolo di espressioni risulterebbe troppo complicata se non impossibile.

In questo paragrafo studiamo un'ulteriore classe di costrutti linguistici, usualmente denominati *comandi*, il cui scopo è essenzialmente quello di consentire la modifica dello stato[8]. Fino ad ora, quest'ultimo è stato introdotto al solo scopo di rappresentare associazioni tra nomi e valori, nomi utilizzabili poi all'interno delle espressioni: in realtà, nei linguaggi imperativi (quelli che stiamo considerando) il ruolo dello stato è ben più rilevante. Un problema di programmazione, in tali linguaggi, si pone proprio in termini della descrizione di uno stato *iniziale* (che rappresenta i *dati* del problema) e di uno stato *finale* (che rappresenta i *risultati* del problema). La soluzione ad un problema siffatto consiste allora nella individuazione di una sequenza di azioni che modificano lo stato iniziale fino a trasformarlo nello stato finale desiderato.

Nel nostro semplice frammento di Java la sintassi dei comandi è data mediante il seguente insieme di produzioni:

```
Com ::=
    Ide = Exp; |
    Block |
    if (Exp) Com else Com |
    while (Exp) Com

Block ::= {Stat_List}
Stat_List ::= Com | Com Stat_List
```

Le configurazioni del sistema di transizioni per i comandi, $\mathtt{S}_{com}$, sono di due tipi: $\langle \mathtt{C}, \sigma\rangle$, dove `C` è un comando e σ uno stato, sono le configurazioni che corrispondono agli stadi iniziali del calcolo (cfr. le configurazioni $\langle \mathtt{E}, \sigma\rangle$ nel sistema $\mathtt{S}_{exp}$), mentre le configurazioni composte da un solo stato σ sono le configurazioni terminali, che corrispondono alla conclusione del calcolo. Riassumendo:

$$\begin{aligned}\Gamma_{com} &= \{\langle \mathtt{C}, \sigma\rangle \mid \mathtt{C} \in \mathtt{Com},\ \ \sigma \in \Sigma\} \cup \\ & \{\sigma \mid \sigma \in \Sigma\} \\ \\ \mathtt{T}_{com} &= \{\sigma \mid \sigma \in \Sigma\}\end{aligned}$$

[7] Rammentiamo che, se b, b' sono elementi di $\mathbf{B}$, $b \wedge b'$ ha valore *tt* se e solo se b e b' sono entrambi *tt*, $b \vee b'$ ha valore *tt* se e solo se almeno uno tra b e b' è *tt* e $\neg b$ è *tt* se e solo se b è *ff*.

[8] Più avanti ci occuperemo anche di introdurre le dichiarazioni, ovvero i costrutti che consentono di definire l'insieme delle associazioni che compongono uno stato.

Le regole che definiscono le transizioni di stato per i comandi sono le seguenti:

$$\frac{\langle \mathtt{E}, \sigma\rangle \rightarrow_{exp} \mathtt{v}}{\langle \mathtt{x=E;}, \sigma\rangle \rightarrow_{com} \sigma[^{\mathtt{v}}/_{\mathtt{x}}]} \qquad (com_{=})$$

Il comando di assegnamento è il costrutto del linguaggio che consente di apportare modifiche allo stato. Si noti l'utilizzo della notazione $\sigma[^{\mathtt{v}}/_{\mathtt{x}}]$ introdotta nella Sezione 4.2 per rappresentare la modifica di stato.

Tutti gli altri comandi servono per controllare l'uso degli assegnamenti, ovvero delle modifiche di stato, e per questo vengono chiamati *costrutti di controllo*.

Il blocco consente di rappresentare una sequenza di comandi, da eseguire nell'ordine in cui essi compaiono nella sequenza.

$$\frac{\langle \mathtt{Slist}, \sigma\rangle \rightarrow_{com} \sigma'}{\langle \{\mathtt{Slist}\}, \sigma\rangle \rightarrow_{com} \sigma'} \qquad (com_{Block})$$

$$\frac{\langle \mathtt{S}, \sigma\rangle \rightarrow_{com} \sigma'' \qquad \langle \mathtt{Slist}, \sigma''\rangle \rightarrow_{com} \sigma'}{\langle \mathtt{S\ \ Slist}, \sigma\rangle \rightarrow_{com} \sigma'} \qquad (com_{Stat-list})$$

La regola $(com_{Stat-list})$ esprime il fatto che, in una sequenza del tipo $\mathtt{S}_1\mathtt{S}_2 \ldots \mathtt{S}_k$, la sottosequenza $\mathtt{S}_2 \ldots \mathtt{S}_k$ inizia ad operare nello stato che risulta dal calcolo corrispondente al primo comando $\mathtt{S}_1$ dell'intera sequenza.

Esempio 5.1 Consideriamo la lista di comandi:

```
x = 2; y = 3; x = x-1;
```

e lo stato $\sigma_0 = \{\mathtt{x} \mapsto \underline{0}, \mathtt{y} \mapsto \underline{0}\}.\Omega$. Abbiamo dunque la derivazione:

$$\begin{array}{ll}
 & \langle \mathtt{x = 2; y = 3; x = x - 1;}, \sigma_0\rangle \\
\rightarrow_{com} & \{\ (com_{Stat-list}), \\
 & \quad \text{(d1)}: \langle \mathtt{x = 2;}, \sigma_0\rangle \rightarrow_{com} \sigma_0[^{\underline{2}}/_{\mathtt{x}}], \\
 & \quad\qquad \sigma_1 = \sigma_0[^{\underline{2}}/_{\mathtt{x}}], \\
 & \quad \text{(d2)}: \langle \mathtt{y = 3; x = x - 1;}, \sigma_1\rangle \rightarrow_{com} \sigma_1[^{\underline{1}}/_{\mathtt{x}}, ^{\underline{3}}/_{\mathtt{y}}] \\
 & \quad\qquad \sigma_1[^{\underline{1}}/_{\mathtt{x}}, ^{\underline{3}}/_{\mathtt{y}}] = \sigma_0[^{\underline{1}}/_{\mathtt{x}}, ^{\underline{3}}/_{\mathtt{y}}]\} \\
 & \sigma_0[^{\underline{1}}/_{\mathtt{x}}, ^{\underline{3}}/_{\mathtt{y}}] = \{\mathtt{x} \mapsto \underline{1}, \mathtt{y} \mapsto \underline{3}\}.\Omega
\end{array}$$

dove (d1) è la derivazione:

(d1):

$$\begin{array}{ll}
 & \langle \mathtt{x = 2;}, \sigma_0\rangle \\
\rightarrow_{com} & \{\ (com_{=}), \\
 & \quad \mathcal{E}[\![\mathtt{2}]\!]\sigma_0 = \underline{2}\} \\
 & \sigma_0[^{\underline{2}}/_{\mathtt{x}}]
\end{array}$$

e, detto σ_1 lo stato $\sigma_0[^{\underline{2}}/_{\mathtt{x}}]$, (d2) è la derivazione:

(d2):

$$\langle \texttt{y} = \texttt{3}; \texttt{x} = \texttt{x} - \texttt{1};, \sigma_1\rangle$$

$$\begin{array}{ll} \rightarrow_{com} & \{\ (com_{Stat-list}), \\ & \quad (\text{d3}){:}\ \langle \texttt{y} = \texttt{3};, \sigma_1\rangle \rightarrow_{com} \sigma_1[^{\underline{3}}/_{\texttt{y}}], \\ & \quad \phantom{(\text{d3}){:}}\ \sigma' = \sigma_1[^{\underline{3}}/_{\texttt{y}}], \\ & \quad (\text{d4}){:}\ \langle \texttt{x} = \texttt{x} - \texttt{1};, \sigma'\rangle \rightarrow_{com} \sigma'[^{\underline{1}}/_{\texttt{x}}], \\ & \quad \phantom{(\text{d4}){:}}\ \sigma'[^{\underline{1}}/_{\texttt{x}}] = \sigma_0[^{\underline{1}}/_{\texttt{x}}, ^{\underline{3}}/_{\texttt{y}}]\} \end{array}$$

$$\sigma_0[^{\underline{1}}/_{\texttt{x}}, ^{\underline{3}}/_{\texttt{y}}]$$

Infine, le sotto-derivazioni (d3) e (d4) sono le seguenti:

(d3):

$$\langle \texttt{y} = \texttt{3};, \sigma_0[^{\underline{2}}/_{\texttt{x}}]\rangle$$

$$\begin{array}{ll} \rightarrow_{com} & \{\ (com_{=}), \\ & \quad \mathcal{E}[\![\texttt{3}]\!]\sigma_0[^{\underline{2}}/_{\texttt{x}}] = \underline{3}\} \end{array}$$

$$\sigma_0[^{\underline{2}}/_{\texttt{x}}, ^{\underline{3}}/_{\texttt{y}}]$$

(d4):

$$\langle \texttt{x} = \texttt{x} - \texttt{1};, \sigma_0[^{\underline{2}}/_{\texttt{x}}, ^{\underline{3}}/_{\texttt{y}}]\rangle$$

$$\begin{array}{ll} \rightarrow_{com} & \{\ (com_{=}), \\ & \quad \mathcal{E}[\![\texttt{x} - \texttt{1}]\!]\sigma_0[^{\underline{2}}/_{\texttt{x}}, ^{\underline{3}}/_{\texttt{y}}]\rangle = \underline{1}\} \end{array}$$

$$\sigma_0[^{\underline{1}}/_{\texttt{x}}, ^{\underline{3}}/_{\texttt{y}}]$$

■

Trattandosi del primo esempio di derivazione in $\rightarrow_{com}$ abbiamo esplicitato tutte le sottoderivazioni e le loro giustificazioni. Si noti che le uguaglianze del tipo $\sigma' = \sigma[\ldots]$ che compaiono nei vari passi, non corrispondono a premesse di regole, ma semplicemente all'introduzione di nomi di comodo per gli stati intermedi del calcolo.

Anche le sottoderivazioni che corrispondono a singoli comandi di assegnamento sono sufficientemente semplici da poter essere esplicitate direttamente nelle giustificazioni. Adotteremo di qui in poi questa convenzione, messa in luce nell'esempio seguente.

Esempio 5.2 Consideriamo di nuovo il comando dell'esempio precedente.

$$\langle \texttt{x} = \texttt{2}; \texttt{y} = \texttt{3}; \texttt{x} = \texttt{x} - \texttt{1};, \sigma_0\rangle$$

$$\begin{array}{ll} \rightarrow_{com} & \{\ (com_{Stat-list}), \\ & \quad (com_{=}){:}\ \mathcal{E}[\![\texttt{2}]\!]\sigma_0 = \underline{2}, \\ & \quad \phantom{(com_{=}){:}}\ \langle \texttt{x} = \texttt{2};, \sigma_0\rangle \rightarrow_{com} \sigma_0[^{\underline{2}}/_{\texttt{x}}], \\ & \quad \phantom{(com_{=}){:}}\ \sigma_1 = \sigma_0[^{\underline{2}}/_{\texttt{x}}], \\ & \quad (\text{d1}){:}\ \langle \texttt{y} = \texttt{3}; \texttt{x} = \texttt{x} - \texttt{1};, \sigma_1\rangle \rightarrow_{com} \sigma_1[^{\underline{1}}/_{\texttt{x}}, ^{\underline{3}}/_{\texttt{y}}] \\ & \quad \phantom{(\text{d1}){:}}\ \sigma_1[^{\underline{1}}/_{\texttt{x}}, ^{\underline{3}}/_{\texttt{y}}] = \sigma_0[^{\underline{1}}/_{\texttt{x}}, ^{\underline{3}}/_{\texttt{y}}]\} \end{array}$$

$$\sigma_0[^{\underline{1}}/_{\texttt{x}}, ^{\underline{3}}/_{\texttt{y}}] = \{\texttt{x} \mapsto \underline{1}, \texttt{y} \mapsto \underline{3}\}.\Omega$$

dove, detto σ_1 lo stato $\sigma_0[^{\underline{2}}/_{\texttt{x}}]$, (d1) è la sottoderivazione:

(d1):

$$\langle \texttt{y} = \texttt{3}; \texttt{x} = \texttt{x} - \texttt{1};, \sigma_1\rangle$$

$$
\begin{array}{ll}
\rightarrow_{com} & \{\ (com_{Stat-list}), \\
& \quad (com_{=}) : \mathcal{E}[\![3]\!]\sigma_1 = \underline{3}, \\
& \qquad\qquad \langle \mathtt{y = 3;}, \sigma_1\rangle \rightarrow_{com} \sigma_1[^{\underline{3}}/_{\mathtt{y}}], \\
& \qquad\qquad \sigma' = \sigma_1[^{\underline{3}}/_{\mathtt{y}}], \\
& \quad (com_{=}) : \mathcal{E}[\![\mathtt{x}-1]\!]\sigma' = \underline{1}, \quad \langle \mathtt{x = x - 1;}, \sigma'\rangle \rightarrow_{com} \sigma'[^{\underline{1}}/_{\mathtt{x}}] \\
& \qquad\qquad \sigma'[^{\underline{1}}/_{\mathtt{x}}] = \sigma_0[^{\underline{1}}/_{\mathtt{x}},^{\underline{3}}/_{\mathtt{y}}]\} \\
& \sigma_0[^{\underline{1}}/_{\mathtt{x}},^{\underline{3}}/_{\mathtt{y}}]
\end{array}
$$

■

Veniamo infine alle regole per il comando condizionale e per il comando iterativo.

$$
\frac{\langle \mathtt{E}, \sigma\rangle \rightarrow_{exp} tt \qquad \langle \mathtt{C1}, \sigma\rangle \rightarrow_{com} \sigma'}{\langle \mathtt{if\ (E)\ \ C1\ else\ C2}, \sigma\rangle \rightarrow_{com} \sigma'} \quad (com_{if-tt})
$$

$$
\frac{\langle \mathtt{E}, \sigma\rangle \rightarrow_{exp} f\!f \qquad \langle \mathtt{C2}, \sigma\rangle \rightarrow_{com} \sigma'}{\langle \mathtt{if\ (E)\ \ C1\ else\ C2}, \sigma\rangle \rightarrow_{com} \sigma'} \quad (com_{if-f\!f})
$$

Il comando condizionale consente di scegliere come proseguire il calcolo a seconda del verificarsi o meno di una condizione, rappresentata dall'espressione logica `E`.

$$
\frac{\langle \mathtt{E}, \sigma\rangle \rightarrow_{exp} f\!f}{\langle \mathtt{while\ (E)\ C}, \sigma\rangle \rightarrow_{com} \sigma} \quad (com_{while-f\!f})
$$

$$
\frac{\langle \mathtt{E}, \sigma\rangle \rightarrow_{exp} tt \qquad \langle \mathtt{C}, \sigma\rangle \rightarrow_{com} \sigma'' \qquad \langle \mathtt{while\ (E)\ C}, \sigma''\rangle \rightarrow_{com} \sigma'}{\langle \mathtt{while\ (E)\ C}, \sigma\rangle \rightarrow_{com} \sigma'} \quad (com_{while-tt})
$$

Il comando di iterazione `while (E) C` consente di ripetere l'esecuzione di `C` (detto *corpo*) finché la condizione `E` (detta *guardia*) è verificata. Ciò si riflette in due regole: la prima che identifica la semantica del comando `while` con quella del comando vuoto nel caso in cui la guardia sia falsa; la seconda esprime invece il fatto che, quando la guardia è vera, l'intero comando viene eseguito di nuovo dopo aver eseguito una volta il corpo. Il lettore attento avrà notato che la regola $(com_{while-tt})$ ha come premessa una transizione che riguarda il comando stesso. Non si tratta, tuttavia, di una definizione circolare in quanto la componente stato della configurazione è diversa.

Esempio 5.3 Vediamo come esempio la semantica del comando `while (x>0) x=x-1;` a partire dallo stato $\sigma = \{x \mapsto \underline{2}\}.\Omega$.

$$
\begin{array}{ll}
& \langle \mathtt{while\ (x>0)\ x=x-1;}, \{\mathtt{x} \mapsto \underline{2}\}.\Omega\rangle \\
\rightarrow_{com} & \{(com_{while-tt}) : \\
& \quad \mathcal{E}[\![\mathtt{x} > 0]\!]\{\mathtt{x} \mapsto \underline{2}\}.\Omega = tt, \\
& \quad (com_{=}) : \mathcal{E}[\![\mathtt{x\text{-}1}]\!]\{\mathtt{x} \mapsto \underline{2}\}.\Omega = \underline{1} \\
& \qquad\qquad \langle \mathtt{x=x-1;}, \{\mathtt{x} \mapsto \underline{2}\}.\Omega\rangle \rightarrow_{com} \{\mathtt{x} \mapsto \underline{1}\}.\Omega, \\
& \quad (\mathrm{d1}): \langle \mathtt{while\ (x>0)\ x=x-1;}, \{\mathtt{x} \mapsto \underline{1}\}.\Omega\rangle \rightarrow_{com} \{\mathtt{x} \mapsto \underline{0}\}.\Omega\ \}
\end{array}
$$

$\{\mathtt{x} \mapsto \underline{0}\}.\Omega$

(d1):

$\langle \mathtt{while\ (x>0)\ x=x-1;}, \{\mathtt{x} \mapsto \underline{1}\}.\Omega \rangle$

$\rightarrow_{com}$ { $(com_{while-tt})$:
$\mathcal{E}[\![\mathtt{x} > \mathtt{0}]\!]\{\mathtt{x} \mapsto \underline{1}\}.\Omega = tt$,
$(com_{=})$: $\mathcal{E}[\![\mathtt{x-1}]\!]\{\mathtt{x} \mapsto \underline{1}\}.\Omega = \underline{0}$,
$\langle \mathtt{x=x-1;}, \{\mathtt{x} \mapsto \underline{1}\}.\Omega \rangle \rightarrow_{com} \{\mathtt{x} \mapsto \underline{0}\}.\Omega$,
(d2): $\langle \mathtt{while\ (x>0)\ x=x-1;}, \{\mathtt{x} \mapsto \underline{0}\}.\Omega \rangle \rightarrow_{com} \{\mathtt{x} \mapsto \underline{0}\}.\Omega$ }

$\{\mathtt{x} \mapsto \underline{0}\}.\Omega$

(d2):

$\langle \mathtt{while\ (x>0)\ x=x-1;}, \{\mathtt{x} \mapsto \underline{0}\}.\Omega \rangle$

$\rightarrow_{com}$ { $(com_{while-ff})$:
$\mathcal{E}[\![\mathtt{x} > \mathtt{0}]\!]\{\mathtt{x} \mapsto \underline{0}\}.\Omega = ff$ }

$\{\mathtt{x} \mapsto \underline{0}\}.\Omega$

■

Si noti che anche nel caso del sistema $\mathtt{S}_{com}$ esistono configurazioni *bloccate*, alle quali cioè non è possibile applicare alcuna transizione. Un esempio di configurazione siffatta è la configurazione $\langle \mathtt{x} = \mathtt{5} - \mathtt{8};, \sigma \rangle$: l'unica regola potenzialmente applicabile è chiaramente $(com_{=})$, che però non può essere applicata in quanto non vi è modo di soddisfare la sua premessa (non esiste infatti alcun $\underline{\mathtt{v}} \in \mathbf{N}$ per cui $\mathcal{E}[\![\mathtt{5} - \mathtt{8}]\!]\sigma = \underline{\mathtt{v}}$). Le configurazioni bloccate di questo tipo corrispondono a situazioni di errore. Si noti come, in generale, il fatto che una configurazione sia bloccata o meno dipende dallo stato: ad esempio, la configurazione $\langle \mathtt{x\ /\ y}, \sigma \rangle$ è bloccata se, nello stato σ, il valore associato ad $\mathtt{y}$ è $\underline{0}$, non lo è altrimenti. Errori di questo tipo sono detti *dinamici*, in quanto rilevabili solo al momento dell'esecuzione. Un'altra categoria di errori dinamici corrisponde a configurazioni che danno origine ad una sequenza *infinita* di passi di derivazione. Si consideri ad esempio la configurazione $\langle \mathtt{while}\ (\mathtt{x} > \mathtt{0})\ \mathtt{x} = \mathtt{x} + \mathtt{1};, \{\mathtt{x} \mapsto \underline{2}\}.\Omega \rangle$. L'unica regola del sistema $\mathtt{S}_{com}$ potenzialmente applicabile è chiaramente $(com_{while-tt})$, dal momento che $\mathcal{E}[\![\mathtt{x} > \mathtt{0}]\!]\{\mathtt{x} \mapsto \underline{2}\}.\Omega = tt$. L'applicazione della regola $(com_{while-tt})$ richiede però, tra le premesse, di determinare una transizione del tipo $\langle \mathtt{while}\ (\mathtt{x} > \mathtt{0})\ \mathtt{x} = \mathtt{x} + \mathtt{1};, \{\mathtt{x} \mapsto \underline{3}\}.\Omega \rangle \rightarrow_{com} \sigma$, per qualche σ. Anche in questo caso l'unica regola potenzialmente applicabile è $(com_{while-tt})$ (essendo $\mathcal{E}[\![\mathtt{x} > \mathtt{0}]\!]\{\mathtt{x} \mapsto \underline{3}\}.\Omega = tt$), ed ancora una delle sue premesse corrisponde a determinare una transizione del tipo $\langle \mathtt{while}\ (\mathtt{x} > \mathtt{0})\ \mathtt{x} = \mathtt{x} + \mathtt{1};, \{\mathtt{x} \mapsto \underline{4}\}.\Omega \rangle \rightarrow_{com} \sigma$. È facile convincersi che questo procedimento non ha mai termine (dal momento che nessuno degli stati intermedi del calcolo corrisponde mai ad una situazione in cui la guardia del comando ha valore *ff*). Nella nostra semantica operazionale le configurazioni bloccate corrispondenti ai due tipi di errori dinamici appena visti non sono distinguibili, anche se esse corrispondono a situazioni diverse dal punto di vista del calcolo. Altri approcci alla semantica operazionale dei linguaggi consentono invece di distinguere vari tipi di configurazioni bloccate.

Esercizi

1. Valutare il blocco

   `{x=y;y=x;}`

   a partire dallo stato σ nei seguenti casi:

   (a) $\sigma = \{\mathtt{x} \mapsto \underline{5}, \mathtt{y} \mapsto \underline{10}\}.\Omega$
   (b) $\sigma = \{\mathtt{x} \mapsto \underline{5}, \mathtt{y} \mapsto \underline{5}\}.\Omega$

2. Valutare il blocco

   `{z=x; x=y; y=z;}`

   a partire dallo stato $\sigma = \{\mathtt{x} \mapsto \underline{5}, \mathtt{y} \mapsto \underline{10}, z \mapsto 0\}.\Omega$

3. Valutare il blocco

   $\{$`z= x/y; w=x%y;`

   `if (w==0) z=0; else z=z+1;`$\}$

   a partire dallo stato σ nei seguenti casi:

   (a) $\sigma = \{\mathtt{x} \mapsto \underline{15}, \mathtt{y} \mapsto \underline{10}, \mathtt{z} \mapsto \underline{10}, \mathtt{w} \mapsto \underline{100}\}.\Omega$

   (b) $\sigma = \{\mathtt{x} \mapsto \underline{10}, \mathtt{y} \mapsto \underline{2}, \mathtt{z} \mapsto \underline{10}, \mathtt{w} \mapsto \underline{100}\}.\Omega$

4. Valutare il blocco

   ```
   {
   z=1;
   while (x != 0) {z=z*x; x=x-1;}
   }
   ```

   a partire da uno stato σ nei seguenti casi:

   (a) $\sigma(\mathtt{x}) = \underline{0}$ e $\sigma(\mathtt{z}) \neq \perp$

   (b) $\sigma(\mathtt{x}) = \underline{4}$ e $\sigma(\mathtt{z}) \neq \perp$

5. Valutare il blocco

   ```
   {
   z = 0;
   while (y<=x) {z = z+1; x = x-y;}
   }
   ```

   a partire da uno stato σ nei seguenti casi:

   (a) $\sigma(\mathtt{x}) = \underline{17}$, $\sigma(\mathtt{y}) = \underline{5}$ e $\sigma(\mathtt{z}) \neq \perp$

   (b) $\sigma(\mathtt{x}) = \underline{5}$, $\sigma(\mathtt{y}) = \underline{8}$ e $\sigma(\mathtt{z}) \neq \perp$

5.3 Dimostrazioni di equivalenza attraverso la semantica operazionale

La semantica operazionale può essere utilizzata per dimostrare proprietà interessanti, come ad esempio l'equivalenza tra espressioni e/o comandi. Informalmente, diciamo che due comandi (espressioni) sono equivalenti se hanno lo stesso comportamento in qualunque contesto, ovvero a partire dallo stesso stato iniziale. Più formalmente ciò può essere espresso come segue.

Definizione 5.4 Due comandi $\mathtt{C}$ e $\mathtt{C}'$ sono *equivalenti* se, per ogni stato σ, vale una delle seguenti condizioni:

a) né $\langle \mathtt{C}, \sigma \rangle$ né $\langle \mathtt{C}', \sigma \rangle$ portano in una configurazione terminale;

b) $\langle \mathtt{C}, \sigma \rangle$ e $\langle \mathtt{C}', \sigma \rangle$ portano nella stessa configurazione terminale. □

Consideriamo ad esempio i due comandi seguenti:

$\mathtt{C}_1$: `if (true) x=5; else x=6;`
$\mathtt{C}_2$: `x=5;`

e dimostriamone l'equivalenza. Sia dunque σ un generico stato e valutiamo $\langle \mathtt{C}_1, \sigma \rangle$ e $\langle \mathtt{C}_2, \sigma \rangle$.

$$\begin{array}{ll} & \langle \mathtt{C}_1, \sigma \rangle \\ \rightarrow_{com} & \{ \ (com_{if-tt}), \ \mathcal{E}[\![\mathtt{true}]\!]\sigma = tt, \\ & \quad (com_{=}) : \mathcal{E}[\![5]\!]\sigma = \underline{5}, \ \langle \mathtt{x} = 5;, \sigma \rangle \rightarrow_{com} \sigma[^{\underline{5}}/_{\mathtt{x}}] \ \} \\ & \sigma[^{\underline{5}}/_{\mathtt{x}}] \end{array}$$

$$\begin{array}{ll} & \langle \mathtt{C}_2, \sigma \rangle \\ \rightarrow_{com} & \{ \ (com_{=}) : \mathcal{E}[\![5]\!]\sigma = \underline{5} \ \} \\ & \sigma[^{\underline{5}}/_{\mathtt{x}}] \end{array}$$

In questo caso, nelle due derivazioni non abbiamo fatto alcuna ipotesi sullo stato iniziale σ ed abbiamo ottenuto lo stesso stato finale: i due comandi, quindi, a partire da un generico stato portano nella stessa configurazione terminale.

Come ulteriore esempio, consideriamo i due comandi seguenti:

$\mathtt{C}_1$: `if (x>2) y=1; else y=2;`
$\mathtt{C}_2$: `if (x<=2) y=2; else y=1;`

La dimostrazione di equivalenza avviene ragionando *per casi*: consideriamo dapprima uno stato σ tale che $\mathcal{E}[\![x > 2]\!]\sigma = tt$. Abbiamo allora le due seguenti derivazioni:

$$\begin{array}{ll} & \langle \mathtt{C}_1, \sigma \rangle \\ \rightarrow_{com} & \{ \ (com_{if-tt}), \ \mathcal{E}[\![\mathtt{x} > 2]\!]\sigma = tt, \\ & \quad (com_{=}) : \mathcal{E}[\![1]\!]\sigma = \underline{1}, \ \langle \mathtt{y} = 1;, \sigma \rangle \rightarrow_{com} \sigma[^{\underline{1}}/_{\mathtt{y}}] \ \} \\ & \sigma[^{\underline{1}}/_{\mathtt{y}}] \end{array}$$

$$\begin{array}{ll} & \langle \mathtt{C}_2, \sigma \rangle \\ \rightarrow_{com} & \{ \ (com_{if-ff}), \ \mathcal{E}[\![\mathtt{x}<=2]\!]\sigma = f\!f, \\ & \quad (com_{=}) : \mathcal{E}[\![1]\!]\sigma = \underline{1}, \ \langle \mathtt{y} = 1;, \sigma \rangle \rightarrow_{com} \sigma[^{\underline{1}}/_{\mathtt{y}}] \ \} \\ & \sigma[^{\underline{1}}/_{\mathtt{y}}] \end{array}$$

Abbiamo così concluso che i due comandi portano nello stesso stato finale, a partire da un generico stato iniziale in cui $\mathcal{E}[\![\mathtt{x} > 2]\!]\sigma = tt$. Ciò non conclude tuttavia la dimostrazione, che deve considerare anche tutti gli altri stati, ovvero quelli in cui $\mathcal{E}[\![\mathtt{x} > 2]\!]\sigma = f\!f$. Sia allora σ un generico stato in cui $\mathcal{E}[\![\mathtt{x} > 2]\!]\sigma = f\!f$. Abbiamo:

$$\begin{array}{ll} & \langle \mathtt{C}_1, \sigma\rangle \\ \to_{com} & \{\ (com_{if-f\!f}),\ \mathcal{E}[\![\mathtt{x} > 2]\!]\sigma = f\!f, \\ & \quad (com_{=}) : \mathcal{E}[\![2]\!]\sigma = \underline{2},\ \langle \mathtt{y} = 2;, \sigma\rangle \to_{com} \sigma[^{\underline{2}}/_{\mathtt{y}}]\ \} \\ & \sigma[^{\underline{2}}/_{\mathtt{y}}] \end{array}$$

$$\begin{array}{ll} & \langle \mathtt{C}_2, \sigma\rangle \\ \to_{com} & \{\ (com_{if-tt}),\ \mathcal{E}[\![\mathtt{x<=2}]\!]\sigma = tt, \\ & \quad (com_{=}) : \mathcal{E}[\![2]\!]\sigma = \underline{2},\ \langle \mathtt{y} = 2;, \sigma\rangle \to_{com} \sigma[^{\underline{2}}/_{\mathtt{y}}]\ \} \\ & \sigma[^{\underline{2}}/_{\mathtt{y}}] \end{array}$$

Un altro tipo di equivalenza, più debole, è quella che si limita a considerare i due costrutti (comandi o espressioni) in un particolare stato. Consideriamo ad esempio le due espressioni

$\mathtt{E}_1 =$ `(x+8) / y` $\qquad\qquad$ $\mathtt{E}_2 =$ `((x*y)+y) / 2`

e dimostriamone l'equivalenza rispetto ad uno stato σ tale che $\sigma(\mathtt{x}) = \underline{5}$ e $\sigma(\mathtt{y}) = \underline{2}$. Nell'ultimo esempio del paragrafo 4.1 abbiamo visto che:

$\langle \mathtt{E}_1, \sigma\rangle \to_{exp} \underline{6}$ $\qquad\qquad$ (i)

Lasciamo per esercizio al lettore la derivazione

$\langle \mathtt{E}_2, \sigma\rangle \to_{exp} \underline{6}$

che, insieme con (i), fornisce l'equivalenza di $\mathtt{E}_1$ ed $\mathtt{E}_2$ in σ. Si noti che gli stati σ tali che $\sigma(\mathtt{x}) = \underline{5}$ e $\sigma(\mathtt{y}) = \underline{2}$ sono infiniti, ma questo non basta per dimostrare l'equivalenza di $\mathtt{E}_1$ ed $\mathtt{E}_2$ in tutti i possibili stati. Infatti, in uno stato σ' tale che $\sigma'(\mathtt{x}) = \underline{1}$ e $\sigma'(\mathtt{y}) = \underline{1}$ le due espressioni danno un risultato diverso (verificarlo per esercizio).

Vediamo un ultimo esempio, in cui è ancora necessario un ragionamento *per casi*. Consideriamo i due comandi:

$\mathtt{C}_1$: `if (E) x=true; else x=false;`
$\mathtt{C}_2$: `x=E;`

dove `E` è una qualsiasi espressione booleana. Consideriamo un generico stato σ e scindiamo la dimostrazione in tre casi.

(i) $\mathcal{E}[\![\mathtt{E}]\!]\sigma$ non è definita. In questo caso entrambi i comandi non portano in alcuna configurazione terminale.

(ii) $\mathcal{E}[\![\mathtt{E}]\!]\sigma = tt$. In questo caso abbiamo le due derivazioni:

$$\begin{array}{ll} & \langle \mathtt{C}_1, \sigma\rangle \\ \to_{com} & \{\ (com_{if-tt}),\ \mathcal{E}[\![\mathtt{E}]\!]\sigma = tt, \\ & \quad (com_{=}):\ \mathcal{E}[\![\mathtt{true}]\!]\sigma = tt,\ \langle \mathtt{x} = \mathtt{true};, \sigma\rangle \to_{com} \sigma[^{tt}/_{\mathtt{x}}]\ \} \\ & \sigma[^{tt}/_{\mathtt{x}}] \end{array}$$

$$\begin{array}{ll} & \langle \mathtt{C}_2, \sigma\rangle \\ \to_{com} & \{\ (com_{=}):\ \mathcal{E}[\![\mathtt{E}]\!]\sigma = tt,\ \langle \mathtt{x} = \mathtt{E};, \sigma\rangle \to_{com} \sigma[^{tt}/_{\mathtt{x}}]\ \} \\ & \sigma[^{tt}/_{\mathtt{x}}] \end{array}$$

(iii) $\mathcal{E}[\![\mathtt{E}]\!]\sigma = \mathit{ff}$. In questo caso abbiamo le due derivazioni:

$$
\begin{array}{ll}
 & \langle \mathtt{C}_1, \sigma \rangle \\
\rightarrow_{com} & \quad \{\ (com_{if-\mathit{ff}}),\ \mathcal{E}[\![\mathtt{E}]\!]\sigma = \mathit{ff}, \\
 & \qquad (com_{=}){:}\ \mathcal{E}[\![\mathtt{false}]\!]\sigma = \mathit{ff},\ \langle \mathtt{x} = \mathtt{false};, \sigma \rangle \rightarrow_{com} \sigma[^{\mathit{ff}}/_{\mathtt{x}}]\ \} \\
 & \sigma[^{\mathit{ff}}/_{\mathtt{x}}]
\end{array}
$$

$$
\begin{array}{ll}
 & \langle \mathtt{C}_2, \sigma \rangle \\
\rightarrow_{com} & \quad \{\ (com_{=}){:}\ \mathcal{E}[\![\mathtt{E}]\!]\sigma = \mathit{ff},\ \langle \mathtt{x} = \mathtt{E};, \sigma \rangle \rightarrow_{com} \sigma[^{\mathit{ff}}/_{\mathtt{x}}]\ \} \\
 & \\
 & \sigma[^{\mathit{ff}}/_{\mathtt{x}}]
\end{array}
$$

Poiché i casi (i), (ii) e (iii) esauriscono tutti i possibili, concludiamo che i due comandi sono equivalenti.

Esercizi

1. Verificare l'equivalenza dei due comandi:

   $\mathtt{C}_1$: `if (true) x=5; else x=3;`
   $\mathtt{C}_2$: `if (false) x=3; else x=5;`

2. Verificare l'equivalenza dei due comandi:

   $\mathtt{C}_1$: `if (x==y) x=x-y; else x=y-y;`
   $\mathtt{C}_2$: `x=0;`

3. Dati i due comandi:

   $\mathtt{C}_1$: `while (x>2) x=x-1;`
   $\mathtt{C}_2$: `x=x-1; while (x>2) x=x-1;`

   determinare uno stato in cui sono equivalenti ed uno in cui non lo sono.

4. Siano $\mathtt{C}_1$, $\mathtt{C}_2$ e $\mathtt{C}_3$ comandi arbitrari e $\mathtt{C}$, $\mathtt{C}'$ i comandi:

   $\mathtt{C}$: $\mathtt{C}_2\ \mathtt{C}_3$
   $\mathtt{C}'$: $\mathtt{C}_1\ \mathtt{C}_2$

   Dimostrare l'equivalenza di $\mathtt{C}'\ \mathtt{C}_3$ e $\mathtt{C}_1\ \mathtt{C}$

5. Siano $\mathtt{C}_1$, $\mathtt{C}_2$ e $\mathtt{C}_3$ comandi arbitrari. Verificare l'equivalenza dei due comandi:

   $\mathtt{C}$: $\mathtt{if\ (E)}\ \{\mathtt{C}_1\ \mathtt{C}_3\}\ \mathtt{else}\ \{\mathtt{C}_2\ \mathtt{C}_3\}$
   $\mathtt{C}'$: $\mathtt{if\ (E)}\ \mathtt{C}_1\ \mathtt{else}\ \mathtt{C}_2\ \mathtt{C}_3$

6. Siano $\mathtt{C}_1$, $\mathtt{C}_2$ e $\mathtt{C}_3$ comandi arbitrari. Dare una condizione sufficiente affinché i comandi

   $\mathtt{C}$: $\mathtt{C}_1\ \mathtt{if\ (E)}\ \mathtt{C}_2\ \mathtt{else}\ \mathtt{C}_3$
   $\mathtt{C}'$: $\mathtt{if\ (E)}\ \{\mathtt{C}_1\ \mathtt{C}_2\}\ \mathtt{else}\ \{\mathtt{C}_1\ \mathtt{C}_3\}$

   siano equivalenti nello stato σ.

5.4 Le dichiarazioni e la loro semantica

Nelle sezioni precedenti abbiamo introdotto i concetti di frame e stato per modellare associazioni tra nomi e valori. Il costrutto linguistico che consente di introdurre i nomi utilizzati in un frammento di programma è quello delle *dichiarazioni*. In questo paragrafo ci occupiamo delle dichiarazioni di *variabili*, che corrispondono ad associazioni modificabili nello stato: la modifica di tali associazioni avviene, come visto nei paragrafi precedenti, mediante il comando di assegnamento. Non ci occuperemo invece delle dichiarazioni di *costanti*, corrispondenti ad associazioni non modificabili. Inoltre, ogni nome viene dichiarato con il suo *tipo*, vale a dire l'insieme dei valori che può essere associato a quel nome. Nel caso del nostro semplice linguaggio ci limitiamo, per il momento, ai tipi `int` e `boolean`, che corrispondono rispettivamente all'insieme dei numeri interi e all'insieme dei valori di verità.

La sintassi di una dichiarazione è data dalle seguenti produzioni:

```
Decl ::=
    int Ide; |
    boolean Ide; |
    int Ide = Exp; |
    boolean Ide = Exp;
```

Ogni identificatore di variabile, prima di essere utilizzato in una espressione o in un comando, deve essere dichiarato. Una dichiarazione può essere fatta in un qualunque punto di una sequenza all'interno di un blocco. Dal punto di vista sintattico, ciò comporta la seguente modifica delle produzioni date nel paragrafo 5.2.

```
Block ::= {Stat_List}
Stat_List ::= Stat | Stat Stat_List
Stat ::= Com | Decl
```

L'utilizzo di un nome all'interno di un assegnamento o di una espressione è lecito se tale nome è stato prima dichiarato.
Vediamo subito alcuni esempi di blocchi.

```
{
    int x;
    x = 10;
    int y;
    y = x + 5;
}
```

Come si vede l'uso di ciascuna variabile all'interno di un comando di assegnamento o di una espressione, segue la sua dichiarazione. Non è invece lecita una sequenza del tipo

```
{
    int x;
    x = 10;
    y = x + 5;
    int y;
}
```

in cui la dichiarazione della variabile `y` segue il suo utilizzo. Vediamo cosa succede nel caso di blocchi annidati.

```
{
    int x;
    x = 10;
        {
            int y;
            y = x + 5;
        }
    ...
}
```

Come si vede nel blocco più interno viene dichiarato l'identificatore `y` e viene usato anche l'identificatore `x`, quest'ultimo dichiarato nel blocco più esterno. Questa situazione è lecita, poiché un nome dichiarato in un blocco è *visibile* all'interno dei blocchi in esso annidati[9]. All'uscita di un blocco, tutti i nomi in esso dichiarati cessano di esistere: questi concetti intuitivi che poco si prestano ad essere spiegati informalmente, troveranno una sistematizzazione chiara e concisa non appena daremo le nuove regole semantiche per le dichiarazioni e per i blocchi.

Nel seguito, dato un blocco, chiameremo variabili *locali* al blocco tutte le variabili dichiarate all'interno del blocco medesimo e chiameremo variabili *globali* al blocco tutte le variabili visibili nel blocco ma non dichiarate in esso.

Prima di formalizzare i concetti sopra esposti, vediamo qualche altro esempio di annidamento.

```
{
    int x;
    x = 10;
        {
            int y;
            y = x + 5;
        }
    x = y+1;
}
```

L'assegnamento `x=y+1` è errato, in quanto il nome `y` non è locale né globale al blocco in cui si trova l'assegnamento stesso. Si noti che, nel blocco più interno, una variabile di nome `y` è stata dichiarata, ma ha cessato di esistere con il termine del blocco annidato.

```
{
    int x;
    x = 10;
        {
            int y;
            y = z + 5;
        }
    int z;
    z = 20;
}
```

L'assegnamento `y = z + 5` è errato in quanto il nome `z` è dichiarato nel blocco più esterno, ma in una posizione successiva al blocco più interno in cui viene utilizzato.

Veniamo alla formalizzazione delle dichiarazioni mediante la semantica operazionale. Introduciamo dapprima un sistema di transizioni per le dichiarazioni, $\mathtt{S}_{dec}$, il cui scopo è di aggiungere nel frame corrente una associazione per ogni nuova variabile dichiarata. Le configurazioni non terminali di $\mathtt{S}_{dec}$ sono del tipo $\langle \mathtt{D}, \sigma \rangle$, dove `D` è un dichiarazione e σ uno stato, mentre le configurazioni terminali sono semplicemente stati.

Osserviamo innanzitutto che una dichiarazione di variabile fissa il tipo dei valori che la variabile potrà assumere in seguito: questo è il motivo della presenza del tipo nella dichiarazione. Nonostante sia necessaria, noi non utilizzeremo in questa definizione semantica l'informazione relativa al tipo dei valori[10].

Inoltre, per ciascun tipo elementare `T` (nel nostro caso `int` o `boolean`) una variabile può essere dichiarata senza inizializzarne il valore (ad esempio `int x;`) oppure inizializzandone il valore a quello di una espressione (ad esempio `boolean vero=true;`). Nel primo caso, il valore associato alla variabile è imprecisato: per evidenziare il fatto che un tale valore non fa parte della definizione del linguaggio, utilizziamo il valore *simbolico* ϖ.

[9] Purché non vi siano dichiarazioni annidate che utilizzano lo stesso nome (si veda l'esempio alla fine di questa sezione).

[10] La nostra definizione semantica consta di due parti, *semantica statica* e *semantica dinamica*: la parte statica che si occupa della coerenza dei tipi verrà discussa in seguito.

$$\frac{\sigma = \varphi.\sigma'' \qquad \sigma' = \varphi[^{\varpi}/_{\mathtt{x}}].\sigma''}{\langle \mathtt{T\ x;}, \sigma\rangle \rightarrow_{dec} \sigma'} \quad (dec_{var-1})$$

$$\frac{\sigma = \varphi.\sigma'' \qquad \langle \mathtt{E}, \sigma\rangle \rightarrow_{exp} \mathtt{v} \qquad \sigma' = \varphi[^{\mathtt{v}}/_{\mathtt{x}}].\sigma''}{\langle \mathtt{T\ x = E;}, \sigma\rangle \rightarrow_{dec} \sigma'} \quad (dec_{var-2})$$

Le dichiarazioni, come visto negli esempi e come formalizzato nella sintassi modificata data in precedenza, vengono introdotte nei blocchi. Dunque, la semantica data per questi ultimi con le regole (com_{Block}) e ($com_{Stat-List}$), va modificata opportunamente. Intuitivamente, ogni blocco corrisponde alla introduzione di un nuovo frame nello stato: tale frame è destinato a contenere le associazioni per le variabili dichiarate in quel blocco e cessa di esistere una volta terminata l'esecuzione del blocco. Le liste di dichiarazioni e comandi (elementi di `Stat_list`) vengono trattate attraverso un opportuno sottosistema, le cui transizioni sono date di seguito.

$$\frac{\langle \mathtt{Slist}, \omega.\sigma\rangle \rightarrow_{slist} \varphi.\sigma'}{\langle \{\mathtt{Slist}\}, \sigma\rangle \rightarrow_{com} \sigma'} \quad (com_{Block})$$

$$\frac{\mathtt{S} \in \mathtt{Decl} \qquad \langle \mathtt{S}, \sigma\rangle \rightarrow_{dec} \sigma'}{\langle \mathtt{S}, \sigma\rangle \rightarrow_{slist} \sigma'} \quad (slist_{dec})$$

$$\frac{\mathtt{S} \in \mathtt{Com} \qquad \langle \mathtt{S}, \sigma\rangle \rightarrow_{com} \sigma'}{\langle \mathtt{S}, \sigma\rangle \rightarrow_{slist} \sigma'} \quad (slist_{com})$$

$$\frac{\langle \mathtt{S}, \sigma\rangle \rightarrow_{slist} \sigma'' \qquad \langle \mathtt{Slist}, \sigma''\rangle \rightarrow_{slist} \sigma'}{\langle \mathtt{S\ \ Slist}, \sigma\rangle \rightarrow_{slist} \sigma'} \quad (slist_{list})$$

La regola (com_{Block}) formalizza il fatto che un blocco viene eseguito in uno stato nel quale è stato impilato un nuovo frame (inizialmente vuoto): tale frame è quello destinato a contenere le dichiarazioni locali al blocco. La conclusione della regola, inoltre, formalizza il fatto che, nello stato risultante dall'esecuzione del blocco, i nomi locali cessano di esistere mentre permangono le eventuali modifiche apportate a nomi globali al blocco.

Vediamo un esempio di valutazione di un blocco, con l'ausilio della rappresentazione grafica dello stato e dei frame in gioco.

```
{
    int x;
    x = 5;
      {                 (1)
       int y;
       y = x + 5;       (2)
       x = x + y;       (3)
      }                 (4)
    x = 2*x;            (5)
}
```

Per comodità, abbiamo numerato alcune linee, per poterle riferire nel seguito. Supponiamo che tale blocco venga eseguito a partire dallo stato vuoto, Ω. Lo stato al punto (1), al momento cioè dell'ingresso nel blocco annidato, è il seguente:

x	5

Dopo l'esecuzione dell'assegnamento (2), lo stato si presenta come segue:

y	10

x	5

Si noti come l'ingresso nel blocco annidato abbia causato la creazione di un nuovo frame, destinato a contenere le variabili dichiarate, appunto, in tale blocco (nel caso specifico `y`).

Dopo l'esecuzione dell'assegnamento (3) abbiamo il seguente stato:

y	10

x	15

All'uscita del blocco annidato, punto (4), il frame precedentemente creato cessa di esistere

x	15

ed è in quest'ultimo stato che viene eseguito l'assegnamento (5), a seguito del quale abbiamo lo stato conclusivo

x	30

Vediamo infine un esempio che mostra come la semantica data consenta anche di trattare il caso di blocchi annidati in cui uno stesso identificatore viene ridefinito (purché non nello stesso blocco). Questa possibilità, pur se nella tradizione dei linguaggi imperativi quali Algol, Pascal e C, non viene contemplata in Java, dove la dichiarazione di una variabile `x` in un blocco annidato dentro un blocco più esterno in cui `x` è già stata dichiarata viene considerato come errore statico.

```
{
    int x = 10;
    int y = 20;
      {                 (1)
       int x = 100;     (2)
       y = y + x;       (3)
      }
    y = y + x;          (4)
}
```

Supponiamo che tale blocco venga eseguito a partire dallo stato vuoto, Ω. Lo stato al punto (1), al momento cioè dell'ingresso nel blocco annidato, è il seguente:

x	10
y	20

Dopo l'esecuzione della dichiarazione (2), lo stato si presenta come segue:

x	100

x	10
y	20

Si noti che la nuova variabile `x` fa parte del frame più recente, corrispondente al blocco annidato. In questo stato viene eseguito l'assegnamento (3), il cui effetto è rappresentato dal nuovo stato:

x	100

x	10
y	120

Si noti che, nell'espressione `y+x`, il riferimento ad `x` è un riferimento alla dichiarazione più recente per `x`. Dopo l'uscita dal blocco annidato, il frame corrispondente cessa di esistere e dunque il riferimento ad `x` in (4) è ora un riferimento alla dichiarazione di `x` del blocco più esterno. L'effetto dell'assegnamento (4) è dunque rappresentato dal seguente stato:

x	$\underline{10}$
y	$\underline{130}$

Esercizi

1. Valutare le seguenti liste di statements mediante il sottosistema $\rightarrow_{slist}$ a partire dallo stato $\sigma = \omega.\Omega$:

   (a) `int a; a=5; int b; b=3; int x=10;`

   (b) `int x; int y; int z;`

2. Valutare le seguenti liste di statements mediante il sottosistema $\rightarrow_{slist}$ a partire dallo stato $\sigma = \omega.\Omega$, mostrando graficamente lo stato dopo l'esecuzione di ciascun elemento della lista:

   (a) `int x=10; int y=20; {int y=30; x = x+y; y=2*y; y=2*y; x=x+y;}`

   (b) `int x=10; int y=20; {int x=40; int y=30; x = x+y; y=2*y;} y=2*y; x=x+y;`

   (c) `int x=10; int y=20; {int x=40; x = x+y; y=x; int y=30; x = x+y;} y=2*y; x=x+y;`

3. Valutare i seguenti blocchi a partire dallo stato $\sigma = \{\mathtt{x} \mapsto \underline{10}, \mathtt{y} \mapsto \underline{10}\}.\Omega$:

   (a) `{int x=50; int z=50; y=x+2*z; x=x+y;}`

   (b) `{int z=50; int y=50; {int x=100; x = x+y; z=x;} x=x+z; y=x+1;}`

4. In riferimento ad una nozione di equivalenza per liste di statements analoga a quella introdotta nella definizione 5.4, dimostrare l'equivalenza delle liste $\mathtt{B}_1$ e $\mathtt{B}_2$ nei due casi seguenti:

   (a)
   ```
   B1 :  int a=5; int x;
   B2 :  int x; int a=5;
   ```

   (b)
   ```
   B1 :  int a; int b=10; a=5;
   B2 :  int  a=5; int b=2*a;
   ```

5.5 Oggetti

Un programma Java consiste di un insieme di *oggetti* che possono eseguire delle azioni, dette *metodi*. Mediante le *chiamate* ai metodi, è possibile chiedere ad un oggetto di eseguire le azioni di cui è capace. Tutti gli oggetti dello stesso tipo fanno parte della medesima *classe*, che può dunque essere vista come una categoria di oggetti. Nel linguaggio che prendiamo in considerazione (un frammento di Java) la definizione di una classe prevede:

- le dichiarazioni delle cosiddette *variabili istanza*, che permettono di associare ad ogni oggetto della classe un proprio *stato*;
- le definizioni dei *metodi*, che descrivono le azioni che ogni oggetto della classe può intraprendere.

Una volta definita una classe (si veda il paragrafo 5.6), si possono dichiarare variabili i cui valori sono oggetti di quella classe. In realtà, in Java tutti i tipi di dato non elementari, a partire dal tipo delle stringhe (sequenze di caratteri), sono trattati con il meccanismo delle classi. Così, ad esempio, se `Foo` è il nome di una classe, una dichiarazione del tipo

```
Foo myobj;
```

definisce una nuova variabile il cui scopo è quello di riferire un nuovo oggetto della classe `Foo`. Una dichiarazione come la precedente, ha lo stesso effetto di una dichiarazione di variabile come quelle viste nella sezione 5.4, ovvero di introdurre nello stato corrente una nuova associazione per il nome `myobj`. Non trattandosi di un tipo elementare, dobbiamo ancora chiarire quale sia il valore associato a tale nome. Osserviamo che una dichiarazione come questa, tuttavia, non comporta la creazione vera e propria dell'oggetto associato alla variabile `myobj`: tale creazione avviene invece tramite l'uso della primitiva `new`. Intuitivamente, l'esecuzione di

```
new Foo
```

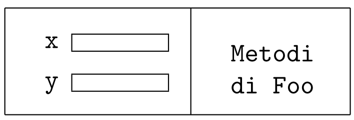
ha l'effetto di creare la vera e propria rappresentazione di un nuovo oggetto della classe `Foo`, con il suo proprio stato (ovvero con le sue proprie variabili istanza) ed i suoi metodi. Molto spesso l'inizializzazione di un oggetto avviene contestualmente alla dichiarazione di una variabile oggetto, come in

$$\texttt{Foo myobj} = \texttt{new Foo};\qquad\qquad (*)$$

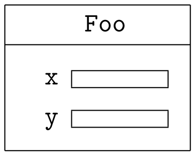
Nella sezione 5.7 daremo la semantica formale di tutti questi concetti: limitiamoci per ora ad una rappresentazione grafica di ciò che accade al momento di una dichiarazione ed inizializzazione come $(*)$. Per semplicità, consideriamo per il momento che la classe `Foo` sia già stata definita e che essa preveda la presenza di due variabili istanza, `x` e `y`, entrambe di tipo `int`. La definizione di `Foo` conterrà in generale anche dei metodi, che per il momento prendiamo in considerazione solo marginalmente. La dichiarazione $(*)$ ha allora i seguenti effetti.

1. Si crea la rappresentazione del nuovo oggetto, ovvero una struttura che:
   - contenga lo stato proprio dell'oggetto, costituito dalle sue variabili istanza `x` e `y`;
   - consenta di accedere quando necessario ai metodi definiti nella classe di appartenenza del nuovo oggetto.

   Una tale struttura può essere del tipo rappresentato di seguito:

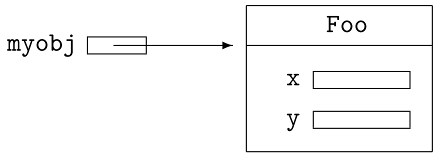


   Si noti che lo stato non è altro che un frame, secondo la terminologia introdotta nella sezione 4.2. Un'altra possibile rappresentazione dell'oggetto è quella descritta nella seguente figura:



Anche in questo caso abbiamo un frame che rappresenta lo stato dell'oggetto; non abbiamo invece una copia dei metodi della classe di appartenenza, quanto piuttosto il nome della classe stessa. Ciò presuppone che le definizioni delle classi diano luogo ad una rappresentazione delle stesse che preveda la possibilità di reperire un metodo della classe a partire dal nome della classe stessa. Questa scelta non può chiaramente essere fatta anche per le variabili istanza, in quanto esse devono essere distinte per ogni oggetto della classe. I metodi, essendo entità che una volta definiti non possono essere modificati, possono invece essere mantenuti in una struttura comune a tutti gli oggetti della medesima classe. La scelta che adottiamo in queste note è la seconda, più vicina ad una reale implementazione del linguaggio Java. Nella prossima sezione vedremo in che modo le definizioni delle classi danno luogo alla struttura cui abbiamo appena accennato.

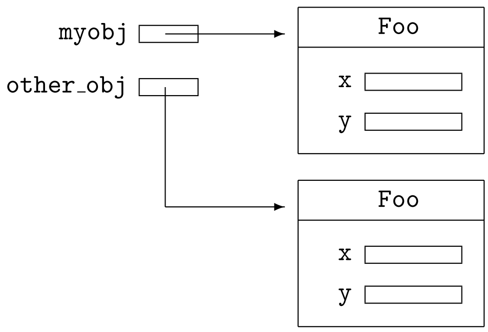
2. Nello stato corrente, si associa la rappresentazione appena costruita alla variabile `myobj`: ciò avviene associando a quest'ultima un *riferimento* all'oggetto appena creato, rappresentato in figura mediante una freccia (in seguito, definiremo formalmente che cosa sono i riferimenti).



Come ulteriore esempio, immaginiamo, a partire dalla situazione precedente, una ulteriore dichiarazione di variabile

`Foo other_obj = new Foo;`

La situazione conseguente alla esecuzione di tale dichiarazione sarà quella descritta nella figura:



in cui lo stato contiene anche una associazione per `other_obj`, la cui rappresentazione è del tutto analoga a quella di `myobj`, seppure da essa distinta.

Le rappresentazioni degli oggetti, ovvero le strutture come quelle associate alle variabili `myobj` e `other_obj` dell'esempio precedente, vengono mantenute in una opportuna struttura semantica, denominata *heap*: dunque ogni invocazione della primitiva `new` richiede la creazione di una nuova struttura e la sua memorizzazione nello *heap*.

Da un punto di vista formale, dobbiamo definire con precisione:

- le rappresentazioni degli oggetti: queste possono essere modellate mediante coppie del tipo

  $$(\mathtt{C}, \varphi)$$

  dove φ è un frame e `C` è il nome della classe dell'oggetto. Ad esempio, la rappresentazione inizialmente associata alla variabile `myobj` dell'esempio precedente è una coppia

  $$(\mathtt{Foo}, \{\mathtt{x} \mapsto \varpi, \mathtt{y} \mapsto \varpi\})$$

  (assumendo che i valori delle variabili istanza siano indefiniti).

- lo *heap*, che può essere modellato mediante una tabella (una funzione a dominio finito) che associa rappresentazioni di oggetti (ovvero coppie del tipo $(\mathtt{c}, \varphi)$) a riferimenti ad oggetti (rappresentati da frecce nelle figure precedenti). I riferimenti sono presi da un insieme opportuno, che chiameremo `Loc`. Dunque uno *heap* è una funzione del tipo

  $$\mathtt{Loc} \rightarrow \mathtt{Ide} \times \Phi$$

  In seguito, denoteremo con `Heap` l'insieme dei possibili heap ed utilizzeremo $\zeta, \zeta', \ldots$ per denotare elementi di `Heap`.

5.6 Classi

Nella sezione precedente abbiamo introdotto parte delle strutture semantiche necessarie per un trattamento formale degli oggetti. Questi ultimi sono elementi di classi, definite all'interno di un programma. Come abbiamo già accennato, nel nostro linguaggio la definizione di una classe comporta quella delle variabili istanza e dei metodi che ogni oggetto di tale classe possiede una volta creato. Senza per il momento addentrarci nei dettagli sintattici, consideriamo un esempio di definizione di classe.

```
class Foo
{
  int x;
  int y;
  <metodi della classe>
}
```

Dal punto di vista semantico, l'effetto di una dichiarazione come la precedente è di mantenere in una opportuna struttura semantica gli ingredienti che saranno necessari al momento della creazione di un nuovo oggetto della classe appena definita. Tali ingredienti, associati al nome della classe, sono un frame che rappresenta le variabili istanza e una opportuna rappresentazione ρ_m dei metodi: vedremo tale rappresentazione nella sezione 5.8, quando discuteremo della semantica delle dichiarazioni ed invocazioni dei metodi. Per il momento, ci accontentiamo di denotare con `Method_Env` l'insieme di tali rappresentazioni (dunque $\rho_m \in \mathtt{Method\_Env}$).

La struttura semantica di cui abbiamo bisogno è allora una tabella del tipo

`classe_1`	φ_1	ρ_m^1
`classe_2`	φ_2	ρ_m^2
`....`	`....`	
`classe_n`	φ_n	ρ_m^n

Tale tabella può essere modellata mediante una funzione del tipo

$$\mathtt{Ide} \rightarrow \Phi \ \times \ \mathtt{Method\_Env}.$$

Indicheremo con `Class_Env` lo spazio delle funzioni del tipo appena definito ed utilizzeremo $\rho_c, \rho'_c, \ldots$ per indicare elementi di `Class_Env`.

Si noti che, al momento della creazione di un nuovo oggetto mediante una dichiarazione del tipo

$$\texttt{Foo myobj} = \texttt{new Foo};$$

la coppia $(\texttt{Foo}, \varphi)$ che viene aggiunta allo heap come rappresentazione di `myobj` (si veda il paragrafo 5.5), viene costruita proprio a partire da quanto associato all'identificatore `Foo` in ρ_c: in particolare, il frame φ non è altro che la prima componente della coppia $\rho_c(\texttt{Foo})$.

5.7 Semantica dei programmi nel linguaggio semplificato

Siamo ora in grado di procedere con la formalizzazione sia sintattica che semantica dei programmi nel nostro linguaggio semplificato. Un programma è costituito da una sequenza di dichiarazioni di classi e da un blocco principale. Nel nostro linguaggio, la funzione del blocco è la stessa del metodo `main` nei programmi Java. Inoltre, per semplicità, assumiamo che tutte le variabili istanza delle classi siano dichiarate come `public`. In realtà nella buona pratica di programmazione è bene che le variabili istanza degli oggetti siano dichiarate come `private` e siano accessibili solo attraverso i metodi della classe. Si noti comunque che il controllo sull'uso privato delle variabili può essere demandato ad un controllo statico senza coinvolgere la semantica dinamica che stiamo trattando.

La sintassi del nostro frammento di Java viene estesa come segue:

```
Prog::= prog { Class_decl_list {Stat_List} }

Class_decl_list ::= Class_decl |
                    Class_decl  Class_decl_list

Class_decl ::=  class Ide { Ivar_decl_list  Method_decl_list }

Ivar_decl_list ::= Ivar_decl |
                   Ivar_decl Ivar_decl_list

Ivar_decl ::= public Type Ide ;

Exp ::=  Ide.Ide

Com ::=  Ide.Ide = Exp; | Ide = new Ide;

Dec ::= Ide Ide; | Ide Ide = new Ide;

Method_decl_list ::= ....
```

La categoria sintattica `Method_decl_list`, che consente di definire i metodi degli oggetti, verrà introdotta nella sezione 5.8. Le nuove produzioni per la categoria sintattica `Com` consentono di esprimere, rispettivamente:

- la modifica di una variabile istanza di un oggetto: in `x.y = E;`, `x` deve fare riferimento ad un oggetto e `y` deve essere l'identificatore di una variabile istanza dichiarata nella classe di appartenenza di tale oggetto (ad esempio `myobj.x = 5;`);
- l'assegnamento di un nuovo oggetto ad una variabile oggetto (ad esempio `myobj = new Foo;`)

La nuova produzione per la categoria sintattica `Exp` consente di rappresentare espressioni che dipendono dal valore di variabili istanza di oggetti. Infine, le nuove produzioni per la categoria sintattica `Dec` consentono di dichiarare variabili oggetto, eventualmente inizializzandole mediante l'operazione `new`.

Esempio 5.5 Consideriamo un semplice programma:

```
prog {
    class Foo
        {
            public int x;
            public int y;
        }
    class Fie
        {
            public int x;
        }

    {
        Foo o1 = new Foo;
        Fie o2 = new Fie;
        o1.x = 5;
        o2.x = 10;
        o1.y = o1.x + o2.x;
    }
    }
```

Il lettore attento avrà notato che il programma non è sintatticamente corretto rispetto alla grammatica data, poiché le classi `Foo` e `Fie` hanno una sequenza vuota di dichiarazioni di metodi. Lasciamo al lettore come esercizio la modifica della grammatica che consenta di definire classi senza variabili istanza o senza metodi. ∎

La semantica di un programma è data dalla costruzione di un ambiente ρ_c per le classi in esso dichiarate e dalla semantica del blocco a partire da tale ambiente. Dato che il blocco può operare su variabili e oggetti, la sua semantica opera, oltre che a partire dall'ambiente ρ_c, anche su uno stato σ e su uno heap ζ, inizialmente vuoti. Formalmente:

$$\frac{\langle \texttt{Cdl},\ \omega\rangle \rightarrow_{classdecl} \rho_c \qquad \langle \texttt{Stat\_List}, \langle \rho_c, \omega.\Omega, \omega\rangle\rangle \rightarrow_{slist} \langle \sigma, \zeta\rangle}{\langle \texttt{prog\{Cdl \{Stat\_List\}\}}\rangle \rightarrow_{prog} \langle \sigma, \zeta\rangle} \quad (prog)$$

Si noti che le configurazioni del sottosistema $\rightarrow_{com}$ sono cambiate rispetto a quelle riportate nella sezione 5.2. Adesso un blocco va eseguito non più rispetto a un semplice stato, ma rispetto ad una tripla $\langle \rho_c, \sigma, \zeta\rangle$ dove:

- ρ_c è l'ambiente delle classi dichiarate all'inizio del programma;
- σ è una pila di frame (inizialmente vuota) necessaria per modellare l'annidamento dei blocchi, come visto nella sezione 4.2;
- ζ è uno heap (inizialmente vuoto) che associa locazioni ad oggetti, necessario per il trattamento degli oggetti come visto nella sezione 5.5.

Da ora in poi chiameremo *stato* una tripla siffatta. Inoltre, le configurazioni terminali del sistema $\rightarrow_{com}$ sono adesso una coppia del tipo $\langle \sigma, \zeta\rangle$, che rappresenta le componenti modificabili dello stato (si noti infatti che l'ambiente delle classi, una volta costruito, non viene più modificato).

Le regole per il sistema $\rightarrow_{com}$ viste nella Sezione 5.2 vanno opportunamente riviste a seguito della modifica dello stato. Rivediamo come esempi la regola per l'assegnamento e la regola (com_{if-tt}), lasciando per esercizio la modifica delle altre.

$$\frac{\langle \texttt{E}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \texttt{v}}{\langle \texttt{x=E;}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{com} \langle \sigma[^{\texttt{v}}/_{\texttt{x}}], \zeta\rangle} \qquad (com_{=})$$

$$\frac{\langle \texttt{E}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} tt \qquad \langle \texttt{C1}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{com} \langle \sigma', \zeta'\rangle}{\langle \texttt{if (E) C1 else C2}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{com} \langle \sigma', \zeta'\rangle} \qquad (com_{if-tt})$$

Come potevamo aspettarci, anche le configurazioni del sistema $\rightarrow_{exp}$ sono modificate, rispetto a quelle viste nella sezione 4.1.1, per tener conto del nuovo stato. Anche per esse, vediamo un esempio di regola modificata:

$$\frac{\langle \texttt{E}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \texttt{n} \qquad \langle \texttt{E}', \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \texttt{n}' \qquad \texttt{m} = \texttt{n} + \texttt{n}'}{\langle \texttt{E+E}', \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \texttt{m}} \qquad (exp_{+})$$

Più significativa è la modifica dell'unica regola di $\rightarrow_{exp}$ in cui la componente stato gioca un ruolo cruciale:

$$\frac{\sigma(\texttt{x}) = \texttt{n}}{\langle \texttt{x}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{exp} \texttt{n}} \qquad (exp_{ide})$$

Anche le regole del sottosistema $\rightarrow_{slist}$ vanno opportunamente riviste. Considieriamo ad esempio la regola $(slist_{list})$ che diventa:

$$\frac{\langle \texttt{S}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{slist} \langle \sigma'', \zeta''\rangle \qquad \langle \texttt{Slist}, \langle \rho_c, \sigma'', \zeta''\rangle\rangle \rightarrow_{slist} \langle \sigma', \zeta'\rangle}{\langle \texttt{S Slist}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{slist} \langle \sigma', \zeta'\rangle} \qquad (slist_{list})$$

Vediamo la definizione del sottosistema $\rightarrow_{classdecl}$ utilizzato in $(prog)$.

$$\boxed{\frac{\langle \texttt{IVdl}, \omega\rangle \rightarrow_{instdecl} \varphi \qquad \langle \texttt{Mdl}, \omega\rangle \rightarrow_{methoddecl} \rho_m}{\langle \texttt{class c \{IVdl Mdl\}}, \rho_c\rangle \rightarrow_{classdecl} \rho_c[^{(\varphi, \rho_m)}/_{\texttt{c}}]} \qquad (classdecl)}$$

$$\boxed{\frac{\langle \texttt{Cd}, \rho_c\rangle \rightarrow_{classdecl} \rho_c' \qquad \langle \texttt{Cdl}, \rho_c'\rangle \rightarrow_{classdecl} \rho_c''}{\langle \texttt{Cd Cdl}, \rho_c\rangle \rightarrow_{classdecl} \rho_c''} \qquad (classdecl_{list})}$$

Nella regola $(classdecl)$, a partire dalla sequenza di dichiarazioni di variabili istanza `IVdl` viene costruito un frame φ (premessa $\langle \texttt{IVdl}, \omega\rangle \rightarrow_{instdecl} \varphi$) che verrà utilizzato per rappresentare lo stato interno di ciascun oggetto della classe. Inoltre, a partire dalla sequenza di dichiarazioni di metodi `Mdl` viene costruito l'ambiente dei metodi ρ_m che associa a ciascun metodo la sua rappresentazione (premessa $\langle \texttt{Mdl}, \omega\rangle \rightarrow_{methoddecl} \rho_m$). Il sottosistema $\rightarrow_{methoddecl}$ verrà discusso nella sezione 5.8.
Il sottosistema $\rightarrow_{instdecl}$ è una semplificazione del sottosistema $\rightarrow_{dec}$, le cui regole sono le seguenti.

$$\boxed{\langle \texttt{public T x;}, \varphi\rangle \rightarrow_{instdecl} \varphi[^{\varpi}/_{\texttt{x}}] \qquad (instdecl)}$$

$$\frac{\langle \texttt{IVd},\ \varphi\rangle \rightarrow_{instdecl} \varphi' \qquad \langle \texttt{IVdl}, \varphi'\rangle \rightarrow_{instdecl} \varphi''}{\langle \texttt{IVd}\ \ \texttt{IVdl}, \varphi\rangle \rightarrow_{instdecl}\ \varphi''} \quad (instdecl_{list})$$

La semplificazione rispetto a $\rightarrow_{dec}$ è dovuta al fatto che l'unica componente dello stato richiesta è il frame costruito a partire dalle dichiarazioni delle variabili istanza.

In riferimento al programma dell'esempio 5.5, l'ambiente delle classi ρ_c dopo la valutazione delle dichiarazioni di `Foo` e `Fie` è la seguente funzione:

$$\rho_c = \{\texttt{Foo} \mapsto (\varphi_{Foo}, \rho_m^{Foo}), \texttt{Fie} \mapsto (\varphi_{Fie}, \rho_m^{Fie})\}$$

dove i frames φ_{Foo} e φ_{Fie} sono:

$$\varphi_{Foo} = \{\texttt{x} \mapsto \varpi, \texttt{y} \mapsto \varpi\}$$
$$\varphi_{Fie} = \{\texttt{x} \mapsto \varpi\}$$

Le strutture semantiche $\rho_m^{Foo}, \rho_m^{Fie} \in$ `Method_Env` sono solo indicate: come vedremo nella sezione 5.8 in questo semplice caso in cui le due classi non definiscono alcun metodo, tali strutture coincidono con ω.

Possiamo rappresentare graficamente la situazione come segue:

ρ_c

`Foo`	`x` $\boxed{\varpi}$ `y` $\boxed{\varpi}$	ω
`Fie`	`x` $\boxed{\varpi}$	ω

Per procedere con la semantica del blocco principale (seconda premessa della regola (*prog*)), dobbiamo formalizzare il trattamento della dichiarazione di variabili oggetto e dell'operazione `new`, visti informalmente nelle sezioni precedenti.
Consideriamo innanzitutto l'operazione `new`: il suo effetto è quello di costruire la rappresentazione del nuovo oggetto, associarla ad un nuovo riferimento nello heap e restituire, oltre allo heap modificato, anche tale riferimento. Dunque il risultato della valutazione di `new c` è una coppia (l, ζ), dove l è il riferimento nel nuovo heap ζ alla rappresentazione creata. Il riferimento l deve essere distinto da tutti i riferimenti già in uso nello heap, e a questo scopo utilizziamo una funzione *newloc* che, dato uno heap, restituisce un riferimento non in uso. Formalmente, la regola per la valutazione di `new c` è la seguente:

$$\frac{\rho_c(\texttt{c}) = (\varphi, \rho_m) \qquad l = newloc(\zeta)}{\langle \texttt{new c}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{new} \langle l, \zeta[^{(\texttt{c},\varphi)}/_l]\rangle} \quad (new)$$

Quando si valuta `new c`, dove `c` è il nome della classe, occorre accedere alla descrizione di `c` nell'ambiente delle classi - premessa $\rho_c(\texttt{c}) = (\varphi, \rho_m)$ - per poi aggiungere nello heap la rappresentazione del nuovo oggetto, costituito dalla coppia $(\texttt{c}, \varphi)$.

Possiamo ora dare la semantica formale della dichiarazione di una variabile oggetto, estendendo il sistema di transizioni $\rightarrow_{dec}$. Nelle regole che seguono `c` sta per il nome di una classe.

$$\frac{\sigma = \varphi.\sigma'}{\langle \texttt{c x;}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{dec} \langle \varphi[^{null}/_{\texttt{x}}].\sigma', \zeta\rangle} \quad (dec_{obj-1})$$

$$\frac{\sigma = \varphi.\sigma' \qquad \langle \texttt{new c}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{new} \langle l, \zeta'\rangle}{\langle \texttt{c x = new c;}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{dec} \langle \varphi[^{l}/_{\texttt{x}}].\sigma', \zeta'\rangle} \quad (dec_{obj-2})$$

Come si vede, in entrambi i casi lo stato viene modificato aggiungendo, nel frame più recente, una nuova coppia per la variabile oggetto appena dichiarata, cui viene associato un riferimento. Quest'ultimo è il riferimento speciale `null` nel caso in cui non venga creata una nuova rappresentazione attraverso l'operazione `new`.

Osserviamo che le configurazioni terminali del sottosistema $\rightarrow_{dec}$ sono ora una coppia in cui:

- la prima componente è la pila di frame dove al frame più recente è stata aggiunta una associazione per la variabile oggetto dichiarata (come accade nel caso delle variabili semplici);
- la seconda componente è lo heap, che può risultare modificato a seguito della inizializzazione della variabile oggetto appena dichiarata (nel caso della regola (dec_{obj-2})).

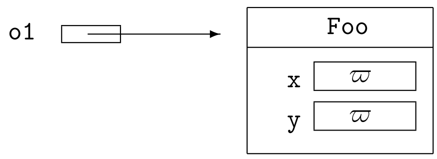
Queste considerazioni implicano ovviamente che anche tutte le regole già definite per $\rightarrow_{dec}$ vanno riviste opportunamente. Riportiamo come esempio la regola (dec_{var-1})

$$\frac{\sigma = \varphi.\sigma'' \qquad\qquad \sigma' = \varphi[^{\varpi}/_{\mathtt{x}}].\sigma''}{\langle \mathtt{T\ x;}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{dec} \langle \sigma', \zeta\rangle} \qquad (dec_{var-1})$$

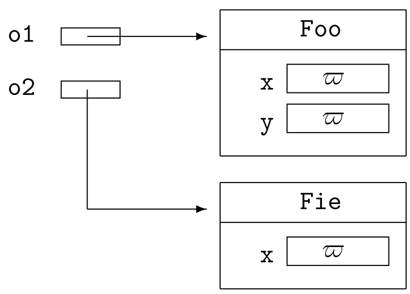
e lasciamo per esercizio la modifica delle altre regole.

In riferimento all'esempio 5.5, possiamo dare una rappresentazione grafica dello stato successivo alla dichiarazione `Foo o1 = new Foo;` come segue, dove gli elementi di `Loc` vengono rappresentati mediante delle frecce.

frame corrente heap



A seguito della seconda dichiarazione `Fie o2 = new Fie;`, la situazione precedente evolve come rappresentato nella seguente figura:



Durante l'esecuzione di un programma è possibile associare ad una variabile oggetto precedentemente dichiarata un nuovo oggetto (purché della stessa classe). Un assegnamento del tipo `o = new c;`, dove `o` è l'identificatore di un oggetto di classe `c`, ha proprio questo scopo. Per trattare tale situazione, estendiamo il sistema $\rightarrow_{com}$ con la nuova regola seguente.

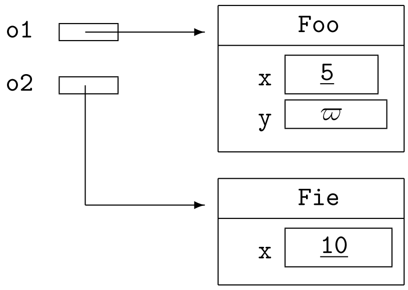
$$\boxed{\frac{\langle \mathtt{new\ c}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{new} \langle l, \zeta'\rangle}{\langle \mathtt{o} = \mathtt{new\ c;}, \langle \rho_c, \sigma, \zeta\rangle\rangle \rightarrow_{com} \langle \sigma[^{l}/_{\mathtt{o}}], \zeta'\rangle} \qquad (com_{=new})}$$

Procediamo con la semantica dell'assegnamento di valori a variabili istanza (come in `o1.x=5` dell'esempio). Dobbiamo estendere opportunamente il sistema $\rightarrow_{com}$ a questo caso.

$$\frac{\langle \mathtt{E}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{exp} \mathtt{v} \qquad \zeta(\sigma(\mathtt{o})) = (\mathtt{c}, \varphi) \qquad \varphi' = \varphi[^{\mathtt{v}}/_{\mathtt{x}}]}{\langle \mathtt{o.x} = \mathtt{E};, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{com} \langle \sigma, \zeta[^{(\mathtt{c},\varphi')}/_{\sigma(\mathtt{o})}]\rangle} \quad (com_{=2})$$

A partire dall'identificatore dell'oggetto in questione (l'identificatore `o` nella regola), si accede alla sua rappresentazione $\zeta(\sigma(\mathtt{o})) = (\mathtt{c}, \varphi)$. Il frame φ associato all'oggetto viene quindi modificato associando alla variabile istanza `x` il valore `v` dell'espressione `E` calcolato nello stato corrente. Tale modifica si riflette nello heap della configurazione di arrivo con l'aggiornamento della seconda componente della coppia $\sigma(\mathtt{o})$ associata nello heap all'oggetto.

Consideriamo di nuovo l'esempio 5.5 e rappresentiamo lo stato successivo ai primi due assegnamenti `o1.x=5; o2.x=10;`.

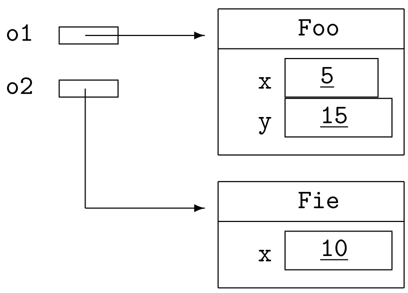


Le variabili istanza possono essere utilizzate anche nelle espressioni, come `o1.x` e `o2.x` nell'assegnamento `o1.y = o1.x + o2.x;` dell'esempio precedente: dobbiamo quindi estendere anche il sottosistema $\rightarrow_{exp}$.

$$\frac{\zeta(\sigma(\mathtt{o})) = (\mathtt{c}, \varphi)}{\langle \mathtt{o.x}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{exp} \varphi(\mathtt{x})} \quad (exp_{ref})$$

Il valore della variabile istanza `x` dell'oggetto `o` viene prelevato nel frame associato a $\sigma(\mathtt{o})$ nello heap corrente.

Concludendo l'esempio 5.5, lo stato dopo l'ultimo assegnamento `o1.y = o1.x + o2.x` è rappresentato dalla seguente figura:



5.8 Metodi

Gli oggetti che vengono utilizzati nei programmi Java sono in grado di compiere azioni, mediante l'esecuzione di *metodi*. Al momento della dichiarazione di una classe vengono introdotti, oltre alle variabili istanza, i metodi propri di quella classe. Ad ogni oggetto della classe potrà essere chiesto di operare mediante l'invocazione dei suoi metodi. È fondamentale avere sempre presente la distinzione tra la *dichiarazione* di un metodo e la sua *invocazione*: la dichiarazione avviene una volta per tutte contestualmente alla dichiarazione della classe e definisce la funzionalità del metodo; l'invocazione o chiamata corrisponde all'utilizzo effettivo del metodo, alla esecuzione vera e propria delle azioni ad esso associate. La chiamata del metodo `m` dell'oggetto `o` corrisponde a chiedere all'oggetto `o` di eseguire le azioni descritte da `m`. In generale, invocazioni diverse dello stesso metodo possono avere effetti diversi e questo per due possibili ragioni:

- il metodo opera in funzione dello stato dell'oggetto, rappresentato dai valori delle sue variabili istanza al momento della chiamata del metodo;
- il metodo è parametrico, cioè definito in funzione di uno o più parametri che possono essere istanziati con valori diversi in chiamate diverse.

Per semplicità, nel nostro linguaggio semplificato consideriamo metodi con un solo parametro. Inoltre, le restrizioni sul nostro linguaggio impongono che gli unici effetti che i metodi possono avere siano la modifica dello stato degli oggetti coinvolti[11].

5.8.1 Dichiarazione dei metodi

Diamo innanzitutto la sintassi delle dichiarazioni di metodi, che avevamo lasciato in sospeso nella sezione precedente, definendo le produzioni per la categoria sintattica `Method_decl_list`.

```
Method_decl_list ::= Method_decl |
                     Method_decl Method_decl_list

Method_decl ::= public void Ide ( Type Ide ) Block
```

Di seguito un esempio di dichiarazione di classe in cui è presente un metodo.

Esempio 5.6 Consideriamo la seguente definizione della classe `Incr`

```
class Incr
    {
        public int x;
        public void inc (int y)
           {
             x = x+y;
           }
    }
```

■

Anche per le dichiarazioni dei metodi abbiamo trattato solo un caso particolare, per semplicità di esposizione. Nello specifico:

- abbiamo trattato solo metodi che non restituiscono un risultato, ovvero operano solo sulle loro variabili di istanza;
- abbiamo assunto che i metodi abbiano un solo parametro.

[11] In Java, la presenza di variabili di tipo `static` consente in generale di modificare anche lo stato globale.

Nel seguito, data una dichiarazione del tipo

```
public void m ( T x ) B
```

chiameremo *corpo* di `m` il blocco `B` e parametro *formale* di `m` l'identificatore `x`.

Supponendo che `o` sia un oggetto della classe `Incr` dell'esempio precedente, la chiamata del metodo `inc` di `o` deve avvenire istanziando il parametro `y` ad un valore intero: detto `v` tale valore, l'effetto dell'esecuzione del metodo è di incrementare di `v` il valore della variabile istanza `x` di `o`. Vedremo più avanti in questa sezione la sintassi e la semantica della chiamata dei metodi. È chiaro comunque che chiamate diverse del metodo possono essere effettuate istanziando in modi diversi il parametro del metodo stesso.

Al momento della dichiarazione di una classe, viene creata la rappresentazione semantica dei suoi metodi attraverso il sottosistema $\rightarrow_{methoddecl}$, utilizzato nella premessa della regola (*classdecl*) della sezione 5.7, riportata per comodità qui di seguito:

$$\frac{\langle \texttt{IVdl}, \omega\rangle \rightarrow_{instdecl} \varphi \qquad \langle \texttt{Mdl}, \omega\rangle \rightarrow_{methoddecl} \rho_m}{\langle \texttt{class a \{ IVdl \ \ Mdl\}}, \rho_c\rangle \rightarrow_{classdecl} \rho_c[^{(\varphi,\rho_m)}/_{\texttt{a}}]} \qquad (classdecl)$$

Lo scopo del sottosistema $\rightarrow_{methoddecl}$ è di creare la struttura ρ_m che contiene una rappresentazione dei metodi della classe. Tale struttura è un elemento dell'insieme `Method_Env` (si veda la sezione 5.5) e può essere vista come una tabella che associa ad ogni metodo `m` tutti gli ingredienti necessari al momento della chiamata del metodo stesso, ovvero:

- il nome utilizzato per il parametro del metodo;
- il blocco che rappresenta le azioni intraprese dal metodo.

Come ormai usuale, anche gli elementi di `Method_Env` vengono rappresentati mediante funzioni. Dunque, ρ_m è una funzione che, dato il nome di un metodo, restituisce una coppia in cui il primo elemento è il nome del parametro del metodo e la seconda componente è il blocco del metodo. Formalmente:

$$\rho_m : \texttt{Ide} \rightarrow \texttt{Ide} \times \texttt{Block}$$

Siamo finalmente in grado di definire il sottosistema $\rightarrow_{methoddecl}$ che formalizza quanto appena esposto.

$$\boxed{\langle \texttt{public void m(T x) B}, \rho_m\rangle \rightarrow_{methoddecl} \rho_m[^{(\texttt{x},\texttt{B})}/_{\texttt{m}}] \qquad (methoddecl)}$$

$$\boxed{\frac{\langle \texttt{Md}, \ \rho_m\rangle \rightarrow_{methoddecl} \rho'_m \qquad \langle \texttt{Mdl}, \rho'_m\rangle \rightarrow_{methoddecl} \rho''_m}{\langle \texttt{Md \ \ Mdl}, \rho_m\rangle \rightarrow_{methoddecl} \rho''_m} \qquad (methoddecl_{list})}$$

5.8.2 Chiamata dei metodi

Abbiamo già accennato al fatto che la chiamata di un metodo `m` dell'oggetto `o` corrisponde a chiedere all'oggetto `o` di eseguire le azioni descritte da `m` e definite nella dichiarazione di `m` presente all'interno della dichiarazione della classe cui `o` appartiene. Nel nostro frammento di Java, la chiamata di un metodo è un comando, secondo la seguente estensione della categoria sintattica `Com`.

```
Com ::=  Ide.Ide(Exp);
```

Nella chiamata `o.m(E)`, `o` deve essere l'identificatore di un oggetto creato in precedenza e `m` deve essere l'identificatore di un metodo della classe di `o`. L'espressione `E` costituisce l'istanziazione del parametro di `m` (ricordiamo che, per semplicità, nel nostro frammento consideriamo metodi che hanno esattamente un parametro).

Riconsideriamo la definizione della classe `Incr` dell'esempio 5.6.

```
class Incr
    {
        public int x;
        public void inc (int y)
            {
              x = x+y;
            }
    }
```

Supponendo che `obj` sia una variabile oggetto creata con la dichiarazione `Incr obj=new Incr;`, una possibile chiamata di metodo è data da `obj.inc(8);`. Intuitivamente, l'effetto di tale chiamata dovrebbe essere di incrementare di $\underline{8}$ il valore della variabile istanza `x` dell'oggetto `obj`. Infatti, l'unico comando del metodo aumenta il valore di `x` del valore di `y`, il parametro, che in questo caso è stato istanziato con `8`.

Per distinguere tra il parametro al momento della dichiarazione ed il parametro al momento della chiamata, chiameremo *parametro attuale* quest'ultimo, ricordando che il primo viene invece riferito come *parametro formale*[12]. Dunque nell'esempio che stiamo considerando, `y` è il parametro formale di `m` e `8` è il parametro attuale nella chiamata `obj.inc(8);`.

Vediamo come, da un punto di vista semantico, si trattano le chiamate dei metodi in modo da ottenere l'effetto intuitivo come delineato nell'esempio. Consideriamo dunque una generica chiamata del tipo:

`o.m(E);`

Le azioni da intraprendere sono le seguenti:

(i) reperire nello stato tutte le componenti necessarie alla valutazione della chiamata, ovvero la coppia $(\mathtt{x}, \mathtt{B})$ associata al metodo `m` nell'ambiente delle classi;

(ii) valutare nello stato corrente l'espressione `E` in modo da ottenere il valore del parametro attuale da associare al parametro formale `x`;

(iii) valutare il corpo `B` a partire da uno stato in cui il parametro attuale è stato opportunamente legato al parametro formale `x`.

Il fatto che la chiamata di un metodo non possa modificare altro che le variabili istanza dell'oggetto coinvolto, si riflette dal punto di vista semantico nella struttura della seconda componente σ dello stato in cui viene eseguito il corpo `B`. In prima approssimazione tale componente è semplicemente

$$\{\mathtt{x} \mapsto \mathtt{v}\}.\Omega \qquad\qquad (\dagger)$$

ovvero una pila che contiene un solo frame contenente una sola associazione, quella tra il parametro formale ed il parametro attuale. In realtà tale scelta è troppo semplicistica: infatti non saremmo in grado, in uno stato siffatto, di accedere alle variabili istanza dell'oggetto in questione, né per reperirne né per modificarne il valore. Riferendoci ancora al metodo `inc` dell'esempio 5.6, non saremmo in grado di valutare l'assegnamento `x=x+y;` in quanto lo stato (†) contiene sì una associazione per l'identificatore `y`, il parametro formale del metodo, ma non contiene alcuna associazione per l'identificatore `x`, la variabile istanza dell'oggetto coinvolto.

[12] Il termine *attuale* viene dall'inglese *actual*, cioè *reale*, *effettivo*.

Un modo per ovviare a tale inconveniente consiste nel supporre che, nel corpo dei metodi, ogni riferimento ad una variabile istanza `x` sia rimpiazzato dal riferimento `this.x`, in cui il simbolo speciale `this` sta ad indicare l'oggetto coinvolto. Così, ad esempio, il precedente assegnamento `x=x+y;` viene trasformato in

$$\texttt{this.x} = \texttt{this.x} \ + \ \texttt{y};$$

Così facendo, basta che lo stato in cui viene eseguito un metodo permetta di accedere, oltre che alla associazione tra parametro attuale e parametro formale, anche alla associazione tra l'identificatore speciale `this` ed il riferimento all'oggetto coinvolto nella chiamata. Ad esempio, in corrispondenza di una chiamata del tipo `o.m(E);`, lo stato (†) diventa

$$\{\texttt{x} \mapsto \texttt{v}, \texttt{this} \mapsto \ell\}.\Omega \qquad\qquad (\dagger\dagger)$$

dove ℓ sta ad indicare il riferimento associato all'oggetto `o` nello stato al momento della chiamata.

Uno stato come (††) riflette in maniera esplicita il fatto che durante l'esecuzione di un metodo un oggetto è in grado di accedere esclusivamente al suo proprio stato interno, oltre che al parametro del metodo. Osserviamo inoltre che un riferimento del tipo `this.x` all'interno di una espressione, viene correttamente risolto a partire dalle associazioni in (††), mediante la regola (exp_{ref}). Analogamente, un assegnamento del tipo $\texttt{this.x} = ...$ viene correttamente risolto a partire dalle associazioni in (††) grazie alla regola ($com_{=2}$).

Formalmente otteniamo la seguente regola per la chiamata dei metodi, che estende il sistema $\rightarrow_{com}$.

$$\frac{\begin{array}{c}\zeta(\sigma(\texttt{o})) = (\texttt{c}, \varphi') \qquad \rho_c(\texttt{c}) = (\varphi, \rho_m) \qquad \rho_m(\texttt{m}) = (\texttt{x}, \texttt{B}) \\ \langle \texttt{E}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{exp} \ \texttt{v} \qquad \sigma' = \omega[{}^{\texttt{v}}/_{\texttt{x}}, {}^{\sigma(\texttt{o})}/_{\texttt{this}}].\Omega \\ \langle \texttt{B}, \langle \rho_c, \sigma', \zeta \rangle\rangle \rightarrow_{com} \ \langle \sigma'', \zeta' \rangle \end{array}}{\langle \texttt{o.m(E)}, \langle \rho_c, \sigma, \zeta \rangle\rangle \rightarrow_{com} \langle \sigma, \zeta' \rangle} \qquad (com_{call})$$

È importante osservare che la precedente regola semantica mette bene in evidenza come le ripercussioni, ovvero modifiche, sullo stato di arrivo si hanno solo sulla componente heap dello stato stesso. Ciò significa che la chiamata di un metodo può modificare solo variabili istanza di oggetti, ed in particolare dell'oggetto coinvolto nella chiamata.

Esempio 5.7 Consideriamo di nuovo la classe dell'esempio 5.6, che in realtà deve essere riscritta come segue per rappresentare i riferimenti alle variabili istanza:

```
class Incr
    {
        public int x;
        public void inc (int y)
           {
             this.x = this.x+y;
           }
    }
```

Consideriamo una chiamata del tipo `obj.inc(10);` a partire dallo stato $\langle \rho_c, \varphi.\Omega, \zeta \rangle$ in cui:

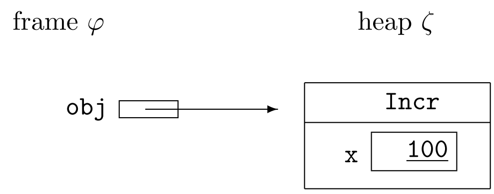
- l'ambiente delle classi ρ_c è il seguente:

ρ_c	`Incr`	`x` $\boxed{\varpi}$	ρ_m

dove ρ_m è a sua volta:

ρ_m	`inc`	`y`	`{this.x=this.x+y}`

- il frame φ e lo heap ζ sono rappresentati dalla seguente figura:



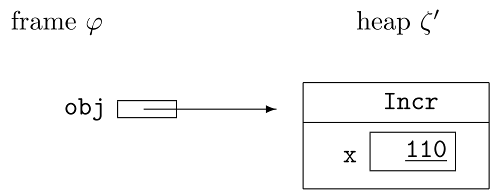
Utilizzando la regola (com_{call}), è facile convincersi che si può costruire la derivazione

$\langle$`obj.inc(10);`$, \langle \rho_c, \sigma, \zeta \rangle\rangle$

$\rightarrow_{com}$

$\langle \sigma, \zeta' \rangle$

dove il nuovo heap ζ' è rappresentato come segue:

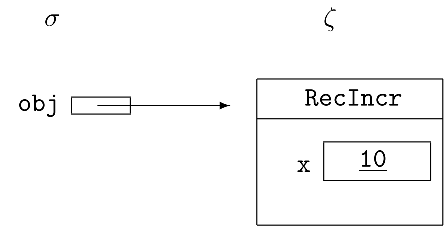


5.8.3 Metodi ricorsivi

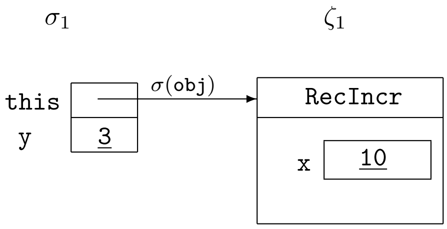
Il linguaggio Java, così come il frammento da noi utilizzato, consente la definizione di metodi *ricorsivi.* Un metodo è ricorsivo se, nella sua definizione, utilizza chiamate a sé stesso. Vediamo brevemente come la nostra semantica operazionale consenta di trattare correttamente anche questa caratteristica del linguaggio. A questo scopo, consideriamo quale esempio la seguente classe.

```
class RecIncr
    {
        public int x;
        public void inc (int y)
           {
             if (y<=1)
                this.x = this.x+1;
             else
                {
                    this.x=this.x+1;
                    this.inc (y-1);
                }
           }
    }
```

Supponiamo che `obj` identifichi un oggetto della classe sopra descritta e supponiamo di eseguire la chiamata `obj.inc(3)` a partire da uno stato in cui la pila di frame e lo heap contengono, tra le altre, le associazioni descritte nella seguente figura.

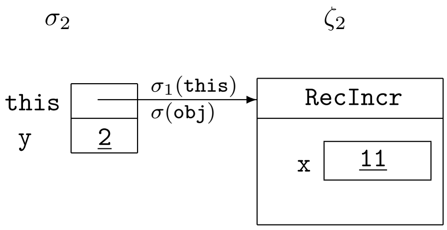


Vediamo come evolve la chiamata `obj.incr(3)`, mostrando graficamente solo le componenti significative dello stato. Secondo la regola (com_{call}), il corpo del metodo viene eseguito nello stato in cui le componenti "pila di frame" e heap sono le seguenti:



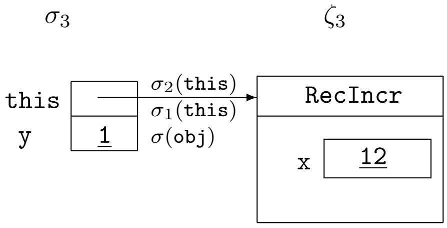
Per chiarezza, abbiamo indicato esplicitamente il valore $\sigma($`obj`$)$ del riferimento associato all'identificatore speciale `this`; si noti inoltre che $\zeta_1 = \zeta$.

Poiché nello stato corrente il valore di `y` è strettamente maggiore di $\underline{1}$, viene eseguito il ramo `else` del corpo del metodo: dopo l'assegnamento `this.x=this.x+1` viene eseguita (ancora grazie alla regola (com_{call})) la chiamata ricorsiva `this.inc(y-1)` nello stato seguente:

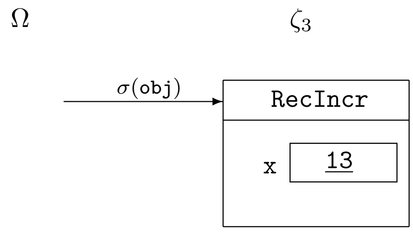


Si noti che la chiamata ricorsiva viene eseguita a partire da un *nuovo* frame che contiene l'associazione per il parametro attuale della chiamata ricorsiva stessa, oltre che l'associazione per l'identificatore speciale `this`: quest'ultimo è associato al riferimento $\sigma_1($`this`$)$, che equivale, a sua volta a $\sigma($`obj`$)$. Dunque, ad ogni chiamata ricorsiva l'identificatore `this` presente nell'unico frame dello stato è sempre associato all'oggetto di partenza.

L'esecuzione della chiamata ricorsiva porta ad un'ulteriore chiamata ricorsiva `this.inc(y-1)` che avviene nello stato:



Poiché in questo stato il valore di `y` è uguale a 1, viene eseguito il solo comando `this.x = this.x +1`. Inoltre la ricorsione ha termine e viene restituito lo stato rappresentato dalla seguente figura:

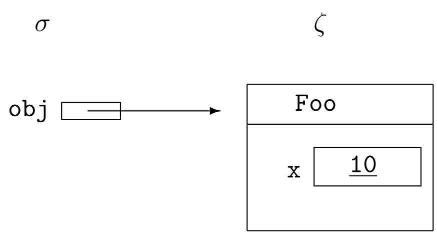


È questo lo stato che viene restituito, a loro volta, dalle chiamate ricorsive del metodo `inc` e dalla stessa chiamata iniziale `obj.this(3);`. Come ci potevamo aspettare, l'effetto di quest'ultima sullo stato complessivo si riflette nel fatto che il valore della variabile istanza `x` di `obj` è stato incrementato di 3.

Consideriamo adesso un ulteriore esempio che mette meglio in luce come gli effetti delle chiamate ricorsive si ripercuotano a ritroso dalla chiamata più annidata a quella iniziale.

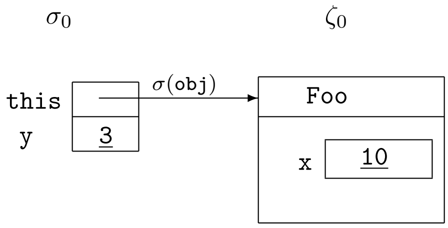
```
class Foo
    {
        public int x;
        public void fie (int y)
           {
             if (y<=1)
                this.x = this.x+1;
             else
                this.fie (y-1);

             this.x = 2*this.x;
           }
    }
```

Come nell'esempio precedente, eseguiamo la chiamata `obj.fie(3)`, assumendo che `obj` identifichi un oggetto della classe `Foo` e che lo stato iniziale sia quello descritto dalla figura seguente.

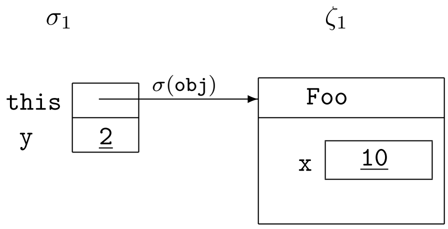


È facile convincersi che la sequenza degli stati in cui vengono eseguite le chiamate ricorsive annidate è la seguente:

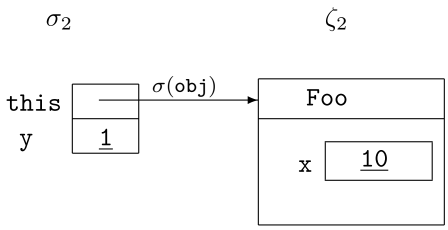
Chiamata iniziale



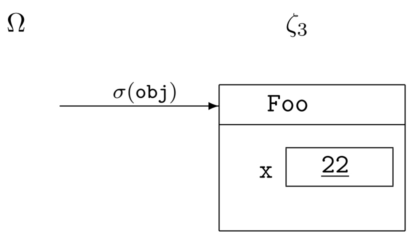
1^a chiamata ricorsiva



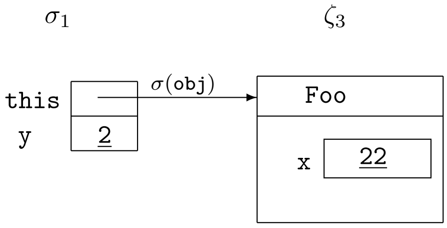
2^a chiamata ricorsiva



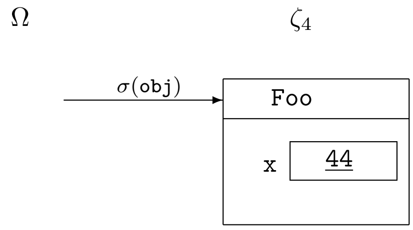
Poiché in quest'ultimo stato il valore di `y` è $\underline{1}$, viene eseguito il primo ramo del comando condizionale (l'assegnamento `this.x = this.x+1;`) e, successivamente, il comando `this.x = 2*this.x;`, con il quale ha termine la 2^a chiamata ricorsiva. Lo stato restituito è il seguente:



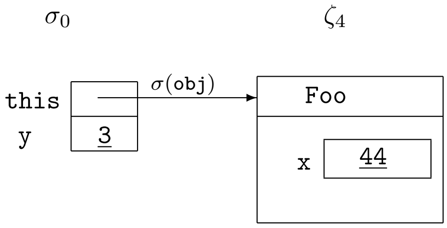
A questo punto si prosegue con l'esecuzione dell'ultimo comando (`this.x=2*this.x;`) della prima chiamata ricorsiva, a partire dallo stato



Si osservi come la componente heap dello stato rifletta le modifiche apportate dalla chiamata ricorsiva annidata. Con l'esecuzione dell'assegnamento `this.x=2*this.x;` ha termine anche la 1^a chiamata ricorsiva, che restituisce lo stato



Si prosegue infine con l'esecuzione dell'ultimo comando (`this.x=2*this.x;`) della chiamata iniziale, a partire dallo stato



che termina restituendo complessivamente lo stato