Elementi di Semantica Operazionale: Nucleo del C

Appunti per gli studenti di Programmazione

Corso di Laurea in Informatica Applicata
Polo Universitario G. Marconi - La Spezia

Università di Pisa
A.A. 2008/09

M.E.Occhiuto

Indice

Prefazione alla versione 2008/2009

I primi due capitoli di queste note sono i primi due capitoli delle note Elementi di Semantica Operazionale, di R.Barbuti, P.Mancarella e F.Turini utilizzate per diversi anni come dispense del corso di Fondamenti di Programmazione dei Corso di Laurea in Informatica e di Informatica Applicata dell'Università di Pisa. I successivi capitoli sono un rimaneggiamento dei capitoli su corrispondenti argomenti della medesima dispensa, nel senso che ne mantengono il più possibile: struttura, considerazioni, esempi ed esercizi. I sistemi di transizione che definiscono la semantica sono diversi perchè diverso è il linguaggio e diverso è lo stato, ma la teoria alla base è la stessa.

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.

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 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. □

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 $\gamma \to \gamma'$ 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 $\mathtt{S} = \langle \Gamma, \mathtt{T}, \to \rangle$, 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 $k \geq 1$, $\gamma_{k-1} \to \gamma_k$. Nel seguito, una derivazione in $\mathtt{S}$ verrà indicata con $\gamma_0 \to \gamma_1 \to \ldots \to \gamma_i$. Inoltre indicheremo con $\gamma \xrightarrow{*} \gamma'$ l'esistenza di una derivazione $\gamma_0 \to \gamma_1 \to \ldots \to \gamma_i$ in cui $\gamma_0 = \gamma$ e $\gamma_i = \gamma'$. □

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

Come primi esempi, consideriamo le grammatiche 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 $\alpha \to_G \beta$ 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 $S \xrightarrow{*}_G \alpha$, dove $\mathcal{L}(G)$ indica il linguaggio generato da G. ■

2.1 Regole condizionali

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

$$\frac{\pi_1 \quad \pi_2 \ldots \pi_n}{\gamma \to \gamma'}$$

dove $\pi_1, \pi_2 \ldots \pi_n$ (dette *precondizioni* o *premesse* della regola) sono i prerequisiti alla transizione $\gamma \to \gamma'$. Così, ad esempio, la definizione di $\to_G$ dell'esempio 2.3 può essere data come segue. Per ogni produzione $(A ::= \eta) \in P$ si definisce una regola condizionale:

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

Se consideriamo, 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 condizionali del corrispondente sistema di transizioni sono:

$$\frac{\alpha = \delta \mathtt{S} \gamma \quad \delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^* \quad \beta = \delta \mathtt{ab} \gamma}{\alpha \rightarrow \beta}$$

$$\frac{\alpha = \delta \mathtt{S} \gamma \quad \delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^* \quad \beta = \delta \mathtt{aSb} \gamma}{\alpha \rightarrow \beta}$$

Più precisamente, quando scriviamo

$$\frac{\pi_1 \quad \pi_2 \ldots \pi_n}{\gamma \rightarrow \gamma'}$$

stiamo in realtà esprimendo un insieme (in generale infinito) di regole, come risulterà chiaro più avanti in questa sezione [1].

Inoltre è necessario essere precisi su quale forma possano assumere le premesse delle regole condizionali, al fine di evitare di scrivere regole prive di significato operazionale. Riconsideriamo, a questo scopo, la regola data in precedenza:

$$\frac{\alpha = \delta \mathtt{S} \gamma \quad \delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^* \quad \beta = \delta \mathtt{ab} \gamma}{\alpha \rightarrow \beta}$$

Si noti che sia nella conclusione che nelle premesse della regola compaiono:

- nomi, come $\alpha, \beta, \gamma, \delta$ che chiameremo *variabili* e che rappresentano generici elementi di un opportuno dominio del discorso (ad esempio, nel caso specifico, α, β, γ e δ rappresentano stringhe);
- costanti, come $\mathtt{S}, \mathtt{a}, \mathtt{b}$ che stanno per oggetti specifici e noti (ad esempio, $\mathtt{S}$ è il simbolo iniziale della grammatica in esame, $\mathtt{a}$ e $\mathtt{b}$ sono i simboli nell'alfabeto della grammatica).

Le premesse della regola sono poi *predicati* o *relazioni* tra espressioni (termini) costruite a partire dalle variabili, dalle costanti e da operatori noti. Ad esempio, la premessa $\alpha = \delta \mathtt{S} \gamma$ esprime un'uguaglianza tra stringhe in cui compaiono:

- le variabili α, γ e δ,
- la costante $\mathtt{S}$,
- l'operatore $=$ di uguaglianza tra stringhe e l'operatore di concatenazione tra stringhe nell'espressione $\delta \mathtt{S} \gamma$(si noti che tale operatore è rappresentato dalla semplice giustapposizione delle stringhe in questione).

Lo scopo di tale premessa è di specificare le parti che compongono la stringa α: un generico prefisso, chiamato δ, seguito dalla categoria sintattica $\mathtt{S}$, seguita dalla parte conclusiva, chiamata γ. Si noti che tali variabili vengono utilizzate anche nella premessa $\beta = \delta \mathtt{ab} \gamma$, esprimendo così il fatto che le stringhe α e β hanno in comune il prefisso (δ) e la parte conclusiva (γ).

Ancora, la premessa $\delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^*$ esprime il fatto che le componenti δ e γ delle stringhe α e β rappresentano stringhe costruite a partire da simboli terminali e categorie sintattiche della grammatica in esame: in tale premessa vengono utilizzati gli operatori noti $\cup$, * e la relazione di appartenenza $\in$.

Come vedremo nel seguito, in molti casi consentiremo anche che le premesse di una regola che definiscono una relazione di transizione $\rightarrow$ possano contenere istanze di transizioni del tipo $\delta \rightarrow' \delta'$ in cui $\rightarrow'$ può essere sia la relazione di transizione di un altro sistema di transizioni sia la relazione di transizione $\rightarrow$ stessa del sistema in esame. In questo secondo caso parleremo di regola *ricorsiva*.

Riassumiamo quanto detto circa le regole condizionali che definiscono una relazione di transizione. Data una regola del tipo:

[1] Dovremmo quindi parlare, più che di regola, di *schema di regola*.

$$\boxed{\dfrac{\pi_1 \quad \pi_2 \ldots \pi_n}{\gamma \rightarrow \gamma'} \qquad (R)}$$

ciascuna premessa può essere:

(i) un'uguaglianza del tipo $x = t$, dove x è una variabile e t è una espressione (termine) costruita a partire da costanti, variabili e operatori elementari;

(ii) un predicato del tipo $t \; rel \; t'$, dove t, t' sono termini e rel è un operatore di relazione elementare;

(iii) una transizione del tipo $\delta \rightarrow \delta'$, dove δ, δ' sono configurazioni e $\rightarrow$ è la relazione di transizione di un sistema di transizioni (che può anche essere il sistema che si sta definendo).

È importante osservare che le variabili utilizzate in una regola condizionale devono essere introdotte allo scopo di determinare condizioni significative per la applicabilità della regola stessa. Nell'esempio precedente, le variabili δ e γ sono state introdotte al fine di mettere in relazione la struttura delle stringhe α e β presenti nella conclusione della regola stessa, come abbiamo già messo in evidenza. Bisogna fare attenzione a non introdurre variabili inutili e condizioni su di esse che potrebbero rendere la regola inapplicabile. Si consideri ad esempio la seguente (pseudo) regola:

$$\frac{\alpha, \beta, \gamma \in \{\mathtt{a}, \mathtt{b}\}^* \quad \alpha = \mathtt{a}\beta\gamma \qquad n, m > 0 \qquad n = m + 1}{\alpha \rightarrow \beta}$$

Mentre è chiaro il ruolo giocato dalla variabile γ, come parte conclusiva della stringa α, è invece oscuro quello giocato dalle variabili n, m a valori naturali. Osserviamo che l'attribuzione di valori concreti alle variabili n e m potrebbe rendere inapplicabile la regola, secondo la definizione di istanziazione o applicabilità data più avanti.

La seguente nozione di *copertura* delle variabili presenti in una regola di transizione consente di stabilire se una regola è ben definita per quanto concerne l'uso delle variabili in essa contenute (anche se ciò non garantisce la correttezza della regola stessa).

Intuitivamente, una variabile è coperta se il suo valore, o direttamente o usando le premesse, è ottenibile dalla configurazione a cui la regola risulta applicabile.

Nella definizione, data una configurazione γ (risp. termine t), indichiamo con $Vars(\gamma)$ (risp. $Vars(t)$) l'insieme di tutte e sole le variabili che compaiono in essa. Analogamente, data una regola condizionale R, indichiamo con $Vars(R)$ l'insieme di tutte e sole le variabili che vi compaiono.

Definizione 2.4 (Copertura delle variabili)
Sia R una regola condizionale

$$\frac{\pi_1 \quad \pi_2 \ldots \pi_n}{\gamma \rightarrow \gamma'}$$

e sia $x \in Vars(R)$. Diciamo allora che x è *coperta* in R se e solo se vale una delle seguenti condizioni:

(1) x occorre in γ;

(2) esiste in R una premessa π_i del tipo $y = t$ tale che y è coperta in R e $x \in Vars(t)$;

(3) esiste in R una premessa π_i del tipo $x = t$ tale che tutte le variabili in $Vars(t)$ sono coperte;

(4) esiste in R una premessa π_i del tipo $\delta \rightarrow' \delta'$ tale che tutte le variabili in δ sono coperte in R e $x \in Vars(\delta')$ □

È utile osservare che le eventuali premesse del tipo $t \; rel \; t'$, con rel operatore di relazione diverso dall'uguaglianza, non contribuiscono alla copertura delle variabili: tali premesse vengono introdotte infatti allo scopo di verificare l'applicabilità di una regola ed esprimono condizioni che hanno a che fare con le componenti delle configurazioni presenti nella regola stessa.

Vediamo alcuni esempi di regole condizionali e di copertura delle loro variabili.

Esempio 2.5 Consideriamo di nuovo la regola condizionale

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

introdotta in precedenza. Le variabili di tale regola sono tutte e sole quelle nell'insieme $\{\alpha, \beta, \gamma, \delta\}$. Osserviamo che:

- α è coperta (caso (1) della Definizione 2.4);
- δ e γ sono coperte (caso (2) della Definizione 2.4 e punto precedente);
- β è coperta (caso (3) della Definizione 2.4 e punto precedente).

Un modo alternativo per esprime la precedente regola è il seguente

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

le cui variabili δ e γ sono evidentemente coperte.

Consideriamo ora la regola

$$\frac{\alpha, \beta, \gamma \in \{\mathtt{a}, \mathtt{b}\}^* \quad \alpha = \mathtt{a} \beta \gamma \qquad n, m > 0 \qquad n = m + 1}{\alpha \to \beta}$$

le cui variabili sono tutte e sole quelle nell'insieme $\{\alpha, \beta, \gamma, n, m\}$. Mentre è chiaro che α, β e γ sono tutte coperte, ciò non vale per le variabili n ed m.

Non diamo, per il momento, esempi di regole condizionali che contengono premesse del tipo $\delta \to' \delta'$: la loro utilità risulterà chiara nel seguito.

Nel resto di queste note considereremo solo regole le cui variabili sono tutte coperte secondo quanto stabilito dalla Definizione 2.4.

In che senso un insieme di regole condizionali definisce la relazione di transizione $\to$ di un sistema di transizioni $\mathtt{S} = \langle \Gamma, \mathtt{T}, \to \rangle$? Per rispondere a questa domanda dobbiamo dare un procedimento che ci consenta di stabilire quando una coppia (γ, γ'), con $\gamma, \gamma' \in \Gamma$, appartiene alla relazione $\to$ secondo quanto stabilito dalle regole condizionali.

Riprendiamo l'esempio riportato all'inizio di questa sezione, in cui la relazione di transizione del sistema è definita dalle seguenti regole condizionali.

$$\frac{\alpha = \delta \mathtt{S} \gamma \quad \delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^* \quad \beta = \delta \mathtt{ab} \gamma}{\alpha \to \beta} \qquad \text{(r1)}$$

$$\frac{\alpha = \delta \mathtt{S} \gamma \quad \delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^* \quad \beta = \delta \mathtt{aSb} \gamma}{\alpha \to \beta} \qquad \text{(r2)}$$

e consideriamo la configurazione $\mathtt{aaSbb}$. Vogliamo verificare l'esistenza o meno di una configurazione λ tale che $\mathtt{aaSbb} \to \lambda$. Prendiamo in esame la regola (r1), con $\alpha = \mathtt{aaSbb}$ e osserviamo che:

- ponendo $\delta = \mathtt{aa}$ e $\gamma = \mathtt{bb}$, la precondizione $\mathtt{aaSbb} = \delta \mathtt{S} \gamma$ è verificata;
- per costruzione δ e γ soddisfano la proprietà espressa dalla seconda precondizione;

- ponendo $\beta = \delta\texttt{ab}\gamma$, ovvero $\beta = \texttt{aaabbb}$, anche l'ultima precondizione della regola è soddisfatta;
- dalle due considerazioni precedenti, la regola (r1) ci permette di concludere l'esistenza della transizione $\texttt{aaSbb} \rightarrow \texttt{aaabbb}$.

Abbiamo dunque concluso, dalla regola (r1), l'esistenza di una configurazione $\lambda = \texttt{aaabbb}$ tale che $\texttt{aaSbb} \rightarrow \lambda$.

Il procedimento che abbiamo intrapreso nell'esempio consiste nel determinare dei valori concreti da utilizzare per rimpiazzare le variabili presenti in una regola, in modo da ottenere un caso particolare della regola stessa, detta *istanza* della regola, in cui non occorrono più variabili e tutte le precondizioni sono soddisfatte. Chiameremo questo procedimento *istanziazione* di una regola.

Si noti che, nello stesso esempio, avremmo potuto arrivare alla conclusione che esiste la transizione $\texttt{aaSbb} \rightarrow \texttt{aaaSbbb}$ applicando un procedimento analogo ma a partire dalla regola (r2)[2].

Consideriamo ora la configurazione aabb. Osserviamo che non è possibile *istanziare* opportunamente né la regola (r1) né la regola (r2) in modo che le rispettive premesse siano soddisfatte. In particolare, in entrambi i casi non è possibile determinare valori concreti per le variabili δ e γ in modo che $\texttt{aabb} = \delta\texttt{S}\gamma$. Possiamo dunque dire che, la relazione di transizione definita dalle regole (r1) e (r2) non contiene alcuna coppia $(\texttt{aabb}, \lambda)$.

Per formalizzare il processo di istanziazione di una regola introduciamo il concetto di *sostituzione*.

Definizione 2.6 (Sostituzione)
Sia $V = \{x_1, \ldots, x_n\}$ un insieme di variabili distinte e sia $c_1, \ldots, c_n$ una sequenza di costanti. L'insieme di associazioni $\vartheta = \{c_1/x_1, \ldots, c_n/x_n\}$ è detta *sostituzione* per le variabili in V. □

Una volta definita una sostituzione, essa può essere applicata per *istanziare* oggetti in cui compaiono le variabili coinvolte nella sostituzione: nel nostro caso specifico applicheremo sostituzioni alle espressioni (termini) o alle configurazioni presenti in una regola condizionale.

Definizione 2.7 (Applicazione di una sostituzione)
Sia X un termine o una configurazione e sia $V = Vars(X)$ l'insieme delle variabili che vi compaiono. Sia inoltre ϑ una sostituzione per le variabili in V. L'applicazione di ϑ a X, denotata da $(X)\vartheta$ è il termine o la configurazione che si ottiene rimpiazzando in X ogni variabile x con la costante c tale che $c/x \in \vartheta$. Il termine o configurazione $(X)\vartheta$ verrà detto *istanza di X via ϑ*. □

Ad esempio, data la configurazione $\delta S\gamma$ dell'esempio precedente e la sostituzione $\vartheta = \{\texttt{aa}/\gamma, \texttt{bb}/\delta\}$, l'istanza $(\delta S\gamma)\vartheta$ è la configurazione aaSbb. Come ulteriore esempio, data l'espressione $E = x > y+1$ e la sostituzione $\vartheta = \{3/x, 0/y\}$, l'istanza $(E)\vartheta$ è l'espressione $3 > 0+1$. Data invece $\vartheta' = \{3/x, 5/y\}$, l'istanza $(E)\vartheta'$ è l'espressione $3 > 5 + 1$. Quest'ultimo esempio mette in luce che l'operazione di istanziazione è puramente *sintattica*: istanziazioni diverse della stessa espressione danno luogo ad espressioni che hanno, in generale, *significati* diversi. Ad esempio, utilizzando il significato comune dei simboli che vi compaiono, la relazione $(E)\vartheta$, ovvero $3 > 0 + 1$, è vera mentre è falsa la relazione $(E)\vartheta'$, ovvero $3 > 5 + 1$.

È evidente che il concetto di istanziazione dato nella Definizione 2.7 può essere facilmente esteso al caso di regole condizionali.

Definizione 2.8 (Istanza di una regola)
Sia R una regola condizionale, sia $V = Vars(R)$ l'insieme di tutte e sole le variabili che occorrono in R e sia ϑ una sostituzione per le variabili in V. L'*istanza di R via ϑ* è la regola condizionale che si ottiene applicando ϑ a tutti i termini e a tutte le configurazioni che occorrono in R. □

Consideriamo ad esempio la regola (r2) data in precedenza

$$\frac{\alpha = \delta\texttt{S}\gamma \quad \delta, \gamma \in (\{\texttt{a}, \texttt{b}\} \cup \{\texttt{S}\})^* \quad \beta = \delta\texttt{aSb}\gamma}{\alpha \rightarrow \beta} \qquad \text{(r2)}$$

[2] Dimostrarlo per esercizio.

e la sostituzione $\vartheta = \{\mathtt{aaSbb}/\alpha, \mathtt{aa}/\delta, \mathtt{bb}/\gamma, \mathtt{aaaSbbb}/\beta\}$. L'istanza di (r2) via ϑ è la regola condizionale

$$\frac{\mathtt{aaSbb} = \mathtt{aaSbb} \quad \mathtt{aa}, \mathtt{bb} \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^* \quad \mathtt{aaaSbbb} = \mathtt{aaaSbbb}}{\mathtt{aaSbb} \rightarrow \mathtt{aaaSbbb}} \quad (\mathrm{r2})\vartheta$$

Osserviamo che le premesse dell'istanza così ottenuta sono tutte soddisfatte. Consideriamo ora la sostituzione $\vartheta' = \{\mathtt{aaSbb}/\alpha, \mathtt{a}/\delta, \epsilon/\gamma, \mathtt{aaSb}/\beta\}$. L'istanza di (r2) via ϑ' è la regola condizionale

$$\frac{\mathtt{aaSbb} = \mathtt{aaS} \quad \mathtt{a}, \epsilon \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^* \quad \mathtt{aaSb} = \mathtt{aaSb}}{\mathtt{aaSbb} \rightarrow \mathtt{aaSb}} \quad (\mathrm{r2})\vartheta'$$

Questa volta, nell'istanza ottenuta le premesse non sono tutte soddisfatte. In particolare è falsa la prima premessa $\mathtt{aaSbb} = \mathtt{aS}$, e ciò ad indicare che la scelta delle costanti associate alle variabili α, δ e γ non consente una corretta "scomposizione" della stringa associata ad α in $\delta\mathtt{S}\gamma$.

Siamo finalmente in grado di definire precisamente in che senso un insieme di regole condizionali definisce una relazione di transizione in un sistema di transizioni.

Definizione 2.9 Sia $\mathtt{S} = \langle \Gamma, \mathtt{T}, \rightarrow \rangle$ un sistema di transizioni la cui relazione di transizione $\rightarrow$ è definita mediante un insieme finito di regole condizionali $(r_1), \ldots, (r_n)$. Date due configurazioni $\gamma, \gamma' \in \Gamma$, la coppia (γ, γ') appartiene alla relazione $\rightarrow$ se e soltanto se esiste una sostituzione ϑ per le variabili in $Vars(r_i)$, con $1 \leq i \leq n$, tale che:

- tutte le premesse di $(r_i)\vartheta$ sono soddisfatte;
- la conclusione di $(r_i)\vartheta$ è $\gamma \rightarrow \gamma'$. □

Consideriamo come esempio le regole di transizione $(r1)$ e $(r2)$ date in precedenza e osserviamo:

- $\mathtt{aaSbb} \rightarrow \mathtt{aaaSbbb}$: infatti la sostituzione ϑ vista in precedenza produce un'istanza della regola $(r2)$ le cui premesse sono tutte soddisfatte;
- $\mathtt{aaSbb} \not\rightarrow \mathtt{aaSaSbb}$: è facile convincersi, infatti, che non è possibile individuare una sostituzione ϑ in modo la conclusione di $(r1)\vartheta$ o $(r2)\vartheta$ sia esattamente $\mathtt{aaSbb} \rightarrow \mathtt{aaSaSbb}$ e in modo che le premesse delle regole così istanziate siano tutte soddisfatte[3].

La Definizione 2.9 suggerisce anche un modo per determinare, a partire da una configurazione γ, una configurazione γ' tale che $\gamma \rightarrow \gamma'$, se quest'ultima esiste. Vediamolo attraverso un esempio, utilizzando ancora le regole $(r1)$ e $(r2)$ viste in precedenza. Supponiamo di voler determinare, se esiste, una configurazione β tale che $\mathtt{abaSbb} \rightarrow \beta$. Analizziamo dapprima la regola $(r1)$:

$$\frac{\alpha = \delta\mathtt{S}\gamma \quad \delta, \gamma \in (\{\mathtt{a}, \mathtt{b}\} \cup \{\mathtt{S}\})^* \quad \beta = \delta\mathtt{ab}\gamma}{\alpha \rightarrow \beta} \quad (\mathrm{r1})$$

È evidente che una qualunque sostituzione che consenta di applicare la Definizione 2.9 deve contenere l'associazione $\mathtt{abaSbb}/\alpha$. L'istanziazione (parziale) della premessa

$$\alpha = \delta\mathtt{S}\gamma$$

con tale associazione dà luogo alla premessa

$$\mathtt{abaSbb} = \delta\mathtt{S}\gamma$$

e ciò suggerisce di estendere la sostituzione mediante le associazioni $\mathtt{aba}/\delta$ e $\mathtt{bb}/\gamma$. A questo punto non abbiamo altra scelta che completare la sostituzione con l'associazione $\mathtt{abaabbb}/\beta$, se vogliamo garantire che tutte le premesse siano soddisfatte. Riassumendo, abbiamo così determinato la sostituzione

$$\vartheta = \{\mathtt{abaSbb}/\alpha, \mathtt{abaabbb}/\beta, \mathtt{aba}/\delta, \mathtt{bb}/\gamma\}$$

[3] $\gamma \not\rightarrow \gamma'$ indica il fatto che $(\gamma, \gamma') \notin \rightarrow$.

in modo che tutte le premesse della regola $(r2)\vartheta$ siano soddisfatte. Con tale sostituzione, secondo la Definizione 2.9, il sistema contiene la transizione $\texttt{abaSbb} \rightarrow \texttt{abaabaa}$.

Ragionando in modo del tutto analogo possiamo costruire la sostituzione

$$\vartheta' = \{\texttt{abaSbb}/\alpha, \texttt{abaaSbbb}/\beta, \texttt{aba}/\delta, \texttt{bb}/\gamma\}$$

per concludere, grazie all'istanza $(r2)\vartheta'$, l'esistenza della transizione $\texttt{abaSbb} \rightarrow \texttt{abaaSbbb}$.

2.2 Esempi di sistemi di transizioni

2.2.1 Grammatiche come sistemi di transizioni

Nel paragrafo precedente, data una grammatica libera G abbiamo definito un sistema di transizioni S_G la cui relazione di transizione corrisponde esattamente alla nozione di *derivazione* nella grammatica, nel senso che $\alpha \rightarrow_G \beta$ se e solo se la stringa β è ottenibile dalla stringa 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 nella stringa.

Esempio 2.10 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 $(A ::= \eta) \in P$:

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

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

2.2.2 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 $\mathtt{x}$ appartiene al linguaggio associato alla categoria sintattica $\mathtt{T}$: scriveremo, ad esempio, $\mathtt{0} \in \mathtt{Cif}$, $\mathtt{n} \in \mathtt{Num}$ e così via.

$\mathtt{S}_{cr}$: 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:

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

La relazione di transizione è data dalle seguenti regole:

$\langle 0,0,0 \rangle \to_{cr} \langle 0,0 \rangle$	$\langle 0,0,1 \rangle \to_{cr} \langle 1,0 \rangle$
$\langle 0,1,0 \rangle \to_{cr} \langle 1,0 \rangle$	$\langle 0,1,1 \rangle \to_{cr} \langle 2,0 \rangle$
$\langle 0,2,0 \rangle \to_{cr} \langle 2,0 \rangle$	$\langle 0,2,1 \rangle \to_{cr} \langle 3,0 \rangle$
$\ldots \quad \ldots$	$\ldots \quad \ldots$
$\langle 9,8,0 \rangle \to_{cr} \langle 7,1 \rangle$	$\langle 9,8,1 \rangle \to_{cr} \langle 8,1 \rangle$
$\langle 9,9,0 \rangle \to_{cr} \langle 8,1 \rangle$	$\langle 9,9,1 \rangle \to_{cr} \langle 9,1 \rangle$

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.

$\mathtt{S}_{n+}$: 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:

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

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 54 e 38:

- stadio iniziale del calcolo

```
 54
 38
```

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

- 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 $\langle \mathtt{54}, \mathtt{38} \rangle \rightarrow_{n+} \langle \mathtt{5}, \mathtt{3}, \mathtt{2}, \mathtt{1} \rangle$, 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 $\langle \mathtt{5}, \mathtt{3}, \mathtt{2}, \mathtt{1} \rangle \rightarrow_{n+} \langle \mathtt{0}, \mathtt{0}, \mathtt{92}, \mathtt{0} \rangle$.

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

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:

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

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

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

$$\frac{\langle \mathtt{c}, \mathtt{c}', \mathtt{0} \rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r} \rangle}{\langle \mathtt{c}, \mathtt{c}' \rangle \rightarrow_{n+} \langle \mathtt{0}, \mathtt{0}, \mathtt{c}'', \mathtt{r} \rangle} \quad \text{(i)}$$

$$\frac{\langle \mathtt{c}, \mathtt{c}', \mathtt{0} \rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r} \rangle}{\langle \mathtt{nc}, \mathtt{c}' \rangle \rightarrow_{n+} \langle \mathtt{n}, \mathtt{0}, \mathtt{c}'', \mathtt{r} \rangle} \quad \text{(ii)}$$

$$\frac{\langle \mathtt{c}', \mathtt{c}, \mathtt{0} \rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r} \rangle}{\langle \mathtt{c}', \mathtt{nc} \rangle \rightarrow_{n+} \langle \mathtt{0}, \mathtt{n}, \mathtt{c}'', \mathtt{r} \rangle} \quad \text{(iii)}$$

$$\frac{\langle \mathtt{c}, \mathtt{c}', 0\rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r}\rangle}{\langle \mathtt{nc}, \mathtt{n}'\mathtt{c}'\rangle \rightarrow_{n+} \langle \mathtt{n}, \mathtt{n}', \mathtt{c}'', \mathtt{r}\rangle} \quad \text{(iv)}$$

$$\langle 0, 0, \mathtt{m}, 0\rangle \rightarrow_{n+} \mathtt{m} \quad \text{(v)}$$

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

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

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

$$\frac{\langle \mathtt{c}, \mathtt{c}', \mathtt{r}\rangle \rightarrow_{cr} \langle \mathtt{c}'', \mathtt{r}'\rangle}{\langle \mathtt{nc}, \mathtt{n}'\mathtt{c}', \mathtt{m}, \mathtt{r}\rangle \rightarrow_{n+} \langle \mathtt{n}, \mathtt{n}', \mathtt{c}''\mathtt{m}, \mathtt{r}'\rangle} \quad \text{(ix)}$$

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:

$$\frac{\langle 9, 5, 0\rangle \rightarrow_{cr} \langle 4, 1\rangle}{\langle 379, 5\rangle \rightarrow_{n+} \langle 37, 0, 4, 1\rangle} \qquad \frac{\langle 2, 4, 0\rangle \rightarrow_{cr} \langle 6, 0\rangle}{\langle 82, 4\rangle \rightarrow_{n+} \langle 8, 0, 6, 0\rangle}$$

Le transizioni (i) ÷ (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 $\langle 0, 0, \mathtt{m}, 0\rangle$ 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) ÷ (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 $\langle 0, 0, \mathtt{m}, 0\rangle$, per la quale è applicabile la regola (v).

Esempio 2.11 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:

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

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

$\langle 9,3,73,0\rangle$

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

$\langle 0,0,273,1\rangle$

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

$\langle 0,0,1273,0\rangle$

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

1273

■

2.2.3 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 $\mathbb{N} = \{\underline{0}, \underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}, \ldots\}$ 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 $\mathbb{N}$[4]. Ad ogni elemento `n` di `Num` corrisponde il *valore* in $\mathbb{N}$ di cui `n` è la *rappresentazione*. Tale associazione è data dalla seguente *funzione di valutazione* $\eta : \mathtt{Num} \rightarrow \mathbb{N}$, 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* $\nu : \mathbb{N} \rightarrow \mathtt{Num}$, 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 $\mathbb{N}$ 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 $\mathbb{N}$. 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`).

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

[4]Si noti la necessità di distinguere tra i simboli utilizzati per denotare le operazioni su $\mathbb{N}$ 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 $\mathbb{N}$. 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 `<=`.

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 $\eta' : \texttt{Binary} \to I\!N$ 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`).

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

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

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

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 $\underline{\texttt{n}} \succ \underline{\texttt{m}}$.

2.3 Altri esempi

In questo paragrafo consideriamo un semplice linguaggio e definiamo varie semantiche per le stringhe del linguaggio stesso attraverso la definizione di diversi sistemi di transizioni. Il linguaggio, che indicheremo con $\mathcal{L}$ è quello delle stringhe non vuote formate dai simboli `a` e `b`, ovvero $\mathcal{L} = (\{\texttt{a}, \texttt{b}\})^+$, che può essere definito dalla seguente grammatica

```
A ::= aA | bA | a | b
```

Esempio 2.12 La prima semantica che vogliamo definire è la seguente: il significato di una stringa $\alpha \in \mathcal{L}$ è il numero naturale dato dal numero di occorrenze del simbolo `a` in α. Quindi, ad esempio, le stringhe `aabab` e `aaa` hanno come semantica il numero naturale 3, così come ogni stringa formata da sole `b` rappresenta il numero naturale 0, e così via. Per esprimere tale semantica mediante un sistema di transizioni $S_1 = \langle \Gamma_1, T_1, \rightarrow_1 \rangle$ possiamo procedere come segue:

- Γ_1 dovrà contenere, tra le altre, configurazioni che corrispondono a stringhe del linguaggio, ovvero l'insieme $\{\alpha | \alpha \in \mathcal{L}\}$
- T_1 dovrà essere costituito dall'insieme dei numeri naturali $\mathbb{N}$
- la relazione di transizione dovrà consentire derivazioni del tipo $\alpha \rightarrow_1^* \underline{\mathtt{n}}$ se e soltanto se il simbolo `a` occorre esattamente $\underline{\mathtt{n}}$ volte in α.

Un primo approccio consiste nel definire un sistema la cui relazione di transizione descrive i singoli passi di calcolo che, a partire da una stringa iniziale α, porta alla determinazione dell'opportuna configurazione finale. Gli stadi intermedi di tale calcolo vengono descritti tramite un ulteriore tipo di configurazioni, definite dal seguente insieme:

$$\{\langle \beta, \underline{\mathtt{m}} \rangle \mid \beta \in \mathcal{L}, \underline{\mathtt{m}} \in \mathbb{N}\}$$

Il significato di una configurazione $\langle \beta, \underline{\mathtt{m}} \rangle$ è il seguente:

- β rappresenta la porzione della stringa iniziale che deve essere ancora analizzata;
- $\underline{\mathtt{m}}$ rappresenta il numero di occorrenze di `a` nella porzione della stringa iniziale già analizzata.

Più precisamente, la relazione di transizione dovrà consentire derivazioni del tipo:

$$\alpha \rightarrow_1^* \langle \beta, \underline{\mathtt{k}} \rangle$$

se e soltanto se $\alpha = \beta'\beta$ e il numero di occorrenze di `a` in β' è $\underline{\mathtt{k}}$. Per definire la relazione di transizione, utilizziamo, quando necessario, un approccio *guidato dalla sintassi*, in cui le stringhe vengono analizzate per casi a seconda della loro struttura sintattica. Tale definizione deve garantire che le proprietà sopra enunciate di $\rightarrow_1$ siano soddisfatte. Nella scrittura delle regole condizionali che definiscono $\rightarrow_1$ utilizziamo per brevità la convenzione di denotare con $\alpha, \beta, \ldots$ stringhe del linguaggio.

$$\frac{\alpha = \mathtt{a}\beta \qquad \underline{\mathtt{m}} = \underline{\mathtt{k}} + \underline{\mathtt{1}}}{\langle \alpha, \underline{\mathtt{k}} \rangle \rightarrow_1 \langle \beta, \underline{\mathtt{m}} \rangle} \qquad \text{(r1)}$$

$$\frac{\alpha = \mathtt{b}\beta}{\langle \alpha, \underline{\mathtt{k}} \rangle \rightarrow_1 \langle \beta, \underline{\mathtt{k}} \rangle} \qquad \text{(r2)}$$

$$\frac{\alpha = \mathtt{a} \qquad \underline{\mathtt{m}} = \underline{\mathtt{k}} + \underline{\mathtt{1}}}{\langle \alpha, \underline{\mathtt{k}} \rangle \rightarrow_1 \underline{\mathtt{m}}} \qquad \text{(r3)}$$

$$\frac{\alpha = \mathtt{b}}{\langle \alpha, \underline{\mathtt{k}} \rangle \rightarrow_1 \underline{\mathtt{k}}} \qquad \text{(r4)}$$

$$\frac{}{\alpha \rightarrow_1 \langle \alpha, \underline{\mathtt{0}} \rangle} \qquad \text{(r5)}$$

È facile convincersi che queste regole definiscono correttamente la relazione di transizione $\rightarrow_1$. Un approccio alternativo consiste nel definire $\rightarrow_1$ utilizzando un approccio *induttivo* che astrae dai singoli passi di calcolo e definisce direttamente la relazione tra le configurazioni iniziali e le configurazioni finali, ovvero tra una stringa e la sua semantica. La nuova definizione si basa sulle seguenti considerazioni informali:

- il numero di occorrenze di `a` nella stringa `a` (risp. `b`) è $\underline{1}$ (risp. $\underline{0}$).
- il numero di occorrenze di `a` nella stringa $\mathtt{a}\alpha$ (risp. $\mathtt{b}\alpha$) è $\underline{\mathtt{n}} + \underline{1}$ (risp. $\underline{\mathtt{n}}$) se il numero di occorrenze di `a` in α è $\underline{\mathtt{n}}$.

La traduzione di queste regole informali produce il seguente insieme di regole condizionali:

$$\frac{\alpha \rightarrow_1 \underline{\mathtt{k}} \qquad \underline{\mathtt{n}} = \underline{\mathtt{k}} + \underline{1}}{\mathtt{a}\alpha \rightarrow_1 \underline{\mathtt{n}}} \qquad \text{(r1')}$$

$$\frac{\alpha \rightarrow_1 \underline{\mathtt{k}}}{\mathtt{b}\alpha \rightarrow_1 \underline{\mathtt{k}}} \qquad \text{(r2')}$$

$$\frac{}{\mathtt{a} \rightarrow_1 \underline{1}} \qquad \text{(r3')}$$

$$\frac{}{\mathtt{b} \rightarrow_1 \underline{0}} \qquad \text{(r4')}$$

Si noti che, in questo caso, le regole condizionali contengono premesse che fanno riferimento alla relazione di transizione stessa. Come già accennato, il sistema che si ottiene in questo modo è più *astratto* del precedente: non si considerano, in questo approccio, configurazioni intermedie che descrivono stadi intermedi del calcolo, ma si definisce direttamente la relazione tra le configurazioni di partenza e le configurazioni terminali. Invece, nel caso delle regole (r1) ÷ (r5) date in precedenza, le transizioni definiscono i singoli passi del calcolo. Si rimanda alla prossima Sezione per un breve confronto tra i due approcci, detti, rispettivamente, semantica di valutazione e semantica naturale.

Esempio 2.13 Definiamo ora una semantica diversa per il linguaggio $\mathcal{L}$: il significato di una stringa α è il numero naturale $\underline{\mathtt{n}}$ se e soltanto se $\alpha = \mathtt{a}^{\underline{\mathtt{n}}}\mathtt{b}^{\underline{\mathtt{k}}}$, con $\underline{\mathtt{n}}, \underline{\mathtt{k}} \geq \underline{0}$. Si noti che in questa semantica non tutte le stringhe di $\mathcal{L}$ hanno un significato: ad esempio, mentre il significato di `aaabb` è $\underline{3}$, la stringa `ababa` non ha alcun significato. Anche nel sistema di transizioni $S_2 = \langle \Gamma_2, T_2, \rightarrow_2 \rangle$ che ci accingiamo a definire avremo che Γ_2 contiene configurazioni *iniziali* del tipo α con $\alpha \in \mathcal{L}$ e che T_2 è l'insieme dei numeri naturali. Questa volta la relazione di transizione dovrà consentire sia il calcolo del numero di occorrenze di `a` sia la verifica che la stringa data ha una struttura opportuna. A questo scopo introduciamo configurazioni "intermedie" del tipo $\langle \beta, \underline{\mathtt{m}}, x \rangle$ in cui le componenti β e $\underline{\mathtt{m}}$ rappresentano rispettivamente (come nell'esempio precedente) la porzione di stringa che rimane da analizzare ed il numero di occorrenze di `a` presenti nella porzione di stringa già analizzata, mentre x segnala l'ultimo simbolo analizzato. In questo modo possiamo fare in modo che configurazioni del tipo $\langle \mathtt{a}\alpha, \underline{\mathtt{n}}, \mathtt{b} \rangle$ siano *bloccate* in quanto corrispondono a stringhe che non hanno semantica.

Le regole condizionali che definiscono $\rightarrow_2$ sono le seguenti:

$$\frac{\underline{\mathtt{m}} = \underline{\mathtt{k}} + \underline{\mathtt{1}}}{\langle \mathtt{a}\beta, \underline{\mathtt{k}}, \mathtt{a}\rangle \rightarrow_2 \langle \beta, \underline{\mathtt{m}}, \mathtt{a}\rangle} \qquad \text{(r1)}$$

$$\frac{}{\langle \mathtt{b}\beta, \underline{\mathtt{k}}, \mathtt{a}\rangle \rightarrow_2 \langle \beta, \underline{\mathtt{k}}, \mathtt{b}\rangle} \qquad \text{(r2)}$$

$$\frac{}{\langle \mathtt{b}\beta, \underline{\mathtt{k}}, \mathtt{b}\rangle \rightarrow_2 \langle \beta, \underline{\mathtt{k}}, \mathtt{b}\rangle} \qquad \text{(r3)}$$

$$\frac{\underline{\mathtt{m}} = \underline{\mathtt{k}} + \underline{\mathtt{1}}}{\langle \mathtt{a}, \underline{\mathtt{k}}, \mathtt{a}\rangle \rightarrow_2 \underline{\mathtt{m}}} \qquad \text{(r4)}$$

$$\frac{}{\langle \mathtt{b}, \underline{\mathtt{k}}, \mathtt{a}\rangle \rightarrow_2 \underline{\mathtt{m}}} \qquad \text{(r5)}$$

Per quanto concerne le transizioni relative alle configurazioni iniziali del tipo α, possiamo introdurre una sola regola condizionale

$$\frac{}{\alpha \rightarrow_2 \langle \alpha, \underline{\mathtt{0}}, \mathtt{a}\rangle} \qquad \text{(r6)}$$

Si noti che in questa regola la terza componente della configurazione di arrivo non corrisponde al suo significato informale descritto in precedenza, ovvero non è l'ultimo simbolo analizzato. Tuttavia, questa formulazione della regola ci evita di prevedere quattro regole distinte corrispondenti ai quattro casi sintattici per α. Osserviamo infine che le regole (r2) e (r3) possono essere inglobate in una sola regola

$$\frac{}{\langle \mathtt{b}\beta, \underline{\mathtt{k}}, x\rangle \rightarrow_2 \langle \beta, \underline{\mathtt{k}}, \mathtt{b}\rangle}$$

In questo esempio, l'approccio astratto non è immediato come nel caso dell'esempio precedente. Ne diamo qui di seguito una possibile versione.

$$\frac{\beta \rightarrow_2 \underline{\mathtt{k}} \qquad \underline{\mathtt{m}} = \underline{\mathtt{k}} + \underline{\mathtt{1}}}{\mathtt{a}\beta \rightarrow_2 \underline{\mathtt{m}}} \qquad \text{(r1')}$$

$$\frac{\beta \rightarrow_2 \underline{\mathtt{0}}}{\mathtt{b}\beta \rightarrow_2 \underline{\mathtt{0}}} \qquad \text{(r2')}$$

$$\frac{}{\mathtt{a} \rightarrow_2 \underline{\mathtt{1}}} \qquad \text{(r3')}$$

$$\frac{}{\mathtt{b} \rightarrow_2 \underline{\mathtt{0}}} \qquad \text{(r4')}$$

La premessa della regola (r2') assicura l'applicabilità della regola stessa solo nel caso in cui la stringa ancora da esaminare, β, non contiene il simbolo $\mathtt{a}$.

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 $\underline{\texttt{n}}$;

(r2) il valore di un'espressione del tipo `E+E'` si ottiene sommando i valori $\underline{\texttt{n}}$ e $\underline{\texttt{m}}$, dove $\underline{\texttt{n}}$ è il valore di `E` e $\underline{\texttt{m}}$ 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 $\underline{\texttt{n}} + \underline{\texttt{m}}$, dove $\underline{\texttt{n}}$ è il valore di `2` e $\underline{\texttt{m}}$ è il valore di `3`.

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

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

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

Sappiamo inoltre che

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

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 $\mathtt{E} \to \mathtt{n}$, 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

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

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 $\mathtt{E} \to \mathtt{n}$ 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:

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

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 $\mathtt{2} \to \mathtt{n}$ e $\mathtt{3} \to \mathtt{m}$, 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 $\mathbb{N}$.

Riassumendo, abbiamo le seguenti configurazioni:

$$\begin{aligned} \Gamma &= \{\mathtt{E} \mid \mathtt{E} \in \mathtt{Exp}\} \cup \\ &\quad\ \{\underline{\mathtt{n}} \mid \underline{\mathtt{n}} \in \mathbb{N}\} \\ \mathtt{T} &= \{\underline{\mathtt{n}} \mid \underline{\mathtt{n}} \in \mathbb{N}\} \end{aligned}$$

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 $\mathtt{E'}$ rappresentano elementi della categoria sintattica Exp.

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

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

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 $\underline{\texttt{k}} = \underline{\texttt{n}} + \underline{\texttt{m}}$ 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 2.2.2. 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 $\texttt{2+3+8} \to \underline{\texttt{m}}$ per qualche $\underline{\texttt{m}}$, applicando opportunamente le regole (r1) e (r2).

`2+3+8`

$\to$ { (r2),
(r1): $\texttt{2} \to \underline{\texttt{2}}$
(d1): $\texttt{3} + \texttt{8} \to \underline{\texttt{11}}, \quad \underline{\texttt{2}} + \underline{\texttt{11}} = \underline{\texttt{13}}$}

$\underline{\texttt{13}}$

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):
`3+8`

$\to$ { (r2),
(r1): $\texttt{3} \to \underline{\texttt{3}}$
(r1): $\texttt{8} \to \underline{\texttt{8}}, \quad \underline{\texttt{3}} + \underline{\texttt{8}} = \underline{\texttt{11}}$}

$\underline{\texttt{11}}$

■

In generale, una derivazione è dunque costituita da una derivazione *principale* e da tante *sottoderivazioni* quante sono le transizioni non banali del tipo $\gamma \to \gamma'$ 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'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]`[5].

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): $\texttt{2} + \texttt{3} \to \underline{\texttt{5}}$

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

(r1): $\texttt{8} \rightarrow \underline{\texttt{8}}$
$\underline{\texttt{5}} + \underline{\texttt{8}} = \underline{\texttt{13}}$ }

$\underline{\texttt{13}}$

con la sottoderivazione (d1) ottenuta come segue:

(d1):
`2+3`

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

$\underline{\texttt{5}}$

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, 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:

$$\begin{aligned} \Gamma_{exp} &= \{\texttt{E} \mid \texttt{E} \in \texttt{Exp}\} \cup \\ &\quad\ \{\underline{\texttt{n}} \mid \underline{\texttt{n}} \in I\!N\} \\ \mathtt{T}_{exp} &= \{\underline{\texttt{n}} \mid \underline{\texttt{n}} \in I\!N\} \end{aligned}$$

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 $\underline{\texttt{n}}+\underline{\texttt{m}} = \underline{\texttt{k}}$ della regola (r2) del paragrafo precedente). Per semplicità, tali premesse verranno rappresentate mediante uguaglianze del tipo $\underline{\texttt{n}} + \underline{\texttt{m}} = \underline{\texttt{k}}$, $\underline{\texttt{n}} \times \underline{\texttt{m}} = \underline{\texttt{k}}$, ecc.

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

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

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

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

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

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

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

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 $\underline{\mathtt{n}}' \neq \underline{0}$, 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 $[(\mathtt{2}+\mathtt{3})]*\mathtt{5} \rightarrow_{exp} [\mathtt{2}+\mathtt{3}]*\mathtt{5}$.

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:

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

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 $\underline{18}$.

`25 - 15 / 2`

$\rightarrow_{exp}$ { $(exp_{\_})$,
(exp_n): `25` $\rightarrow_{exp}$ $\underline{25}$,
(d1): `15 / 2` $\rightarrow_{exp}$ $\underline{7}$,
$\underline{25} \geq \underline{7}, \quad \underline{25} - \underline{7} = \underline{18}$ }

$\underline{18}$

dove (d1) è data da:

(d1):

`15 / 2`

$\rightarrow_{exp}$ { (exp_{div}),
(exp_n): `15` $\rightarrow_{exp}$ $\underline{15}$,
(exp_n): `2` $\rightarrow_{exp}$ $\underline{2}$,
$\underline{2} \neq \underline{0}, \quad \underline{15}\ div\ \underline{2} = \underline{7}$ }

$\underline{7}$

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). `Ide` è la categoria sintattica a partire dalla quale è possibile generare identificatori. 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[6]. Per descrivere la semantica di espressioni siffatte, aggiungiamo un ulteriore ingrediente al nostro sistema di transizioni, il cosiddetto *stato*. In questa sezione introduciamo lo stato nella sua forma più semplice, ovvero come un insieme di associazioni tra nomi e valori, mentre successivamente avremo bisogno di uno stato più strutturato, per trattare alcuni meccanismi di **C**, quali l'annidamento di blocchi e i puntatori.

4.1 I frame

Nella sua forma più semplice, uno stato non è altro che un insieme di associazioni del tipo $\langle \mathtt{i}, \mathtt{v} \rangle$ dove `i` è un identificatore e `v` è un valore. Useremo il termine *frame* per fare riferimento ad un tale insieme. Indichiamo genericamente con `Ide` l'insieme degli identificatori validi in **C**, e con `Val` l'insieme dei valori associati ai nomi nello stato, per quanto visto sino ad ora `Val` è l'insieme $\mathbb{N}$, allora uno stato 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 frame 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 frame. Inoltre, adotteremo l'usuale notazione $\varphi(\mathtt{x})$ per denotare il valore *associato ad* `x` *nello frame* φ. Dato uno frame 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 frame 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 frame φ; un modo per rappresentare tale frame 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 frame *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.

4.2 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 $\underline{\mathtt{n}}$.

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

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

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

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

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

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

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

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

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

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:

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

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 $\varphi(\mathtt{y}) = \underline{2}$.

$\langle$`(x+8) / y`$, \varphi\rangle$

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

$\underline{6}$

(d1):

$\langle$`(x+8)`$, \varphi\rangle$

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

$\underline{13}$

(d2):

$\langle$x+8, $\varphi\rangle$

$\rightarrow_{exp}$ { (exp_+),
(exp_{ide}): $\langle$x, $\varphi\rangle \rightarrow_{exp} \underline{5}$,
($exp^{:}_{n}$) $\langle 8, \varphi\rangle \rightarrow_{exp} \underline{8}$,
$\underline{13} = \underline{5} + \underline{8}$ }

$\underline{13}$

■

4.3 Strutturazione dello stato

Per poter trattare correttamente caratteristiche del linguaggio **C** più complesse quali l'annidamento dei blocchi, le chiamate di funzione e i puntatori è necessario dare allo stato una strutturazione più sofisticata rispetto a quella introdotta nella Sezione 4.1. Lo stato qui introdotto è quello su cui viene data la semantica del nucleo del C, definita nelle sezioni successive. Tale stato è costituito da due componenti l'*ambiente* e la *memoria*, che vengono definite nelle prossime sezioni. Per definire il nuovo stato è necessario introdurre i domini della semantica, due di questi li abbiamo già introdotti nella Sezione 4.1, e cioè `Ide` e `Val`, quest'ultimo è esteso all'insieme degli interi. Riassumendo i domini sono i seguenti:

- `Ide` l'insieme degli identificatori validi del **C**
- `Val` $= \mathcal{I}nt \cup \{\varpi\} \cup \{$`Undef`$\}$. Dove $\mathcal{I}$nt è l'insieme dei valori interi, `Undef` è il valore idefinito che rappresenta ad una situazione di errore mentre ϖ è il valore indefinito di tipo `int`.
- `Loc` l'insieme degli indirizzi delle locazioni di memoria, possono essere rappresentati come numeri naturali. Nel seguito useremo la notazione 0_L, 1_L, 2_L ...ecc, per indicare rispettivamente le locazioni di indirizzo $0, 1, 2$ ecc. e non confonderle con i valori corrispondenti.
- `FunctDecl` l'insieme delle dichiarazioni di funzione. Le vedremo quando tratteremo le funzioni.
- `Den` = `Loc` $\cup$ `FunDecl` $\cup$ `Unbound`. L'insieme delle *denotazioni* è l'unione dell'insieme `Loc` degli indirizzi di memoria e dell'insieme `FunDecl` delle definizioni di funzione e della denotazione speciale `Undef`, che rappresenta una situazione di errore.

Nel seguito useremo φ, φ', ecc. per denotare frame e $\sigma, \sigma', ecc.$ per denotare ambienti e μ, μ'per denotare memorie.

4.3.1 La memoria

La memoria di un calcolatore è una sequenza di locazioni, che contengono valori, identificate da un indirizzo. La memoria può pertanto essere rappresentata da un frame, ovvero da una funzione μ: `Loc` $\mapsto$ `Val`, che rappresenta per ogni locazione utilizzata il valore che vi è contenuto. L'insieme delle memorie è di seguito denotato `M`.

4.3.2 L'ambiente

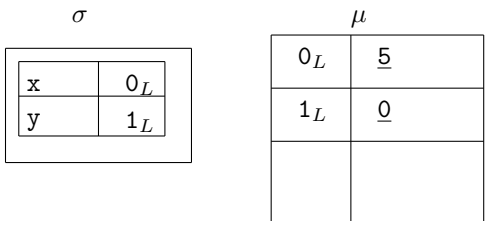
L'ambiente è una *pila* (o *stack* in inglese) di frame. Una pila è una sequenza i cui elementi sono inseriti e acceduti utilizzando una politica di accesso di tipo l'ultimo inserito è il primo accessibile (ovvero *fifo* first in first out). Gli stack che sono una struttura dati molto usata nell'informatica hanno un insieme abbastanza standardizzato di operazioni:

- `EmptyStack` o `nil` o Ω per denotare la pila vuota , ovvero la pila senza elementi.
- `top` restituisce l'elemento in testa alla pila ovvero l'ultimo elemento inserito. Non modifica la pila.
- `pop` restituisce l'elemento in testa alla pila. Modifica la pila, eliminando l'elemento in testa.

- `push` inserisce un elemento in testa alla pila. Nel seguito la push è denotata con il .

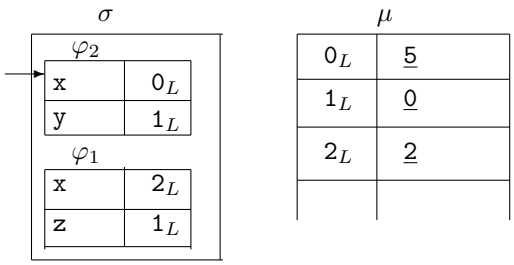
I frame, elementi dello stack (cioè dell'ambiente), sono diversi dai frame che costituivano lo stato nella sezione precedente, poichè sono funzioni φ: `Ide` $\mapsto$ `Den`

Nel nuovo stato le variabili sono rappresentate da ambiente e memoria nel seguente modo: per ogni variabile cè un legame nell'ambiente che associa all'identificatore l'indirizzo della locazione di memoria in cui è memorizzato il valore di tale identificatore. Si consideri la rappresentazione grafica di uno stato $\langle \sigma, \mu \rangle$ in cui l'ambiente è costituito da un stack contenente un unico frame φ e la memoria μ.



Lo stato raffigurato graficamente contiene 2 variabili la prima ha nome `x`, valore $\underline{5}$ ed è memorizzata nella locazione di indirizzo 0_L, la seconda ha nome `y`, valore $\underline{0}$ ed è memorizzata nella locazione di indirizzo 1_L. La ragione di questa *indirezione* e la presenza di 2 componenti: stack e memoria, è legata alla necessità di rappresentare le locazioni di memoria che, come vedremo più avanti, sono valori in **C**.

La necessità invece di strutturare l'ambiente in più frame memorizzati in una pila è legata alla struttura a *blocchi* del linguaggio che permette di definire 2 variabili con lo stesso nome una definita in un blocco più interno rispetto all'altra, questo sarà chiaro più avanti quando vedremo nel dettaglio le caratteristiche del linguaggio. Per il momento è necessario sapere che questo implica la possibilità che in uno stato siano 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 $\underline{5}$ e $\underline{2}$ è il valore di `x` nello stato $< \sigma, \mu >$? È il valore che si trova ricercando un'associazione per `x` a partire dal frame più "recente", ovvero il frame in testa allo stack, contrassegnato dalla freccia.



Essendo l'ambiente una *pila* di frame la locazione associata ad un identificatore è quello che si trova ricercandolo nei frame dell'ambiente dall'alto verso il basso. Di conseguenza il valore dell'identificatore `x` è $\underline{5}$: il frame φ_2 è stato infatti impilato sul frame φ_1 e la locazione individuata per `x` è 0_L. Si noti infine che questa rappresentazione delle variabili permetterebbe di rappresentare anche la condivisione dei valori. La variabile `z` e la variabile `y` nell'esempio condividono infatti il valore essendo entrambe associate (*legate*, *bound* in inglese) alla stessa locazione di memoria. Tale possibilità non è però sfruttata nella semantica del `C` definita nel seguito, che non permette in realtà di costruire lo stato rappresentato sopra. Esistono però moltri altri linguaggi di programmazione che utilizzano stati siffatti.

Nel seguito, indicheremo con Φ l'insieme dei possibili frame. Inoltre, se σ è un ambiente e φ un frame, indichiamo con $\varphi.\sigma$ il nuovo ambiente ottenuto aggiungendo il frame φ all'ambiente σ: nel ambiente così ottenuto, φ è il frame *più recente*. Indicando con Ω l'ambiente vuoto (ovvero la pila vuota di frame), l'insieme Σ degli ambienti può allora essere definito induttivamente come segue:

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

Una formalizzazione dell'ambiente σ dell'esempio precedente è la seguente:

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

dove φ_1 e φ_2 sono le funzioni:

$$\varphi_1(i) = \begin{cases} \mathtt{1_L} & \text{se } i{=}\mathtt{z} \\ \mathtt{2_L} & \text{se } i{=}\mathtt{x} \\ \bot & \text{altrimenti} \end{cases} \qquad \varphi_2(i) = \begin{cases} \mathtt{1_L} & \text{se } i{=}\mathtt{y} \\ \mathtt{0_L} & \text{se } i{=}\mathtt{x} \\ \bot & \text{altrimenti} \end{cases}$$

Dato uno stato σ ed un identificatore `i`, continuiamo ad indicare con $\sigma(\mathtt{i})$ il valore associato ad `i` in σ[7]. Avremo allora:

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

Consideriamo ancora l'esempio precedente, con $\sigma = \varphi_2.\varphi_1.\Omega$, e la memoria μ sopra definita. Abbiamo: $\sigma(\mathtt{x}) = \mathtt{0_L}$ poiché $\varphi_2(\mathtt{x}) = \mathtt{0_L}$. Inoltre, $\sigma(\mathtt{z}) = \mathtt{1_L}$ poiché $\varphi_2(\mathtt{z}) = \bot$ e $\varphi_1(\mathtt{z}) = \mathtt{1_L}$. Inoltre, poichè $\mu(\mathtt{0_L}) = \underline{5}$ anche $\mu(\sigma(\mathtt{x})) = \underline{5}$.

4.3.3 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 oprazioni che modificano lo 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.

Inserzione di un legame in un frame

Introduciamo dapprima una notazione per l'inserzione di un legame in un frame. Siano allora φ un frame, `x` un elemento del dominio, ovvero di `Ide`, e `d` un elemento del codominio, ovvero di `Den`. Denotiamo con $\varphi[^{\mathtt{d}}/_{\mathtt{x}}]$ un frame in cui all'identificatore `x` è associato la denotazione `d`, mentre ad ogni altro identificatore $\mathtt{y} \in \mathtt{Ide}$ (diverso da `x`) è associata la stessa denotazione associato ad `y` in φ. Ricordando che un frame φ non è altro che una funzione $\varphi : \mathtt{Ide} \mapsto \mathtt{Den}$, quanto appena detto può essere espresso formalmente in questo modo:

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

Si noti che $\varphi[^{\mathtt{d}}/_{\mathtt{x}}]$ è ancora un frame, ovvero una funzione da `Ide` in `Den`. Si noti inoltre che l'operazione così definita inserisce un legame per l'identificatore `x`, che risulta ora associato alla denotazione `d`. La definizione così data risulta corretta anche nell'eventualità che un legame per `x` esista ed in questo caso tale legame risulterà modificato. Tale eventualità non si dovrebbe verificare per i frame dell'ambiente, in quanto questo darebbe luogo ad un errore statico di ridefinizione di un identificatore già definito. La semantica data di seguito assume la correttezza sintattica e di tipo e di ogni altro errore rilevabile staticamente.

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

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

Modifica della memoria

Anche la memoria è un frame cioè una funzione $\mu : \texttt{Loc} \mapsto \texttt{Val}$, e l'operazione di modifica ha la stessa notazione sintattica della modifica di frame, ma una diversa semantica. Infatti, le locazioni modificabili con una operazione di modifica della memoria sono solo quelle che risultano definite nella memoria che si vuole modificare. Pertanto, nel caso della memoria, l'operazione di modifica non effettua l'inserzione del legame per l nel caso non ne esista alcuno ma in questo ultimo caso, lascia la memoria invariata. Formalmente:

$$\mu[^{\texttt{v}}/_{l}](l_1) = \begin{cases} \mu(l_1) & \text{se } l_1 \neq l \\ \texttt{v} & \text{se } l_1 = l \wedge \mu(l) \neq \bot \\ \bot & \text{se } l_1 = l \wedge \mu(l) = \bot \end{cases}$$

Analogomente a quanto osservato per i frame, $\mu(l) = \texttt{v}$ stabilisce semplicemente che il valore associato ad l nella memoria μ è $\texttt{v}$, mentre è *ignoto* (e non necessariamente *indefinito*) il valore associato ad ogni altra locazione che non sia l. Ancora, la scrittura $\mu(l) = v$ non equivale a dire che la memoria μ contiene una sola locazione l: non corrisponde cioè a stabilire che $\mu = \{l \mapsto \texttt{v}\}$. Infatti, la scrittura $\{l \mapsto \texttt{v}\}$ indica una memoria in cui solo la locazione l è allocata e contiene il valore $\texttt{v}$, mentre ogni altra locazione risulta libera. Inoltre, entrambe le scritture precedenti non equivalgono a $\mu[^{\texttt{v}}/_{l}]$, in quanto quest'ultima sta ad indicare una memoria che associa ad l il valore $\texttt{v}$ e si comporta come μ, qualunque esso sia, in corrispondenza di ogni altro identificatore. Infine notiamo che $\mu[^{\texttt{v}}/_{l}]$ rappresenta se una memoria diversa da μ nel caso in cui risulti verificata la condizione: $\mu(l) \neq \bot \wedge \mu(l) \neq \texttt{v}$.

Allocazione di memoria

L'inserzione di nuovi legami nella memoria corrisponde alla allocazione di memoria su una machina, e differisce dall'inserzione di un nuovo legame in un generico frame perchè nel caso della memoria non vengono forniti parametri (cioè un elemento del dominio e un elemento del codominio) eccetto la memoria stessa. Tale operazione è di seguito chiamata *free* ed ha il seguente tipo: $\texttt{M} \mapsto \texttt{Loc} \times \texttt{M}$.

$$free(\mu) = \langle l, \mu_1 \rangle$$

Con $\mu_1(l_1) = \varpi \ \wedge \ \mu(l) = \bot \ \wedge \ \forall l_1 \neq l, \mu_1(l_1) = \mu(l_1)$ L'operazione *free* calcola una locazione e una memoria, la locazione l è una locazione che risultava libera nella memoria originaria μ, inoltre la nuova memoria μ_1 contiene le stesse locazioni di μ con gli stessi valori ed inoltre la locazione l in cui è ora memorizzato lo speciale valore indefinito ϖ.

Esercizi

1. Dire quali sono le locazioni associate agli identificatori $\texttt{x}$ ed $\texttt{y}$ nel frame φ in tutti i casi seguenti:
   (a) $\varphi = \{\texttt{x} \mapsto 0_L, \texttt{y} \mapsto 1_L\}$
   (b) $\varphi = \{\texttt{x} \mapsto 0_L, \texttt{z} \mapsto 1_L\}$
   (c) $\varphi = \omega[^{0_L}/_{\texttt{y}}]$
   (d) $\varphi = \omega[^{0_L}/_{\texttt{y}}][^{1_L}/_{\texttt{x}}]$
   (e) $\varphi = (\omega[^{0_L}/_{\texttt{y}}][^{1_L}/_{\texttt{x}}])[^{2_L}/_{\texttt{x}}]$

2. Data la memoria $\mu = \{0_L \mapsto \underline{4}, 1_L \mapsto \underline{0}, 2_L \mapsto \underline{10}\}$, dire quali sono i valori associati agli identificatori $\texttt{x}$, $\texttt{y}$ e $\texttt{z}$, ovvero cosa valgono $\mu(\sigma(x))$, $\mu(\sigma(y))$ e $\mu(\sigma(z))$, nello stato $\langle \sigma, \mu \rangle$ in tutti i casi seguenti, assumendo φ_1 il frame definito per il caso a) dell'esercizio precedente e φ_2 il frame per il caso b):
   (a) $\sigma = \varphi_1.\varphi_2.\Omega$
   (b) $\sigma = \varphi_2.\varphi_1.\Omega$

5 La semantica operazionale del nucleo di C

A partire da questo paragrafo affronteremo lo studio della semantica in stile operazionale di un frammento di linguaggio di programmazione il **C**. Ci limiteremo a considerare i tipi di dato elementari `int` e, 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 ma i costrutti per manipolarli consentono di definire espressioni per valori interi e valori logici (o booleani). Questi ultimi sono i valori di verità nell'insieme $\mathbb{B} = \{tt, ff\}$ che nei linguaggi di programmazione sono rappresentati sintatticamente con i simboli `true` e `false` rispettivamente. In **C**, i valori booleani (`true` e `false`) non esistono ma vengono utilizzati i valori interi, in particolare $\underline{0}$ rappresenta `false` e ogni altro intero rapresenta `true`. Di seguito daremo la semantica delle espressioni come se i valori booleani esistessero, come valori calcolati ma non come espressioni. Per quanto riguarda gli operatori per definire espressioni (le *proposizioni*) il cui valore è un valore di verità (ad esempio `5>3`) e per combinare tali espressioni, il **C**, come ogni linguaggio, mette a disposizione gli operatori `&&`, `||` e `!` il cui significato è l' operatore logico $\wedge, \vee$ e $\neg$ rispettivamente [8]. La sintassi delle espressioni è la seguente:

Exp ::= ConstVal | Ide | (Exp) | Exp Op Exp |! Exp
Op ::= +| − | ∗ |/|%|==|! =| < |<=| > |>=|&&|||
ConstVal ::= ...

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

- $\mathtt{v}, \mathtt{v}', \ldots$ denotano valori in $\mathcal{I}nt$, 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 per valori naturali, nel caso in cui lo stato era un solo frame, in questo caso però lo stato è una coppia $\langle \sigma, \mu \rangle$ ambiente, memoria, ma la regola semantica si complica notevolmente per tenere conto della possibilità che le espressioni modifichino lo stato, eventualità che non si verifica per le espressioni la cui sintassi è definita sopra, ma si verificherà per le espressioni che vedremo nell'ultimo paragrafo. Per tenere conto di questa eventualità dobbiamo prevedere che la semantica delle espressioni restituisca una coppia valore, stato. In effetti però la componente dello stato che può essere modificata è la sola memoria, di conseguenza la valutazione di un'espressione calcola una coppia valore, memoria , ovvero $\langle \mathtt{v}, \mu \rangle$.

Riassumendo , abbiamo le seguenti configurazioni per $\mathtt{S}_{exp}$:

$$\begin{aligned} \Gamma &= \{\langle \mathtt{E}, \langle \sigma, \mu \rangle\rangle \mid \mathtt{E} \in \mathtt{Exp} \wedge \sigma \in \Sigma \wedge \mu \in \mathtt{M}\} \cup \\ &\quad \{\langle \underline{\mathtt{v}}, \langle \sigma, \mu \rangle\rangle \mid \underline{\mathtt{v}} \in \mathtt{Val} \wedge \sigma \in \Sigma, \wedge \mu \in \mathtt{M}\} \\ \mathtt{T} &= \{\langle \underline{\mathtt{v}}, \langle \sigma, \mu \rangle\rangle \mid \underline{\mathtt{v}} \in \mathtt{Val} \wedge \sigma \in \Sigma, \wedge \mu \in \mathtt{M}\} \end{aligned}$$

mentre le regole possono essere riassunte nelle due seguenti regole:

[8] Rammentiamo che, se b, b' sono elementi di $\mathbb{B}$, $b \wedge b'$ 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.

$$\frac{\langle \mathtt{E}, \langle \sigma, \mu \rangle \rangle \rightarrow_{exp} \langle \mathtt{v}, \langle \sigma, \mu' \rangle \rangle \qquad \langle \mathtt{E}', \langle \sigma, \mu' \rangle \rangle \rightarrow_{exp} \langle \mathtt{v}', \langle \sigma, \mu'' \rangle \rangle \qquad \mathtt{v}'' = \mathtt{v}\ [|\mathrm{op}|]\ \mathtt{v}'}{\langle \mathtt{E}\ \mathrm{op}\ \mathtt{E}', \langle \sigma, \mu \rangle \rangle \rightarrow_{exp} \langle \mathtt{v}'', \langle \sigma, \mu'' \rangle \rangle} \qquad (exp_{\mathrm{op}})$$

L'equazione semantica definita stabilisce che la sottoespressione `E` viene valutata per prima infatti essa è valutata nello stato di $\langle \sigma, \mu \rangle$ di valutazione dell'intera espressione $\langle \mathtt{E}\ \mathrm{op}\ \mathtt{E}', \langle \sigma, \mu \rangle \rangle$. Essa produce una coppia $\langle \mathtt{v}, \mu' \rangle$. La memoria calcolata μ' è utilizzata per costruire lo stato di valutazione della sottoespressione `E'`, che calcola la coppia $\langle \mathtt{v}', \mu'' \rangle$. La memoria μ'' calcolata è la memoria secondo elemento della coppia risultato della valutazione dell'intera sottoespressione, mentre il valore $\mathtt{v}'$ è usato per calcolare il valore finale $\mathtt{v}''$. Come nel caso delle regole semantiche del paragrafo precedente utilizziamo delle premesse che corrispondono ad operazioni "elementari" tra valori che si suppone di saper calcolare: $[|\mathrm{op}|]$ sta per la funzione semantica che permette di calcolare l'operatore op in questo modo la regola definita vale per tutti gli operatori binari del linguaggio siano essi aritmetici $(+, -, *, /, \%$, logici $\&\&, ||$ o di relazione $=, \neq, >, <, >=, <=$.

La regola seguente definisce la semantica delle espressioni di operatori unari, analoga a quella di espressioni con operatori binari.

$$\frac{\langle \mathtt{E}, \langle \sigma, \mu \rangle \rangle \rightarrow_{exp} \langle \mathtt{v}, \mu' \rangle \qquad \mathtt{v}' = \neg \mathtt{v}}{\langle \mathrm{uop}\ \mathtt{E}, \langle \sigma, \mu \rangle \rangle \rightarrow_{exp} \langle \mathtt{v}', \langle \sigma, \mu' \rangle \rangle} \qquad (exp_{\mathrm{uop}})$$

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[9]. 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 **C** 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 ::= {StatList}
StmtList ::= Com | Com StmtList

Le configurazioni del sistema di transizioni per i comandi, $\mathtt{S}_{com}$, sono di due tipi: $\langle \mathtt{C}, \langle \sigma, \mu \rangle \rangle$, dove `C` è un comando e $\langle \sigma, \mu \rangle$ uno stato, sono le configurazioni che corrispondono agli stadi iniziali del calcolo (cfr.

[9] 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 configurazioni $\langle \mathtt{E}, \langle \sigma, \mu \rangle \rangle$ nel sistema $\mathtt{S}_{exp}$), mentre le configurazioni composte da una sola memoria μ sono le configurazioni terminali, che corrispondono alla conclusione del calcolo. Riassumendo:

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

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

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

Il comando di assegnamento è il costrutto del linguaggio che consente di apportare modifiche allo stato. Si noti l'utilizzo della notazione $\mu[{}^{\mathtt{v}}/_{l}]$ introdotta nella Sezione **??** 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 o dichiarazioni da eseguire nell'ordine in cui essi compaiono nella sequenza. In questa prima fase della definizione della semantica, in cui le dichiarazioni non sono ancora state introdotte, la semantica del blocco è una semplice parentesi. Nel paragrafo successivo in cui vengono introdotte le dichiarazioni, verrà definita la corretta semantica del blocco.

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

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

La regola $(com_{Stmt-list})$ 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 $\langle \sigma, \mu \rangle$, con $\sigma = \{\mathtt{x} \mapsto 0_L, \mathtt{y} \mapsto 1_L\}.\Omega$ e $\mu = \{0_L \mapsto \underline{4}, 1_L \mapsto \underline{0}\}$. Abbiamo la derivazione:

$$\begin{aligned} & \langle \mathtt{x} = 2; \mathtt{y} = 3; \mathtt{x} = \mathtt{x} - 1;, \langle \sigma, \mu \rangle \rangle \\ \rightarrow_{com} \quad & \{\ (com_{Stmt-list}), \\ & \quad (\mathrm{d1}):\ \langle \mathtt{x} = 2;, \langle \sigma, \mu \rangle \rangle \rightarrow_{com} \langle \sigma, \mu[{}^{\underline{2}}/_{0_L}] \rangle \\ & \qquad\quad \mu_1 = \mu[{}^{\underline{2}}/_{0_L}], \\ & \quad (\mathrm{d2}):\ \langle \mathtt{y} = 3; \mathtt{x} = \mathtt{x} - 1;, \langle \sigma, \mu_1 \rangle \rangle \rightarrow_{com} \langle \sigma, (\mu_1[{}^{\underline{1}}/_{0_L}])[{}^{\underline{3}}/_{1_L}] \rangle \\ & \qquad\quad (\mu_1[{}^{\underline{1}}/_{0_L}])[{}^{\underline{3}}/_{1_L}] = \{0_L \mapsto \underline{1}, 1_L \mapsto \underline{3}\}. \end{aligned}$$

dove (d1) è la derivazione:

(d1):

$\langle \mathtt{x} = \mathtt{2};, \langle \sigma, \mu \rangle \rangle$

$\rightarrow_{com}$ { $(com_{=})$,
$\langle \mathtt{2}, \langle \sigma, \mu \rangle \rangle = \langle \underline{2}, \langle \sigma, \mu \rangle \rangle$
$0_L = \sigma(\mathtt{x})$
}

$\langle \sigma, \mu_1 \rangle$
$\mu_1 = \{0_L \mapsto \underline{2}, 1_L \mapsto \underline{0}\}$

e (d2) è la derivazione:

(d2):

$\langle \mathtt{y} = \mathtt{3}; \mathtt{x} = \mathtt{x} - \mathtt{1};, \langle \sigma, \mu_1 \rangle \rangle$

$\rightarrow_{com}$ { $(com_{Stmt-list})$,
(d3): $\langle \mathtt{y} = \mathtt{3};, \langle \sigma, \mu_1 \rangle \rangle \rightarrow_{com} \langle \sigma, \mu' \rangle$
$\mu' = \mu_1[^{\underline{3}}/_{1_L}]$,
(d4): $\langle \mathtt{x} = \mathtt{x} - \mathtt{1};, \langle \sigma, \mu' \rangle \rangle \rightarrow_{com} \langle \sigma, \mu'[^{\underline{1}}/_{0_L}] \rangle$,
$\mu'[^{\underline{1}}/_{0_L}] = (\mu[^{\underline{1}}/_{0_L}])[^{\underline{3}}/_{1_L}]$}

$\langle \sigma, \mu'' \rangle$
$\mu'' = \{0_L \mapsto \underline{1}, 1_L \mapsto \underline{3}\}$

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

(d3):

$\langle \mathtt{y} = \mathtt{3};, \langle \sigma, \mu_1 \rangle \rangle$

$\rightarrow_{com}$ { $(com_{=})$,
$\langle \mathtt{3}, \langle \sigma, \mu_1 \rangle \rangle = \langle \underline{3}, \langle \sigma, \mu_1 \rangle \rangle$
$1_L = \sigma(\mathtt{y})$
}

$\langle \sigma, \mu' \rangle$
$\mu' = \{0_L \mapsto \underline{2}, 1_L \mapsto \underline{3}\}$

(d4):

$\langle \mathtt{x} = \mathtt{x} - \mathtt{1};, \langle \sigma, \mu' \rangle \rangle$

$\rightarrow_{com}$ { $(com_{=})$,
$\langle \mathtt{x} - \mathtt{1}, \langle \sigma, \mu' \rangle \rangle = \langle \underline{1}, \langle \sigma, \mu' \rangle \rangle$
$0_L = \sigma(\mathtt{x})$
}

$\langle \sigma, \mu' \rangle$
$\mu' = \{0_L \mapsto \underline{1}, 1_L \mapsto \underline{3}\}$

■

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 $\mu' = \mu[\ldots]$ 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.

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

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

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

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

$$\frac{\langle \texttt{E}, \langle \sigma, \mu \rangle \rangle \rightarrow_{exp} \langle ff, \langle \sigma, \mu'' \rangle \rangle}{\langle \texttt{while (E) C}, \langle \sigma, \mu \rangle \rangle \rightarrow_{com} \langle \sigma, \mu'' \rangle} \quad (com_{while-ff})$$

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

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 5.2 Vediamo come esempio la semantica del comando `while (x>0) x=x-1;` a partire dallo stato $\langle \sigma, \mu \rangle$. $\sigma = \{\texttt{x} \mapsto 0_L\}.\Omega$ e $\mu = \{0_L \mapsto \underline{2}\}$

$\langle$`while (x>0) x=x-1;`, $\langle \sigma, \mu \rangle\rangle$

$\rightarrow_{com}$ $\{(com_{while-tt})$:
$\langle \texttt{x} > 0, \langle \sigma, \mu \rangle\rangle = \langle tt, \mu \rangle$
$(com_{=})$: $\langle$ `x-1`, $\langle \sigma, \mu \rangle\rangle = \langle \underline{1}, \langle \sigma, \mu \rangle\rangle$
$0_L = \sigma(x)$
$\langle \texttt{x} = \texttt{x} - 1;, \langle \sigma, \mu \rangle\rangle \rightarrow_{com} \langle \sigma, \mu' \rangle$ $\mu' = \{0_L \mapsto \underline{1}\}$
(d1): $\langle$ `while (x>0) x=x-1;`, $\langle \sigma, \mu' \rangle\rangle \rightarrow_{com} \langle \sigma, \mu'' \rangle$
$\mu'' = \{0_L \mapsto \underline{0}\}$
μ''

(d1):

$\langle$`while (x>0) x=x-1;`, $\langle \sigma, \mu' \rangle\rangle$

$\rightarrow_{com}$ $\{$ $(com_{while-tt})$:
$\langle \texttt{x} > 0, \langle \sigma, \mu' \rangle\rangle = \langle tt, \langle \sigma, \mu' \rangle\rangle$
$(com_{=})$: $\langle$`x-1`, $\langle \sigma, \mu' \rangle\rangle = \langle \underline{0}, \langle \sigma, \mu' \rangle\rangle$
$0_L = \sigma(x)$
$\langle$`x=x-1;`,$\langle \sigma, \mu' \rangle\rangle \rightarrow_{com} \langle \sigma, \mu'' \rangle \mu'' = \{0_L \mapsto \underline{0}\}$
(d2): $\langle$ `while (x>0) x=x-1;`, $\langle \sigma, \mu'' \rangle\rangle$

$\langle \sigma, \mu'' \rangle$

(d2):

$\langle$`while (x>0) x=x-1;`, $\langle\sigma, \mu''\rangle\rangle$

$\rightarrow_{com}$ { $(com_{while-ff})$:
$\langle$`x > 0`, $\langle\sigma, \mu''\rangle\rangle = \langle ff, \mu''\rangle$ }

$\mu'' = \{0_L \mapsto \underline{0}\}$

■

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 $\langle \mathtt{x = y/0;}, \langle\sigma, \mu\rangle\rangle$: 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 $\underline{v} \in \mathcal{I}nt$ per cui $\mathtt{y/0} = \underline{v}$). 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 $\langle \mathtt{x \ / \ y}, \langle\sigma, \mu'\rangle\rangle$ è bloccata se, nello stato $\langle\sigma, \mu'\rangle$, il valore associato ad `y` (ovvero $\mu'(\sigma(\mathtt{x}))$) 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 $\langle$`while` $(\mathtt{x > 0})\ \mathtt{x = x + 1;}, \langle\sigma, \mu'\rangle\rangle$, nel caso in cui $\mu'(\sigma(\mathtt{x})) > \underline{0}$. L'unica regola del sistema $\mathtt{S}_{com}$ potenzialmente applicabile è chiaramente $(com_{while-tt})$, dal momento che $\langle \mathtt{x > 0}, \langle\sigma, \mu'\rangle\rangle = \langle tt, \langle\sigma, \mu'\rangle\rangle$. L'applicazione della regola $(com_{while-tt})$ richiede però, tra le premesse, di determinare una transizione del tipo $\langle$`while` $(\mathtt{x > 0})\ \mathtt{x = x + 1;}, \langle\sigma, \mu''\rangle\rangle$. Con $\mu'' = \mu'[^{(\mu'(\sigma(x)))+1}/_{\sigma(x)}]$. Anche in questo caso l'unica regola potenzialmente applicabile è $(com_{while-tt})$ (essendo $\langle \mathtt{x > 0}, \langle\sigma, \mu''\rangle\rangle = \langle tt, \langle\sigma, \mu''\rangle\rangle$, ed ancora una delle sue premesse corrisponde a determinare una transizione del tipo $\langle$`while` $(\mathtt{x > 0})\ \mathtt{x = x + 1;}, \langle\sigma, \mu'''\rangle\rangle$. Con $\mu''' = \mu''[^{(\mu''(\sigma(x)))+1}/_{\sigma(x)}]$. È facile convincersi che questo procedimento non ha mai termine (dal momento che non esiste una memoria μ intermedia del calcolo che corrisponde ad una situazione in cui la guardia del comando ha valore $\langle ff, \langle\sigma, \mu\rangle\rangle$). 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 $\langle\sigma, \mu\rangle$ nei seguenti casi:

   (a) $\sigma = \{\mathtt{x} \mapsto 0_L, \mathtt{y} \mapsto 1_L\}.\Omega$, $\mu = \{0_L \mapsto \underline{10}, 1_L \mapsto \underline{5}\}$

   (b) σ come sopra ma $\mu = \{0_L \mapsto \underline{5}, 1_L \mapsto \underline{5}\}$

2. Valutare il blocco

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

   a partire dallo stato $\langle\sigma, \mu\rangle$ $\sigma = \{\mathtt{z} \mapsto 2_L\}.\{\mathtt{x} \mapsto 0_L, \mathtt{y} \mapsto 1_L\}.\Omega$, $\mu = \{0_L \mapsto \underline{10}, 1_L \mapsto \underline{5}, 2_L \mapsto \underline{0}\}$

3. Valutare il blocco

   {`z= x/y; w=x%y;`
   `if (w==0) z=0; else z=z+1;`}

   a partire dallo stato $\langle\sigma, \mu\rangle$ con $\sigma = \{\mathtt{z} \mapsto 2_L, \mathtt{w} \mapsto 3_L\}.\{\mathtt{x} \mapsto 0_L, \mathtt{y} \mapsto 1_L\}.\Omega$, e per μ nei seguenti casi:

(a) $\mu = \{0_L \mapsto \underline{100}, 1_L \mapsto \underline{10}, 2_L \mapsto \underline{0}, 3_L \mapsto \underline{0}\}$

(b) $\mu = \{0_L \mapsto \underline{15}, 1_L \mapsto \underline{2}, 2_L \mapsto \underline{10}, 3_L \mapsto \underline{1}\}$

4. Valutare il blocco

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

a partire dallo stato $\langle \sigma, \mu \rangle$, nei seguenti casi:

(a) $\sigma = \{\mathtt{x} \mapsto 0_L, \mathtt{z} \mapsto 1_L\}.\Omega$, $\mu = \{0_L \mapsto \underline{4}, 1_L \mapsto \underline{5}\}$

(b) $\mu(\sigma(\mathtt{x})) = \underline{0}$ e $\mu(\sigma(\mathtt{z})) \neq \bot$

5. Valutare il blocco

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

a partire da uno stato $\langle \sigma, \mu \rangle$ nei seguenti casi:

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

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

5.3 Le dichiarazioni e la loro semantica

Nelle sezioni precedenti abbiamo introdotto lo stato costituito da due componenti, ambiente e memoria, 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. Le costanti nelle prime versioni del `C` vengono definite attraverso la direttiva `#define` e trattate in fase di compilazione. 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`, che corrispondono all'insieme dei numeri interi.

La sintassi di una dichiarazione è data dalle seguenti produzioni:

Decl ::=| Type Ide;
| Type Ide = Exp;

Type ::= int

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 ::= {StmtList }

StmtList ::=Stmt | Stmt StmtList

Stmt ::= 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[10]. 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.

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

```
{
    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:

i) nel frame *corrente* (quello in testa allo stack) un'associazione per la nuova variabile dichiarata, che indica la locazione di memoria che conterrà il valore della variabile,

ii) allocare nella memoria una nuova lcazione che deve contenere il valore della variabile.

Le configurazioni non terminali di $\mathtt{S}_{dec}$ sono del tipo $\langle \mathtt{D}, \langle \sigma, \mu \rangle \rangle$, dove `D` è un dichiarazione e $\langle \sigma, \mu \rangle$ uno stato. Formalmente:

$$\begin{aligned} \Gamma_{dec} &= \{\langle \mathtt{D}, \langle \sigma, \mu \rangle \rangle \mid \mathtt{D} \in \mathtt{Dec}, \quad \sigma \in \Sigma, \mu \in \mathrm{M}\} \cup \\ &\quad \{\langle \sigma, \mu \rangle \mid \sigma \in \Sigma, \mu \in \mathrm{M}\} \\ \\ \mathtt{T}_{dec} &= \{\langle \sigma, \mu \rangle \mid \sigma \in \Sigma, \mu \in \mathrm{M}\} \end{aligned}$$

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[11].

Inoltre, per ciascun tipo elementare `T` (nel nostro caso `int`) una variabile può essere dichiarata senza inizializzarne il valore (ad esempio `int x;`) oppure inizializzandone il valore a quello di una espressione (ad esempio `int i=0;`). Nel primo caso, il valore associato alla variabile non è definito, e quindi usiamo il valore *simbolico* ϖ definito proprio per queste situazioni.

[11] 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.

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

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

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 ($com_{Stmt-List}$), 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 `Stmt_list`) vengono trattate attraverso un opportuno sottosistema, le cui transizioni sono date di seguito.

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

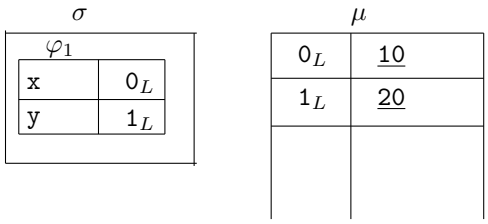
$$\frac{\langle \mathtt{S}, \langle \sigma, \mu \rangle \rangle \rightarrow_{slist} \langle \sigma'', \mu'' \rangle \qquad \langle \mathtt{Slist}, \langle \sigma'', \mu'' \rangle \rangle \rightarrow_{slist} \langle \sigma', \mu' \rangle}{\langle \mathtt{S\ \ Slist}, \langle \sigma, \mu \rangle \rangle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

La regola (com_{Block}) formalizza il fatto che un blocco viene eseguito in uno stato nel quale è stato inserito nello stack 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. Si noti che necessariamente $\sigma' = \sigma$ cioè lo stack dello stato risultante è uguale allo dello stato iniziale di valutazione del blocco. Infatti, le uniche regole che modificano lo stack sono le quelle che definiscono la semantica delle dichiarazioni che però modificano solo il frame in testa allo stack, che alla fine dell'esecuzione del blocco viene rimosso.

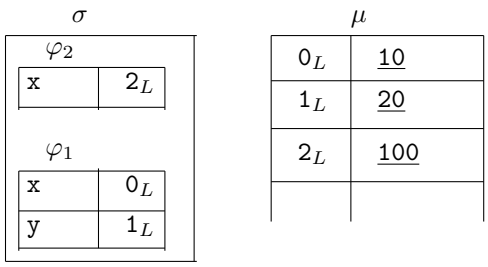
Vediamo un esempio di programma a blocchi che consente anche di trattare il caso di blocchi annidati in cui uno stesso identificatore viene ridefinito (purché non nello stesso blocco). Questa possibilità, pur se nella tradizione dei linguaggi imperativi quali Algol, Pascal e C, non viene contemplata 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. Mostreremo gli stati calcolati usando la rappresentazione grafica come nel paragrafo 4.3

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

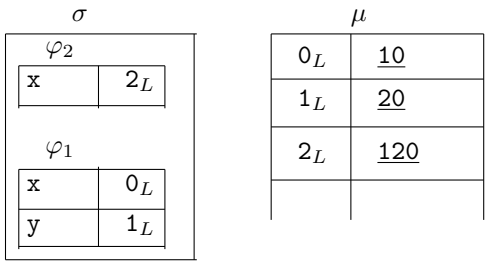
Supponiamo che tale blocco venga eseguito a partire dallo stato iniziale, $\langle \sigma, \mu \rangle$ con $\sigma = \omega.\Omega$ e $\mu = \{\}$. Lo stato al punto (1), al momento cioè dell'ingresso nel blocco annidato, è il seguente:



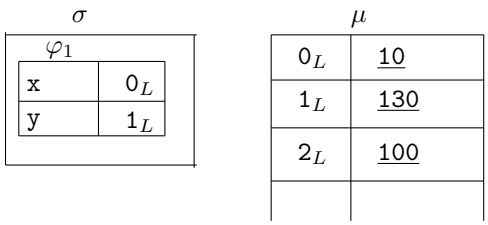
Dopo l'esecuzione della dichiarazione (2), lo stato si presenta come segue:



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:



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:



La locazione 2_L nella memoria resta allocata e contiene sempre il valore 100 ma non è più raggiungibile da nessuna variabile dello stack. Nei sistemi reali le locazioni di memoria vengono recuperate per essere successivamente riutilizzate, con un'operazione chiamata *garbage collector*.

Esercizi

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

   (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$\langle\sigma,\mu\rangle$ con $\sigma = \omega.\Omega$ e $\mu = \{\}$, 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 $\langle\sigma,\mu\rangle$ con $\sigma = \{\mathtt{x} \mapsto 0_L, \mathtt{y} \mapsto 1_L\}.\Omega$ e $\mu = \{0_L \mapsto \underline{1}0, 1_L \mapsto \underline{1}0\}$:

   (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;}`

5.4 Puntatori

Una delle caratteristiche del **C** è quella di includere gli indirizzi di memoria tra i valori. Questo vuol dire che il programmatore **C** può direttamente manipolare (accedere e modificare) i contenuti delle locazioni di memoria specificandone l'indirizzo. Questa è una caratteristica a *basso livello* ereditata dai linguaggi assembler che consente di esprimere operazioni che diversamente avrebbero richiesto l'introduzione nel linguaggio di meccanismi abbastanza sofisticati (e costosi dal punto di vista dell'implementazione). Tipicamente, meccanismi per manipolare i valori dei dati strutturati con dimensioni variabili, *allocazione dinamica* della memoria. Introducendo i puntatori, invece, e rendendo nota o affidando al programmatore l'implementazione dei dati l'onere della gestione dei dati è delegato al programmatore. Questo possibilità che da un lato sembra dare al programmatore maggiore *potenza* di calcolo, è da un altro lato molto pericolosa perchè gli consente di manipolare anche in modo errato i dati, anche distruggendoli, e andando incontro a errori molto difficilmente individuabili. Infatti nei linguaggi, anche procedurali, ma di successive generazioni (ad esempio Java) i puntatori non esistono. Anche le successive versioni di **C** hanno aggiunto meccanismi per la manipolazione dei dati strutturati che permettono di evitare l'uso dei puntatori. Dal punto di vista semantico i puntatori sono interessanti perchè risultano semplici, dal momento che sono a *basso livello* ma conoscerne la semantica, come per qualunque costrutto, aiuta a comprendere i programmi e i programmi che utilizzano puntatori sono tutt'altro che semplici.

In **C** le variabili che contengono valori che sono indirizzi di memoria devono essere dichiarate specificando il tipo del valore contenuto nella locazioni il cui indirizzo è il valore della variabile dichiarata seguito da un `*`. I costrutti che in **C** consentono di manipolare locazioni sono due:

- `&` che calcola una locazione e si applica ad un sottoinsieme delle espressioni, nel linguaggio didattico questo sottoinsime è ristretto ai soli identificatori. La locazione calcolata, in questo caso è la locazione in cui è memorizzato il valore dell'identificatore.

- `*` che si applica ad una qualunque espressione, la calcola ottenendo un valore `v` che viene però trasformato in una locazione l. Il valore finale calcolato è il valore `v'` contenuto nella locazione l.

Si consideri il seguente esempio che definisce tre variabili `i, j` e `ptri i` and `ptri` sono inizializzate `i` ha valore `1` e `ptri` ha valore l'indirizzo di `i, j` invece non è inizializzata nella dichiarazione, ma il suo valore è definito con un assegnamento risultante della somma della costante `6` e con il *deferenziato* di `ptri`, quest'ultimo è proprio il valore di `i`.

```
int main(){
   int i=1;                          (1)
   int j;                               (2)
   int *ptri=&i;                    (3)
   j=*ptri +6;                      (4)
 }
```

La semantica dei puntatori può essere data in due modi: aggiungendo le locazioni al dominio dei valori o definendo dei meccanismi per trasformare locazioni in valori e viceversa. La seconda è possibile ed anzi molto semplice perchè le locazioni sono numeri naturali ed è la soluzione adottata nella semantica presentata in queste note. Pertanto, i domini della semantica rimangono invariati, ma abbiamo due funzioni che trasformano locazioni in valori e viceversa:

- $\nu_{\text{loc}} : \texttt{Loc} \mapsto \texttt{Val}$ che trasforma locazioni in valori, nel modo ovvio, cioè $\nu_{\text{loc}}(0_L) = 0$, $\nu_{\text{loc}}(1_L) = 1$,... ecc.
- $\eta_{\text{loc}} : \texttt{Val} \mapsto \texttt{Loc}$ che trasforma valori in locazioni, nel modo ovvio, cioè $\eta_{\text{loc}}(0) = 0_L$, $\eta_{\text{loc}}(1) = 1_L$,... ecc.

La regole semantica per `&`, che estende le regole delle espressioni, è la seguente.

$$\frac{\sigma(\texttt{ide}) = l \qquad \texttt{v} = \nu_{\text{Loc}}(l)}{\langle \&\texttt{ide}, \langle \sigma, \mu \rangle \rangle \rightarrow_{dec} \langle \texttt{v}, \langle \sigma, \mu \rangle \rangle} \rightarrow_{exp} \&$$

Si noti che per come è definita la semantica, tale operatore non può modificare la memoria. Questa proprietà non è dovuta alla limitazione dell'operando ai soli identificatori ma risulta comunque vera nel linguaggio completo.

La regola semantica per `*`, che estende anch'essa le regole delle espressioni, è mostrata di seguito.

$$\frac{\langle \texttt{Exp}), \langle \sigma, \mu \rangle \rangle \rightarrow_{exp} \langle \texttt{v}, \mu' \rangle \qquad l = \eta_{\text{Loc}}(\texttt{v}) \qquad \texttt{v}' = \mu'(l)}{\langle *\texttt{Exp}, \langle \sigma, \mu \rangle \rangle \rightarrow_{dec} \langle \texttt{v}', \langle \sigma, \mu' \rangle \rangle} \rightarrow_{exp} *$$

Si noti che l'esecuzione dell'operatore `*` modifica in generale la memoria, infatti μ' può essere diverso da μ.

Per chiarire le regole semantiche, calcoliamo gli stati del programma definito sopra.

Lo stato al punto (1):

σ

φ_1	
i	0_L

μ

0_L	$\underline{1}$

Al punto (2), dopo la dichiarazione di `j`, lo stato risulta:

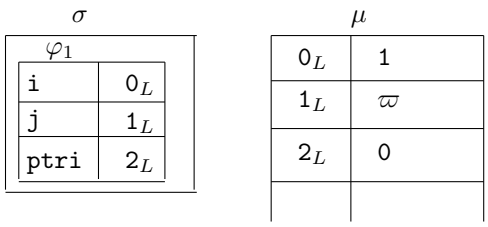
σ

φ_1	
i	0_L
j	1_L

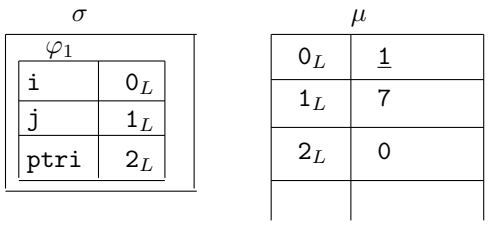
μ

0_L	$\underline{1}$
1_L	ϖ

Al punto (3), dopo la dichiarazione del puntatore `ptri`, lo stato risulta:



Si noti che il valore memorizzato nella locazione 2_L è `0`, in quanto la locazione associata a `i` ha indirizzo 0_L. $\nu_{loc}(\underline{0}_L) = 0$. Al punto (4), dopo la dichiarazione del puntatore `ptri`, lo stato risulta:



I puntatori sarebbero molto poco utili se non fosse possibile modificare il contenuto di una locazione ottenuta attraverso un'operazione di dereferenziazione, in effetti questo è possibile in **C** con il comando di assegnamento la cui sintassi è più complessa di quella presentata fin qui. Pertanto è necessario estendere l'assegnamento consentendo di specificare a sinistra una espressione che contiene un deferenziamento. La sintassi dell'assegnamento è estesa in accordo alla seguente sintassi:

Com ::= Ide = Exp;
*Exp=Exp
...

Le regola semantica per l'assegnamento di un identificatore di variabile rimane invariata mentre la seguente regola viene aggiunta nel caso in cui la parte sinistra dell'assegnamento contenga un'operazione di dereferenziamento.

$$\frac{\langle \mathtt{E}, \langle \sigma, \mu \rangle \rangle \rightarrow_{exp} \langle \mathtt{v}, \langle \sigma, \mu' \rangle \rangle \qquad \langle \mathtt{E}', \langle \sigma, \mu' \rangle \rangle \rightarrow_{exp} \langle \mathtt{v}', \langle \sigma, \mu'' \rangle \rangle \qquad l = \eta_{loc}(\mathtt{v}')}{\langle *\mathtt{E}' = \mathtt{E};, \langle \sigma, \mu \rangle \rangle \rightarrow_{com} \langle \sigma, \mu''[^{\mathtt{v}}/_{l}] \rangle} \qquad (com_{=})$$

Esercizi

1. Si consideri il seguente programma:

```
int main(){
   int i=1;     (1)
   int *ptri=&i;  (2)
   int j=*ptri+i;    (3)
   i=j;   (4)
```

```
 j=2*(*ptri); (5)
}
```

Si disegnino gli stati ai punti (1), (2), (3), (4), (5).

2. Si consideri is seguente programma:

```
int main(){
   int i=3;     (1)
   int *ptri=&i;  (2)
   int j=*ptri+i;    (3)
   *ptri=j;   (4)
   j=2*(*ptri); (5)
  }
```

Si disegnino gli stati ai punti (1), (2), (3), (4), (5).

5.5 Programmazione strutturata

Una metodologia molto utile per scrivere programmi complessi è la metodologia di programmazione per raffinamenti successivi. Tale metodologia può così essere descritta: Ogni volta che occorra esprimere un'operazione complessa, espressa cioè da una complessa struttura di controllo, si supponga di avere a disposizione nuovi operatori (con i quali formare espressioni) e nuovi comandi (con i quali esprimere nuove strutture di controllo), mediante i quali sia possibile realizzare tale operazione con una struttura di controllo più semplice di quella direttamente esprimibile nel linguaggio. La stesura del programma può così procedere speditamente esprimendo via via tutte le operazioni con strutture che fanno riferimento ad applicazioni di questi nuovi operatori (espressioni) e a nuovi comandi, introdotti dall'utente.

Una volta terminata la stesura del programma, si provvede a definire i nuovi comandi ed i nuovi operatori introdotti, con strutture che, a seconda della complessità, fanno riferimento o, ad ancora nuovi, ma più semplici comandi ed operatori, o, a comandi ed operatori del linguaggio. Se il linguaggio di programmazione mette a disposizione dei meccanismi di astrazione sul controllo che permettono di estendere il linguaggio con nuovi comandi o nuovi operatori, definiti dall'utente, allora il programma può essere scritto in accordo a tale metodologia. In questo modo il programma risulterà più semplice da scrivere, più compatto e comprensibile, più semplice da verificare e correggere. Un altro aspetto rilevante nella stesura di programmi è il seguente. Molto spesso programmi, con scopi diversi, hanno parti in comune o simili, cioè parti che calcolano la stessa funzione. Si osservino ad esempio i programmi di ordinamento degli array che utilizzano il metodo `swap` per scambiare i valori di due elementi di una variabile. I processi di definizione di operazioni complesse in termini di operazioni più semplici (raffinamento), e di generalizzazione di sequenze mediante parametri, sono aspetti diversi di un unico processo di astrazione.

In `C` i meccanismi di astrazione sul controllo, che permettono di utilizzare la metodologia di programmazione per raffinamenti successivi, si chiamano *funzioni*.

5.5.1 Funzioni

Per definire sintassi e semantica delle funzioni è necessario distinguere tra la *dichiarazione* di una funzione e la sua *invocazione*: la dichiarazione avviene una volta per tutte; l'invocazione o chiamata corrisponde all'utilizzo effettivo del metodo, alla esecuzione vera e propria delle azioni ad esso associate. In generale, invocazioni diverse della stessa funzione possono avere effetti diversi poichè la funzione è parametrica, cioè definita astraendo su uno o uno o più valori. Questo è ottenuto definendo i parametri (appunto) cioè nomi a cui sono associati dei tipi e che al momento dell'invocazione sono istanziati con i valori, che

quindi risulteranno in generale diversi in chiamate diverse. Per semplicità, nel nucleo di **C** consideriamo funzioni con un solo parametro.

Le informazioni che devono essere specificate quando si definisce una funzione, che sono poi quelle che sono necessarie al momento della chiamata della funzione sono:

- il tipo del risultato.
- il nome della funzione Ide_f. Utilizzato per l'invocazione.
- il tipo e il nome del parametro (*formale*) Ide_a.
- il *corpo* della funzione. Il blocco contenente la lista di istruzioni che viene eseguita ogni volta che la funzione viene invocata.

Per l'invocazione del metodo si specifica:

- Il nome della funzione da invocare (Ide_f)
- l'espressione valore del parametro (*attuale*)

Consideriamo il seguente esempio di programma nel linguaggio didattico:

```
int main(){
   int square (int p) {return p*p; }
   int x=square(3);
   printf("quadrato di 3 \%", x);   }
```

La prima riga del programma mostra un esempio di definizione di funzione. La funzione definita ha nome `square`, calcola un valore di tipo `int`, ha un parametro di nome `p` e di tipo `int` e un corpo costituito da un'unica istruzione `return p*p`. Nel corpo del programma principale, l'espressione che inizializza la variabile `x` nella dichiarazione è un esempio di invocazione della funzione `square` con parametro attuale `3`. Per poter invocare una funzione precedentemente definita le informazioni fornite al momento della definizione devono essere reperibili nello stato al momento dell'invocazione. Questo viene ottenuto inserendo nell'ambiente come denotazioni le definizioni delle funzioni, in realtà nella semantica definita dal momento che non ci occupiamo di controllo dei tipi, è sufficiente il nome del parametro (formale), il corpo e poi è necessario inserire l'ambiente di definizione della funzione. Tale informazione è necessaria per la politica così detta di `scoping statico` (l'alternativa sarebbe lo `scoping dinamico`) del linguaggio. Per illustrare la differenza tra scoping statico e dinamico, si consideri il seguente programma:

```
int main(){
   int y=1;
   int square (int p) {return p*p; }
   int z=0;
   int x=square(3);
   printf("quadrato di 3 \%", x);   }
```

Il programma è analogo a quello mostrato precedentemente ma leggermente più complicato per la definizione nel `main` di 2 variabili: `y` definita prima della definizione della funzione `square` e `z` definita dopo. Lo scoping di un linguaggio definisce quali sono gli identificatori noti e di conseguenza utilizzabili, in una funzione. Con lo scoping statico gli identificatori noti sono quelli esistenti nell'ambiente al momento della definizione della funzione. Con lo scoping dinamico viceversa gli identificatori utilizzabili in una funzione sono quelli esistenti nell'ambiente di invocazione. Nell'esempio sopra, in caso di scoping statico, la variabile `y` è utilizzabile in `square` mentre `z` no. In caso di scoping dinamico entrambe le variabili sono utlizzabili.

In definitiva le informazioni che devono essere memorizzata nell'ambiente per una funzione sono tre:

1. il nome del parametro formale
2. il corpo della funzione

3. l'ambiente di definizione

`FunDecl` è infatti una tripla $\texttt{Ide} \times \texttt{SList} \times \Sigma$ e la definizione di una funzione inserisce nell'ambiente associata al nome della funzione la tripla $\langle \texttt{x}, \texttt{B}, \sigma \rangle$, con σ ambiente di definizione della funzione.

$$\frac{\sigma = \varphi.\sigma'' \qquad \sigma' = \varphi[^{\langle \texttt{ide}_p, \texttt{B}, \sigma' \rangle}/_{\texttt{ide}_f}].\sigma''}{\langle \texttt{int ide}_f\texttt{(int ide}_p\texttt{) \{B\}}, \langle \sigma, \mu \rangle \rangle \rightarrow_{dec} \langle \sigma' \mu \rangle} \rightarrow_{fundecl}$$

5.5.2 Invocazione di funzione

È abbastanza ovvio che l'invocazione di funzione richiede la lista di statements che costituiscono il corpo della funzione. Il problema che si pone è in quale stato tali istruzioni sono eseguite. Ovvero quali sono gli identificatori contenuti nello stato, lo chiamiamo, di *esecuzione* e quali valori sono associati a tali identificatori? Come si fa a calcolare un valore? Lo stato di esecuzione di una funzione è costituito da

- come primo componente l'ambiente di definizione della funzione (scoping statico), memorizzato come terzo componente nella denotazione associata alla funzione, esteso con alcuni nuovi legami *locali*, come vedremo,
- come secondo componenete la memoria dell'invocazione, in cui sono state allocate nuove locazioni, come vedremo, per ontenere i valori degli identificatori *locali*.

Gli identificatori introdotti riguardano il parametro e al valore di ritorno. Per entrambi viene allocata una locazione di memoria e inserito un nuovo legame nell'ambiente. Nel caso del parametro il legame è tra il nome del formale associato ad una nuova locazione in cui è contenuto il valore dell'attuale. nel caso del valore di ritorno il nome dell'identificatore è standard `retval` e il valore è ϖ. La semantica dell'invocazione di funzione, che estende il sistema $\rightarrow_{exp}$ può ora essere definita. La prima regola definita mostra che la valutazione dell'invocazione del metodo calcola il valore di `retval` nello stato risultante dalla valutazione del corpo del metodo invocato. Il corpo del metodo viene valutato in uno stato σ' costituito da un solo frame che contiene un legame per il parametro, formale legato al valore dell'attuale. Il formale è l'identificatore `x` reperito in ρ_s e l'attuale è l'espressione `E` a cui il metodo è applicato. La semantica è formalmente definita dalle seguenti regole:

$$\frac{\begin{array}{c}\sigma(\texttt{m}) = \langle \texttt{ide}_p, \texttt{B}, \sigma'' \rangle \qquad \langle \texttt{E}, \langle \sigma, \mu \rangle \rangle \rightarrow_{exp} \ \langle \texttt{v}, \langle \sigma, \mu' \rangle \rangle \\ \langle l, \mu'' \rangle = \texttt{free}(\mu') \qquad \langle l', \mu''' \rangle = \texttt{free}(\mu'') \\ \sigma' = \omega[^{l}/_{\texttt{ide}_p}, ^{l'}/_{\texttt{retval}}].\sigma'' \qquad \mu'''' = (\mu'''[^{\texttt{v}}/_{l}]) \qquad \langle \texttt{B}, \langle \sigma', \mu'''' \rangle \rangle \rightarrow_{com} \ \langle \sigma', \mu''''' \rangle\end{array}}{\langle \texttt{m(E)}, \langle \sigma, \mu \rangle \rangle \rightarrow_{com} \langle \mu'''''(l'), \langle \sigma, \mu''''' \rangle \rangle} \ (exp_{FunCall})$$

$$\frac{\langle \texttt{E}, \langle \sigma, \mu \rangle \rangle \rightarrow_{exp} \ \langle \texttt{v}, \langle \sigma, \mu' \rangle \rangle \quad l = \sigma(\texttt{retval})}{\langle \texttt{return E}, \langle \sigma, \mu \rangle \rangle \rightarrow_{com} \langle \sigma, \mu'[^{\texttt{v}}/_{l}] \rangle} \ (exp_{return})$$

Esercizi

1. Si calcolino gli stati ai punti (1), (2), (3), (4), (5), (6), e (7) del seguente programma **C**.

```
int main(){
    int i=10;                    (1)
```

```
    int f (int x) {                    (3)
        int z=x+1;                     (4)
        if (x > 0) return z*2;
        else return z/2                (5)
        }                              (2)
    i=f(5);                            (6)
    int y=i+10;                        (7)
}
```

2. Si calcolino gli stati ai punti (1), (2), (3), (4), (5), (6), e (7) del seguente programma **C**.

```
int main(){
    int i=10;                          (1)
    int f (int x) {                    (3)
        int i=0;                       (4)
        while (x > 0)
            {x=x-2;
            i=i+1;}                    (5)
        return i;}                     (2)
    int j=0;
    int y=f(5);                        (6)
    j=y+10;                            (7)
}
```

6 Equivalenza di programmi

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 semantica in qualunque contesto, ovvero a partire dallo stesso stato iniziale. Più formalmente ciò può essere espresso come segue.

Definizione 6.1] Due comandi `C` e `C'` sono equivalenti se per ogli stato $\langle \sigma, \mu \rangle$ vale una delel seguenti condizioni:

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

b) $\langle \texttt{C}, \langle \sigma, \mu \rangle \rangle \rightarrow^* \langle \sigma', \mu' \rangle \rangle$ e $\langle \texttt{C}', \langle \sigma, \mu \rangle \rangle \rightarrow^* \langle \sigma', \mu'' \rangle \rangle$ e $\forall x \in \texttt{Ide} \mid \sigma'(x) = l \neq \perp$ vale che $\mu'(l) = \mu''(l)$.

Questa equivalenza è molto forte in quanto richiede che i due comandi abbiano come semantica due stati in cui l'ambiente è uguale e le memorie siano uguali per tutte le locazioni che risultano raggiungibili dall'ambiente. La dimostrazione di equivalenza può essere data per casi se necessario e si ottiene derivando la configurazione finale a partire da un generico stato iniziale $\langle \sigma, \mu \rangle$ su cui non vengono fatte assunzioni.

Un'altra forma di equivalenza, detta equivalenza *debole*, molto simile alla precedente prevede che vengano fatte delle assunzioni sullo stato iniziale $\langle \sigma, \mu \rangle$ ad esempio sul valore di una o più variabili cioè che ad esempio $\mu(\sigma(x)) = v$.

Esercizi

1. Si dimostri l'equivalenza dei seguenti frammenti di programmi:

```
C1 :  if (x==y)  x=x-y else  x= y-y;
C2 :  x=0;
```

2. Si dimostri l'equivalenza dei seguenti frammenti di programmi:

C_1 : `if (true) x=5; else x=3;`
C_2 : `if (false) x=3; else x=5;`