

## Traduzione semantica e analisi statica

### 5.1 Introduzione

Accanto all'esigenza di riconoscere se una frase è corretta, si pone naturalmente l'obiettivo di tradurre o trasformare la frase, come fa un compilatore quando converte un programma da un linguaggio come C++ a un codice macchina. Allo studio dei metodi di traduzione è dedicato questo capitolo.

Una traduzione è una corrispondenza, in particolare una funzione, tra le frasi del linguaggio sorgente e quelle del linguaggio pozzo. Come nel caso della decisione circa la validità d'una frase, anche per la traduzione si danno due punti di vista: quello generativo impiega gli schemi sintattici di traduzione per generare le coppie di frasi, sorgente e pozzo, che si corrispondono nella traduzione. Uno schema di traduzione è infatti costituito dall'accoppiamento di due grammatiche generative.

Il punto di vista riconoscitivo o algoritmico si avvale degli automi traduttori, che si differenziano dai riconoscitori per la propria capacità di emettere la traduzione voluta.

Tali metodi vanno sotto il nome di teoria sintattica della traduzione. Essi sono il coronamento dei metodi sintattici, che sono impiegati nel progetto di tutti i compilatori, ma da soli certo non bastano per tale scopo. Essi infatti mancano d'una dimensione fondamentale, lo studio del significato (o semantica) delle frasi.

Per definire il significato d'un linguaggio, si esporranno le grammatiche attributi, un valido metodo ingegneristico per progettare i compilatori in modo ordinato.

#### *Frontiera tra sintassi e semantica*

Non è semplice chiarire la distinzione, spesso arbitraria, tra sintassi e semantica. L'etimologia di queste parole non conduce oltre la vaga spiegazione che la prima considera la struttura e la seconda il significato delle frasi o il messaggio che devono comunicare. Nella linguistica i due termini sono stati gli emblemi

d'una distinzione tra forma e contenuto che, anche se apparentemente ben posta, diventa elusiva quando si vuole approfondire.

Nel campo dei linguaggi formalizzati, è più facile delineare la divisione tra metodi sintattici e semantici, che per quanto arbitraria, è da tutti accettata perché ha ragioni pragmatiche.

La prima diversità tra sintassi e semantica sta nei domini delle entità oggetto dello studio e degli operatori impiegati per manipolarle. La sintassi impiega i concetti e gli operatori della teoria formale dei linguaggi, e descrive gli algoritmi mediante gli automi. Le entità trattate dalla sintassi sono dunque gli alfabeti, le successioni di simboli o stringhe, le operazioni di concatenamento, ripetizione, sostituzione o cambiamento di alfabeto. Le operazioni aritmetiche (somma, prodotto, ecc.) e i numeri sono del tutto estranei alla sintassi. Invece la semantica non limita *a priori* le entità di cui si serve, che possono essere dei numeri, o anche delle strutture-dati più complesse (insiemi di numeri, tabelle o relazioni, ecc.), come quelle definibili nei comuni linguaggi di programmazione. Ben inteso, la semantica si avvale della sintassi come di un'utile struttura per l'applicazione delle proprie funzioni.

La seconda diversità risiede nella maggiore complessità computazionale degli algoritmi semantici rispetto a quelli sintattici. Si è ricordato che i linguaggi formali qui studiati sono quasi esclusivamente quelli deterministici, riconoscibili e traducibili con un tempo di calcolo lineare, cioè proporzionale alla lunghezza della frase da analizzare. Questa limitazione presenta grandi vantaggi, ma non consente tutti i controlli di correttezza, realmente necessari per analizzare un linguaggio: ad es. non è possibile verificare con i metodi sintattici se l'identificatore d'una variabile in un programma Java sia stato dichiarato correttamente, prima di essere usato in un'espressione. In effetti tale controllo non è fattibile in tempo lineare. Per questo e altri controlli si ricorre a formulazioni non sintattiche, che possono essere dette semantiche.

Approfondendo il confronto, la scelta tra i modelli semantici o sintattici è soltanto una forma di convenienza. Infatti è noto dalla teoria della computabilità che ogni funzione calcolabile, in particolare quella che decide se una stringa è valida e la traduce, può essere realizzata mediante una macchina di Turing. Essa appartiene certamente alla categoria dei formalismi sintattici, perché si limita a agire sulle stringhe con le operazioni della teoria dei linguaggi formali. Ma questo modello, anche se superiore dal punto di vista della potenza teorica, è praticamente inutilizzabile: nessun programmatore si è mai sognato di codificare un programma con le istruzioni d'una macchina di Turing. L'esperienza ha mostrato che la leggibilità e convenienza dei metodi sintattici, automi e grammatiche, decadono rapidamente appena si esce dal modello formale dei linguaggi liberi dal contesto.

### 5.1.1 Contenuti

Nel senso comune la parola traduzione indica la corrispondenza tra due frasi appartenenti a lingue umane diverse. Anche nel campo dei linguaggi artifi-

ciali si incontrano molti casi di traduzione: la compilazione da un linguaggio programmatico al codice macchina d'un processore; la trasformazione di un documento dal formato HTML (usato nella Rete) al formato PDF (usato per i documenti non modificabili), ecc.. Il linguaggio di partenza della traduzione è detto *sorgente* e quello d'arrivo è detto *pozzo*.

L'esposizione inizierà con una definizione astratta della corrispondenza tra due linguaggi formali.

Il secondo passo presenterà le traduzioni basate su trasformazione locali del testo sorgente, prodotte dalla sostituzione di un carattere con un altro (o con una stringa), in accordo con una tabella di traslitterazione tra i due alfabeti, sorgente e pozzo.

Il terzo passo porterà alle traduzioni definite mediante espressioni regolari e traduttori finiti, ossia automi finiti arricchiti della capacità di emettere una stringa.

Il quarto passo presenterà gli schemi sintattici o grammatiche di traduzione, che, invece del linguaggio regolare del modello precedente, usano una grammatica libera per definire i linguaggi sorgente e pozzo. Tali traduzioni sono anche caratterizzate dalla macchina astratta che le calcola, il traduttore a pila, che sfrutta gli algoritmi di parsificazione del capitolo precedente.

Le precedenti classi di traduzioni sono dette puramente sintattiche, ma solo in piccola parte rispondono alle esigenze del progetto dei compilatori, perché molte trasformazioni linguistiche necessarie sono più articolate di quelle esprimibili con tali metodi. Ciò non di meno, la loro conoscenza dà la base concettuale e serve da guida ai metodi semantici di traduzione, che sono quelli realmente applicati nella compilazione. Inoltre, le traduzioni sintattiche hanno un'altra utilità: permettono di confrontare tra loro due linguaggi per scoprire le somiglianze e la natura delle differenze.

A mo' di guida alla lettura, conviene evidenziare alcune analogie significative tra la struttura della teoria dei linguaggi e quella della teoria delle traduzioni. Sul piano della definizione insiemistica: l'insieme delle frasi del linguaggio corrisponde all'insieme delle coppie (stringa sorgente, stringa pozzo) costituenti la relazione di traduzione. Sul piano delle definizioni generative: la grammatica del linguaggio diventa quella della traduzione, che genera coppie di frasi sorgente e pozzo. Sul piano delle definizioni operative: l'automa riconoscitore a stati finiti o a pila diventa l'automa traduttore o l'analizzatore sintattico, che calcolano la traduzione. Queste corrispondenze appariranno chiaramente nel corso del capitolo.

Il quinto passo sono le traduzioni semantiche guidate dalla sintassi, un approccio semiformale che si appoggia sui concetti precedenti e permette di progettare in modo ordinato i compilatori. Questo metodo si esprime nel modello delle grammatiche con attributi; esse specificano regola per regola le funzioni da applicare per calcolare la traduzione in maniera compositiva.

L'approccio sarà esemplificato da diversi casi tipici. L'aggiunta di attributi e funzioni semantiche ai traduttori finiti trova applicazione nell'analisi lessicale, il primo stadio della compilazione. Altri casi significativi sono: il controllo dei

tipi, la traduzione delle istruzioni condizionali e l'analisi sintattica guidata dalla semantica.

Da ultimo il capitolo espone un altro utile e importante metodo di analisi semantica, specializzata per i programmi eseguibili, anziché per generici linguaggi tecnici. Si tratta dell'analisi statica del grafo di flusso d'un programma, modellato come automa a stati finiti. Il metodo è alla base delle tecniche di verifica e ottimizzazione dei programmi largamente impiegate nei moderni compilatori.

Con l'ultimo argomento si completa il quadro dei metodi elementari di compilazione.

## 5.2 Relazione e funzione di traduzione

La teoria delle traduzioni ha una base matematica profonda, ma tenuto conto delle finalità operative del libro, si tratteranno soltanto i concetti essenziali.<sup>1</sup> Siano dati due alfabeti  $\Sigma$  e  $\Delta$ , detti rispettivamente *sorgente* e *pozzo*. Una traduzione è una corrispondenza tra le stringhe sorgente e pozzo, che può essere formalizzata come una relazione matematica tra i due linguaggi universali  $\Sigma^*$  e  $\Delta^*$ , ossia come un sottoinsieme del prodotto cartesiano  $\Sigma^* \times \Delta^*$ .

Una relazione di traduzione  $\rho$  è un insieme di coppie di stringhe  $(x, y)$ , con  $x \in \Sigma^*$  e  $y \in \Delta^*$ :

$$\boxed{\rho = \{(x, y), \dots\} \subseteq \Sigma^* \times \Delta^*}$$

Si dice che la stringa pozzo  $y$  è *immagine* o *traduzione* della stringa sorgente  $x$ , e che le due stringhe si corrispondono nella traduzione. Data una relazione di traduzione  $\rho$ , i *linguaggi sorgente*  $L_1$  e *pozzo*  $L_2$  sono rispettivamente definiti come le proiezioni della relazione sulla prima e sulla seconda componente:

$$\begin{aligned} L_1 &= \{x \in \Sigma^* \mid \text{per qualche } y : (x, y) \in \rho\} \\ L_2 &= \{y \in \Delta^* \mid \text{per qualche } x : (x, y) \in \rho\} \end{aligned}$$

Un altro modo per formalizzare la traduzione considera l'insieme delle stringhe pozzo che nella traduzione corrispondono alla stessa stringa sorgente. Una traduzione è allora una funzione:

$$\boxed{\tau : \Sigma^* \rightarrow \text{parti finite di } \Delta^*; \quad \tau(x) = \{y \in \Delta^* \mid (x, y) \in \rho\}}$$

dove  $\rho$  è una relazione di traduzione. La funzione mappa così una stringa sorgente nell'insieme delle sue immagini, ossia in un linguaggio.

---

<sup>1</sup>Una formalizzazione matematica rigorosa della traduzione si trova in Berstel [8] e Sakarovitch [42].

Si noti che, applicando ripetutamente la funzione a ogni frase del linguaggio sorgente, si ottiene un insieme di linguaggi, la cui unione costituisce il linguaggio pozzo:

$$L_2 = \tau(L_1) = \{y \in \Delta^* \mid \exists x \in \Sigma^* : y \in \tau(x)\}$$

Un caso restrittivo, ma praticamente rilevante, si ha quando l'immagine di ogni stringa sorgente è unica.

La funzione di traduzione non è generalmente totale nel dominio del linguaggio universale sorgente; ma può essere resa totale, se si conviene che, dove  $\tau(x)$  non è definita, si assegna alla stringa immagine il valore  $y = \text{errore}$ .

La traduzione inversa  $\tau^{-1} : \Delta^* \rightarrow \Sigma^*$  mappa le stringhe pozzo in quelle sorgente:

$$\tau^{-1}(y) = \{x \in \Sigma^* \mid y \in \tau(x)\}$$

A seconda delle proprietà matematiche della funzione, si hanno i seguenti casi di traduzione.

- totale, ogni stringa di alfabeto sorgente ha un'immagine;
- a un solo valore<sup>2</sup> o univoca; nessuna stringa sorgente ha due diverse immagini;
- a più valori: una stringa sorgente ha più traduzioni;
- iniettiva: stringhe sorgente diverse hanno immagini diverse, ovvero a ogni stringa di alfabeto pozzo corrisponde al più una stringa sorgente; la traduzione inversa è dunque univoca;
- suriettiva: una funzione è suriettiva quando l'immagine coincide con il codominio, ovvero quando ogni stringa di alfabeto pozzo è immagine di almeno una stringa del dominio;
- biunivoca (o biiettiva): vi è corrispondenza uno a uno tra le stringhe sorgente e pozzo.

Per concretizzare le idee, si pensi alla corrispondenza tra un programma sorgente scritto in linguaggio di alto livello (es. Java) e il codice macchina d'un certo processore. Per chi sa un po' di programmazione, è ovvio che la traduzione è totale: infatti per ogni programma corretto esiste un codice macchina, e per ogni programma scorretto l'immagine è l'*errore*.

La traduzione è a più valori, poiché la stessa istruzione Java può essere codificata da diverse sequenze di istruzioni macchina.

La traduzione non è iniettiva perché due programmi sorgente possono avere la stessa traduzione: si pensi a due cicli iterativi `while` e `for` che possono essere tradotti nello stesso codice macchina, costituito da istruzione condizionale e salto.

La traduzione non è suriettiva, poiché si possono scrivere programmi macchina che non hanno un corrispondente programma sorgente.

Se però si fissa l'attenzione su un particolare compilatore da Fortran al linguaggio macchina, non solo esso calcola una funzione di traduzione totale

<sup>2</sup>Si intende nel codominio delle stringhe pozzo, non dei linguaggi pozzo.

(infatti qualsiasi stringa sorgente è elaborata dal compilatore), ma la funzione è evidentemente a un solo valore.

La traduzione però potrebbe non essere iniettiva (per le ragioni prima esposte) e certamente non è suriettiva, perché vi sono programmi macchina che il compilatore non produce in nessun caso.

Un decompilatore, dato un codice macchina, ricostruisce un equivalente programma sorgente. Questa traduzione non è però la funzione inversa della compilazione, perché compilatore e decompilatore sono programmi diversi, ed è improbabile che essi scelgano con la stessa logica le loro traduzioni.

Banalmente, il decompilatore  $\delta$ , partendo dal codice macchina  $\tau(x)$  prodotto dal compilatore  $\tau$ , costruirà un programma Fortran  $\delta(\tau(x))$  che di sicuro differisce da  $x$  nella spaziatura!

La compilazione, essendo una corrispondenza tra le stringhe di due linguaggi solitamente infiniti, non può essere specificata elencando le infinite coppie della relazione di traduzione. L'esposizione continua con la presentazione, partendo dai più semplici, di vari modi per specificare e realizzare una traduzione.

### 5.3 Traslitterazioni OMOMORFISMO

Il primo modo di definire una traduzione è attraverso la ripetizione, carattere per carattere, di trasformazioni puntuali. La trasformazione più semplice è la traslitterazione o omomorfismo, già definita nel cap. 2 a p. 80. Ogni carattere sorgente è trascodificato in un corrispondente carattere pozzo (o più in generale in una stringa).

Conviene rivedere l'esempio 2.84 di p. 80, come illustrazione d'una traduzione definita tramite omomorfismo.

La traduzione definita da un omomorfismo alfabetico è evidentemente univoca, ma non sempre è univoca quella inversa; nell'esempio citato il quadratino  $\square$  è l'immagine di tutte le lettere greche, quindi la traduzione inversa non è univoca:

$$h^{-1}(\square) = \{\alpha, \dots, \omega\}$$

Se l'omomorfismo cancella certi caratteri sorgente, come in questo caso i caratteri start-text, end-text, la traduzione inversa non è mai univoca, perché in ogni punto del testo si può inserire una stringa arbitraria composta con i caratteri cancellati.

Se anche la funzione inversa è univoca, la corrispondenza tra la stringa sorgente e la stringa pozzo è biunivoca, quindi è possibile risalire dalla seconda alla prima.

Questa situazione si incontra nei cifrari usati per nascondere un messaggio segreto. Un elementare cifrario, definito mediante traslitterazione, è quello di Giulio Cesare, che codifica una lettera di posto  $i$  (con  $i = 1, \dots, 26$ ) dell'alfabeto latino con la lettera di posto  $(i + k) \bmod 26$ , dove  $k$  è una costante, che

funge da chiave segreta.

Si sottolinea che la traslitterazione è un processo che non considera il contesto della lettera via via tradotta. Va da sè che questo modo di operare è del tutto insufficiente per la compilazione dei linguaggi tecnici.

## 5.4 Traduzioni regolari

Per definire una relazione di traduzione si può sfruttare l'approccio delle espressioni regolari, modificato in modo che gli elementi, cui si applicano gli operatori di unione, concatenamento e stella, siano delle coppie di stringhe sorgente e pozzo. Così facendo, una "frase" derivata dall'espressione regolare è il concatenamento di coppie di stringhe; separando i caratteri della sorgente da quelli del pozzo, si ottengono le due stringhe corrispondenti nella traduzione.

Tale espressione regolare definisce dunque una relazione di traduzione che è detta regolare o razionale.

*Esempio 5.1.* Traslitterazione coerente d'un operatore.

Il testo sorgente è una lista di numeri separati dal segno "/" di divisione. La traduzione traslittera il segno di divisione nel segno ":" o nel segno "÷", ma sempre nello stesso modo. Per semplicità i numeri sono scritti in notazione unaria.

Gli alfabeti sono:

$$\Sigma = \{1, /\} \quad \Delta = \{1, \div, :\}$$

Le stringhe sorgente sono del tipo  $c(/c)*$ , dove per brevità  $c$  sta per una cifra unaria,  $1^+$ . Sono ad es. corrette le traduzioni:

$$(3/5/2, 3 : 5 : 2), (3/5/2, 3 \div 5 \div 2)$$

ma non la  $(3/5/2, 3 : 5 \div 2)$ , perché gli operatori sono diversamente tradotti. Questa trasformazione non può essere formulata come omomorfismo, perché non è univoco il carattere pozzo da sostituire al segno di divisione (per inciso la traduzione inversa è un omomorfismo alfabetico). L'espressione regolare della traduzione è:

$$(1, 1)^+ ((/, :)(1, 1)^+)^* \cup (1, 1)^+ ((/, \div)(1, 1)^+)^*$$

Per migliore leggibilità, le coppie saranno scritte come frazioni, con l'elemento sorgente (risp. pozzo) a numeratore (risp. a denominatore):

$$\left(\frac{1}{1}\right)^+ \left(\frac{/}{:} \left(\frac{1}{1}\right)^+\right)^* \cup \left(\frac{1}{1}\right)^+ \left(\frac{/}{\div} \left(\frac{1}{1}\right)^+\right)^*$$

I termini che derivano dall'espressione sono delle stringhe i cui elementi sono coppie di caratteri sorgente e pozzo, come ad es.  $\frac{/}{:}$ . Così la stringa

$$\begin{array}{c} 1 / 11 \\ 1 \div 11 \end{array}$$

proiettata nella prima e seconda componente, produce le due stringhe corrispondenti ( $1/11$ ,  $1 \div 11$ ).

**Definizione 5.2.** Un' espressione regolare o razionale di traduzione (e.r.t.)  $r$  è un'espressione regolare con gli operatori unione, concatenamento stella (croce) i cui termini sono coppie  $(u, v)$ , spesso scritte  $\frac{u}{v}$ , di stringhe, anche vuote, di alfabeto rispettivo sorgente e pozzo.

Sia  $C \subset \Sigma^* \times \Delta^*$  l'insieme delle coppie  $(u, v)$  che compaiono nell'espressione. La relazione di traduzione, detta regolare o razionale, definita dalla e.r.t., contiene le coppie  $(x, y)$  di stringhe sorgente e pozzo, tali che:

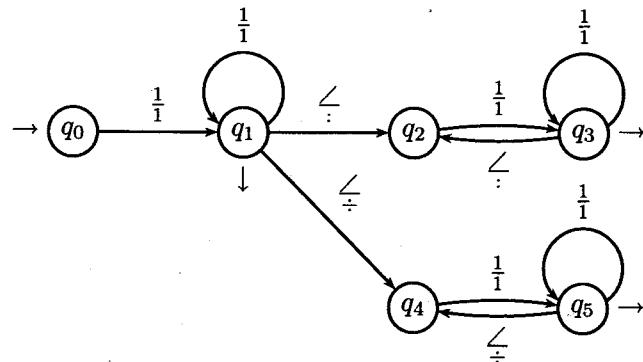
- esiste una stringa  $z \in C^*$  appartenente all'insieme regolare definito da  $r$ ;
- $x$  e  $y$  sono rispettivamente la proiezione di  $z$  sulla prima componente e sulla seconda.

È facile vedere che l'insieme delle stringhe sorgente definite da una e.r.t. è un linguaggio regolare, così come l'insieme delle stringhe pozzo. Si noti però che non è affatto detto che una relazione di traduzione avente linguaggio sorgente e linguaggio pozzo regolari sia definibile mediante una e.r.t.: un esempio trattato più avanti è la relazione che fa corrispondere a una stringa del linguaggio sorgente universale la stringa specularmente riflessa.

#### 5.4.1 Automa riconoscitore a due ingressi

Fissando l'attenzione sull'insieme  $C$  delle coppie usate nella e.r.t., pensato come se fosse un alfabeto terminale, è immediato associare un automa finito alla traduzione: esso è il riconoscitore del linguaggio regolare di alfabeto  $C$ . Ciò è illustrato dal prossimo esempio.

*Esempio 5.3.* Es. 5.1 cont.: riconoscitore di relazione regolare di traduzione.



Quest'automa può essere materializzato come una macchina dotata di due nastri d'ingresso, detti sorgente e pozzo, ciascuno scandito da una testina di lettura. Inizialmente nei nastri vi sono le stringhe sorgente e pozzo,  $x$  e  $y$ , le testine stanno sui primi caratteri e lo stato è quello iniziale. La macchina esegue le mosse specificate dal grafo; così, nello stato  $q_1$ , leggendo la barra “ $/$ ” dal nastro sorgente e il segno “ $\div$ ” dal nastro pozzo, l'automa si porta nello stato  $q_4$ , e sposta entrambe le testine d'una posizione. Se la macchina si trova in uno stato finale quando entrambi i nastri sono stati completamente letti, la coppia di stringhe  $(x, y)$  appartiene alla relazione di traduzione.<sup>3</sup>

L'automa può ad es. verificare che due stringhe, come

$$(11/1, 1 \div 1) \equiv \frac{11/1}{1 \div 1}$$

non si corrispondono nella traduzione, perché il calcolo

$$q_0 \xrightarrow{\frac{1}{1}} q_1$$

non può proseguire con la lettura della prossima coppia di caratteri,  $\frac{1}{\div}$ . Sebbene il concetto di e.r.t. e di riconoscitore a due nastri può sembrare inutile per il progetto dei compilatori, poiché nella compilazione la stringa immagine non è data, ma deve essere calcolata dal traduttore stesso, questo approccio ha valore come tecnica di specifica della traduzione stessa e come metodo per la costruzione rigorosa di funzioni di traduzione. Esso permette inoltre di trattare in modo uniforme le traduzioni calcolate dagli analizzatori lessicali e da quelli sintattici.

### Forme dell'automa a due ingressi

Nel descrivere il riconoscitore a due ingressi si può supporre, senza perdita di generalità, che ogni sua mossa legga uno, e un solo carattere dal nastro sorgente; invece dal nastro pozzo una mossa può leggere zero o più caratteri.

#### Definizione 5.4. 2I-automa.

Un automa finito a due ingressi o 2I-automa possiede, come un automa finito a un solo ingresso (p. 107), un insieme di stati  $Q$ , lo stato iniziale  $q_0$ , un insieme  $F \subseteq Q$  di stati finali. La funzione di transizione è

$$\delta : (Q \times \Sigma \times \Delta^*) \rightarrow \text{parti finite di } Q$$

Se  $q' \in \delta(q, a, u)$ , l'automa, leggendo  $a$  dal primo nastro e  $u$  dal secondo, può andare nello stato prossimo  $q'$ . La condizione di riconoscimento è del tutto analoga a quella degli automi finiti non deterministici.

<sup>3</sup>Questo modello di macchina è noto come automa di Rabin e Scott. Spesso si fa l'ipotesi che ogni nastro sia terminato da una speciale marca di fine testo. Inoltre, per maggiore generalità, la macchina può avere più di due nastri di ingresso, così permettendo di definire relazioni tra più di due stringhe.

Quando in un  $2I$ -automa si proiettano le etichette degli archi del grafo sulla prima componente, si ottiene un automa con un solo ingresso di alfabeto  $\Sigma$ , che è detto l' *automa d'ingresso soggiacente* alla traduzione. Esso riconosce il linguaggio sorgente.

Talvolta si considera una riformulazione del modello precedente, detta  $2I$ -automa in forma normale in cui una mossa può leggere un carattere da uno solo dei due nastri. Più precisamente, le etichette degli archi sono dei seguenti tipi:

*FERTA*  
*MQRTA*

$\frac{a}{\epsilon}, a \in \Sigma$  lettura dal nastro sorgente;  
 $\frac{\epsilon}{b}, b \in \Delta$ , lettura dal nastro pozzo.

Un automa in forma normale muove dunque una sola testina alla volta. Come spesso accade con le forme normali, anche ora per ottenere la normalizzazione si accresce il numero di stati dell'automa, perché alla mossa  $\frac{a}{b}$  si sostituisce la doppietta equivalente di mosse normalizzate  $\frac{a}{\epsilon} \rightarrow q_r \rightarrow \frac{\epsilon}{b}$  dove  $q_r$  è un nuovo stato, e così via.

D'altra parte, per migliorare l'espressività e la concisione del modello, si può permettere di scrivere delle espressioni regolari di traduzione sulle etichette del  $2I$ -automa. Come per gli automi, questa generalizzazione non aumenta la potenza del modello, ma facilita la descrizione di situazioni un po' complesse. Inoltre, per brevità, nelle etichette si potrà omettere l'elemento sorgente (numeratore) o pozzo (denominatore) quando esso è la stringa vuota, scrivendo ad es.

$$\frac{(a)^* b}{d} \mid \frac{(a)^* c}{e}$$

al posto di

$$\frac{(a)^*}{\epsilon} \frac{b}{d} \mid \frac{(a)^*}{\epsilon} \frac{c}{e}$$

L'espressione dice che una sequenza di caratteri  $a$ , se è seguita da  $b$ , si traduce in  $d$ ; se è seguita da  $c$ , si traduce in  $e$ .

### Equivalenza dei modelli

In armonia con la nota equivalenza delle espressioni regolari e degli automi finiti, vale la seguente proprietà.

Proprietà 5.5. La famiglia delle relazioni regolari di traduzione e quella delle relazioni riconosciute da un  $2I$ -automa finito (in generale non deterministico) coincidono.

Una e.r.t. definisce un linguaggio regolare  $R$  avente come alfabeto un insieme di coppie di stringhe  $(u, v) \equiv \frac{u}{v}$ , dove  $u \in \Sigma^*, v \in \Delta^*$ . Separando in tale linguaggio i caratteri terminali sorgente e pozzo, si ottiene una significativa formulazione del rapporto esistente tra linguaggi e traduzioni regolari.

*Proprietà 5.6. Teorema di Nivat.*

Le seguenti condizioni sono equivalenti:

1. La relazione di traduzione  $\rho \subseteq \Sigma^* \times \Delta^*$  è regolare.
2. Esiste un alfabeto  $\Omega$ , un linguaggio regolare  $R$  di alfabeto  $\Omega$  e due omomorfismi alfabetici  $h_1 : \Omega \rightarrow \Sigma \cup \{\epsilon\}$  e  $h_2 : \Omega \rightarrow \Delta \cup \{\epsilon\}$  tali che

$$\rho = \{(h_1(z), h_2(z)) \mid z \in R\}$$

3. Se i due alfabeti sorgente e pozzo sono disgiunti, esiste un linguaggio regolare  $R'$  di alfabeto  $\Sigma \cup \Delta$  tale che

$$\rho = \{(h_\Sigma(z), h_\Delta(z)) \mid z \in R'\}$$

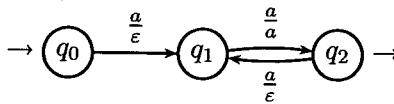
dove  $h_\Sigma$  e  $h_\Delta$  sono rispettivamente le proiezioni dall'alfabeto  $\Sigma \cup \Delta$  agli alfabeti sorgente e pozzo.

*Esempio 5.7. Divisore per due.*

La stringa immagine è quella sorgente dimezzata. La relazione di traduzione  $\{(a^{2n}, a^n) \mid n \geq 1\}$  è definita dalla e.r.t.

$$\left(\frac{aa}{a}\right)^+$$

Un  $2I$ -automa equivalente  $A$  è mostrato in figura



Per applicare il teorema di Nivat (ramo 2 dell'enunciato) si consideri la e.r.t. ricavata dall'automa  $A$ :

$$\left(\frac{aa}{\epsilon a}\right)^+$$

Per chiarezza di scrittura, si ridenominano le coppie:

$$\frac{a}{\epsilon} = c \quad \frac{a}{a} = d$$

L'alfabeto da considerare è  $\Omega = \{c, d\}$ . La e.r.t., sostituendo alle frazioni i nuovi simboli, diventa il linguaggio regolare  $R = (cd)^+$ . Gli omomorfismi alfabetici

|     | $h_1$ | $h_2$      |
|-----|-------|------------|
| $c$ | $a$   | $\epsilon$ |
| $d$ | $a$   | $a$        |

producono la relazione di traduzione voluta. Ad es., preso  $z = cdcd \in R$ , si ha  $h_1(z) = aaaa$ ,  $h_2(z) = aa$ .

Per applicare il ramo 3 del teorema, si modifichi l'alfabeto pozzo in  $\Delta = \{b\}$ , per disgiungerlo da quello sorgente, ridefinendo la traduzione come  $\{(a^{2n}, b^n) \mid n \geq 1\}$ . La e.r.t. coerentemente modificata

$$\left( \begin{array}{c|c} a & a \\ \hline \epsilon & b \end{array} \right)^+$$

si trasforma ora nel linguaggio regolare  $R' \subseteq (\Sigma \cup \Delta)^*$  dell'enunciato, cancellando le linee di frazione e concatenando le stringhe a numeratore e quelle a denominatore:

$$R' = (aa\epsilon b)^+ = (aab)^+$$

Le proiezioni sugli alfabeti sorgente e pozzo definiscono le stringhe corrispondenti nella traduzione.

La nota corrispondenza tra il modello degli automi finiti e delle grammatiche lineari a destra (p. 106), permette di scrivere una cosiddetta grammatica di traduzione al posto del 2I-automa o della e.r.t.. È sufficiente mostrare ciò sul precedente esempio:

$$S \rightarrow \frac{a}{\epsilon} Q_1 \quad Q_1 \rightarrow \frac{a}{a} Q_2 \quad Q_2 \rightarrow \frac{a}{\epsilon} Q_1 \mid \epsilon$$

Ogni regola grammaticale corrisponde a una mossa del 2I-automa. La regola  $Q_2 \rightarrow \epsilon$  abbrevia  $Q_2 \rightarrow \frac{\epsilon}{\epsilon}$ .

Questo genere di definizione mediante una grammatica sarà più avanti preferito nelle traduzioni sintattiche, che hanno come supporto una grammatica libera.

Molte proprietà dei linguaggi e delle espressioni regolari trovano analogia formulazione<sup>4</sup> per le relazioni regolari di traduzione. In particolare l'unione, l'intersezione e il complemento di relazioni regolari sono relazioni regolari; e si potrebbe anche enunciare una proprietà simile al lemma di pompaggio 2.74 per le relazioni regolari.

#### 5.4.2 Funzione di traduzione e automa traduttore

Conviene lasciare la prospettiva, interessante ma troppo statica della relazione tra due linguaggi, per studiare un automa come l'algoritmo che calcola una funzione di traduzione. Il nuovo modello è il traduttore finito o IO-automa, che legge la stringa sorgente dall'ingresso e scrive nell'uscita la stringa immagine. Si approfondirà il caso più rilevante delle funzioni di traduzione a un solo

<sup>4</sup>Vedasi [8, 42].

valore, e, all'interno di queste, si studieranno quelle calcolabili da un automa deterministico.

Conviene iniziare con un esempio di traduttore.

*Esempio 5.8.* Traduzione non deterministica.

Si vuole tradurre una stringa  $a^n$  nella stringa  $b^n$ , se  $n$  è pari,  $c^n$ , se  $n$  è dispari.  
La relazione

$$\rho = \{(a^{2n}, b^{2n}) \mid n \geq 0\} \cup \{(a^{2n+1}, c^{2n+1}) \mid n \geq 0\}$$

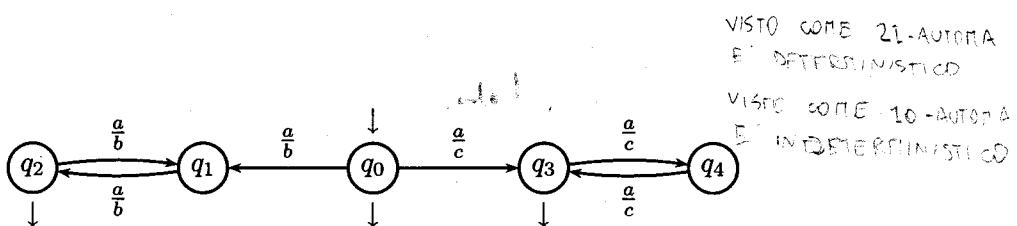
definisce la funzione di traduzione

$$\tau(a^n) = \begin{cases} b^n, & n \text{ pari,} \\ c^n, & n \text{ dispari.} \end{cases}$$

L'e.r.t. è

$$\left(\frac{a^2}{b^2}\right)^* \cup \frac{a}{c} \left(\frac{a^2}{c^2}\right)^*$$

L'automa a due ingressi corrispondente è deterministico:



Infatti soltanto dallo stato  $q_0$  escono due frecce, ma le loro etichette sono diverse.<sup>5</sup> Ciò significa che la macchina può controllare deterministicamente se una coppia di stringhe come  $\frac{aaaa}{bbbb}$ , inizialmente disposte sui due nastri, appartiene alla relazione.

Il traduttore o *IO*-automa ha lo stesso grafo del *2I*-automa, ma ora l'arco  $q_0 \xrightarrow{\frac{a}{b}} q_1$  significa: leggendo  $a$  dall'ingresso, emetti  $b$ ; poiché l'arco  $q_0 \xrightarrow{\frac{a}{c}} q_3$  similmente dice di scrivere  $c$  alla lettura di  $a$ , vi è una scelta indeterministica tra due azioni di scrittura. Data ad es. la stringa sorgente  $aa$  vi sono due calcoli possibili:

$$q_0 \rightarrow q_1 \rightarrow q_2; \quad q_0 \rightarrow q_3 \rightarrow q_4$$

ma solo il primo è valido perché raggiunge uno stato finale. Risulta allora  $\tau(aa) = bb$ . L'indeterminismo del traduttore si manifesta chiaramente nell'automa d'ingresso soggiacente, che è indeterministico.

<sup>5</sup>Se il *2I*-automa presenta mosse che non leggono da uno dei nastri, la condizione di determinismo deve essere formulata più attentamente, si veda [42].

Questo esempio ha mostrato una traduzione a un solo valore che non può essere calcolata deterministicamente da un *IO-automa* finito. Se ne conclude che la nota proprietà di equivalenza tra automi finiti deterministici e non, non vale nel caso dei *IO-automi* o traduttori.

### Traduttore sequenziale

Nelle applicazioni il calcolo della funzione di traduzione deve essere svolto efficientemente. L'algoritmo, via via che scandisce l'ingresso, deve costruire la stringa immagine. Al termine della lettura, ossia alla lettura della marca di fine testo, l'automa potrà anche concatenare alla stringa un ultimo pezzo, che dipende dallo stato finale in cui l'automa si trova. Segue la formalizzazione d'un modello deterministico di traduttore, detto sequenziale<sup>6</sup>.

**Definizione 5.9.** Un traduttore finito o *IO-automa sequenziale*  $T$  è una macchina deterministica così definita. Esso ha un insieme  $Q$  di stati, un alfabeto sorgente  $\Sigma$  e pozzo  $\Delta$ , uno stato iniziale  $q_0$ , un insieme  $F \subseteq Q$  di stati finali.

Vi sono inoltre tre funzioni, tutte a un solo valore:

1. la funzione di transizione,  $\delta$ , calcola lo stato prossimo;
2. la funzione di uscita,  $\eta$ , calcola la stringa da scrivere in concomitanza d'una mossa;
3. la funzione finale  $\varphi$  calcola l'ultimo suffisso da concatenare (eventualmente) alla traduzione, al termine del calcolo.

I domini delle funzioni sono:

$$\delta : Q \times \Sigma \rightarrow Q, \quad \eta : Q \times \Sigma \rightarrow \Delta^*, \quad \varphi : F \times \{\dashv\} \rightarrow \Delta^*$$

Graficamente, la coppia di funzioni  $\delta(q, a) = q'$ ,  $\eta(q, a) = u$  è rappresentato dall'arco  $q \xrightarrow{\frac{a}{u}} q'$ , e significa: nello stato  $q$ , leggendo in ingresso  $a$ , emetti la stringa  $u$  e va nello stato  $q'$ .

La funzione  $\varphi(r, \dashv) = v$  significa: al termine della lettura, se lo stato finale è  $r$ , emetti la stringa  $v$ .

Per una stringa sorgente  $x$ , la traduzione  $\tau(x)$  realizzata da  $T$  è il concatenamento di due stringhe, prodotte dalla funzione di uscita e da quella finale:

$$\begin{aligned} \tau(x) = \{ &yz \in \Delta^* \mid \exists \text{ un calcolo etichettato } \frac{x}{y} \text{ terminante nello stato } r \in F \\ &\wedge z = \varphi(r, \dashv) \} \end{aligned}$$

<sup>6</sup>Questa terminologia [42] non è assentata; altri [8] chiamano sotto-sequenziale questo tipo di traduttore.

La macchina è deterministica: infatti l'automa d'entrata  $\langle Q, \Sigma, \delta, q_0, F \rangle$  soggiacente a  $T$  è deterministico e inoltre la funzione d'uscita e quella finale sono a un solo valore.

Si noti che, pur se l'automa soggiacente è deterministico, se tra due stati di  $T$  vi fosse l'arco  $\frac{a}{\{b\} \cup \{c\}}$ , il comportamento dell' *IO*-automa non sarebbe deterministico.

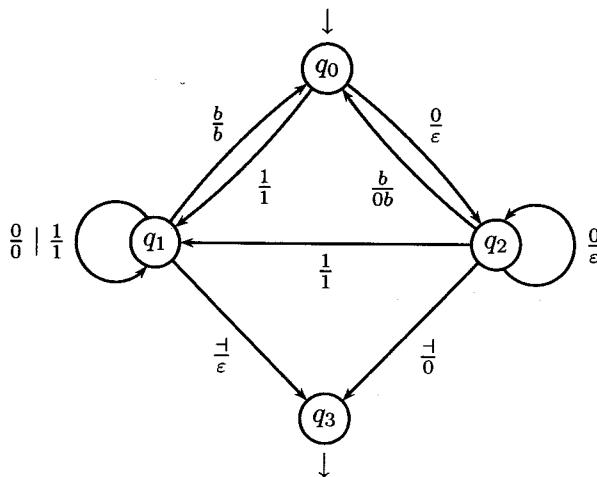
Si dice *funzione sequenziale* una funzione calcolabile mediante un traduttore sequenziale.

*Esempio 5.10.* Zeri non significativi.

Un testo è una lista di numeri binari interi, separati da uno spazio bianco ( $b$ ). La traduzione, evidentemente univoca, sopprime gli zeri non significativi. La e.r.t. è:

$$\left( \left( \frac{0^+}{0} \mid \left( \frac{0}{\varepsilon} \right)^* \frac{1}{1} \left( \frac{0}{0} \mid \frac{1}{1} \right)^* \right) \frac{b}{b} \right)^* \left( \frac{0^+}{0} \mid \left( \frac{0}{\varepsilon} \right)^* \frac{1}{1} \left( \frac{0}{0} \mid \frac{1}{1} \right)^* \right) \frac{\dashv}{\varepsilon}$$

Il traduttore deterministico sequenziale è:



Il calcolo della traduzione di  $00b01 \dashv$  attraversa gli stati  $q_0 q_2 q_2 q_0 q_2 q_1 q_3$  e scrive  $\varepsilon. \varepsilon. 0b. \varepsilon. 1. \varepsilon = 0b1$ .

L'esempio non sfrutta il concetto di funzione finale offerto dal modello di traduttore sequenziale. Per scorgerne l'utilità, si pensi alla funzione di traduzione seguente:

$$\tau(a^n) = \begin{cases} p, & n \text{ pari}, \\ d, & n \text{ dispari}. \end{cases}$$

Il traduttore sequenziale ha due stati, entrambi finali, corrispondenti alla classe di parità della stringa sorgente. Esso non scrive nulla nell'effettuare le mosse, poi, a seconda dello stato finale raggiunto al termine della lettura, emette  $p$  o  $d$ .

È infine interessante notare che la composizione di due funzioni sequenziali è ancora una funzione sequenziale.

### Traduzioni sequenziali a due passate riflesse

Data una funzione di traduzione, specificata per mezzo d'una espressione regolare o d'un automa a due ingressi, non sempre esiste un traduttore sequenziale, cioè una macchina deterministica, che la calcola direttamente. Tuttavia un risultato teorico afferma che la funzione può essere sempre calcolata operando in due passate deterministiche: prima si traduce la stringa sorgente con un traduttore sequenziale, ottenendo una stringa intermedia. Essa è poi tradotta nella stringa pozzo desiderata, per mezzo d'un secondo traduttore sequenziale, che però scandisce la stringa intermedia da destra a sinistra.

*Esempio 5.11.* Traduzione regolare in due passate (es. 5.8 p. 269).

Si ricorda che la traduzione della stringa  $a^n$  in  $b^n$ , se  $n$  è pari, in  $c^n$ , se  $n$  è dispari, non può essere calcolata deterministicamente da un traduttore sequenziale, perché soltanto al termine della lettura si può conoscere la classe di parità della stringa, quando è troppo tardi per emettere la traduzione. Si mostra come calcolare la traduzione con due passate sequenziali operanti in senso inverso.

Il primo traduttore sequenziale calcola la traduzione intermedia

$$\tau_1 = \left( \frac{a}{a'} \frac{a}{a''} \right)^* \left[ \frac{a}{a'} \right]$$

che trasforma in  $a'$  (risp. in  $a''$ ) le  $a$  di posto dispari e pari. L'ultimo termine può mancare.

Il secondo traduttore sequenziale legge la stringa intermedia dal fondo, e calcola la traduzione

$$\tau_2 = \left( \frac{a''}{b} \frac{a'}{b} \right)^* \cup \left( \frac{a'}{c} \frac{a''}{c} \right)^* \frac{a'}{c}$$

dove la scelta tra le alternative è controllata dal primo carattere della stringa intermedia riflessa. Si ha ad es.:

$$\tau_2((\tau_1(aa))^R) = \tau_2((a'a'')^R) = \tau_2(a''a') = bb$$

In molte applicazioni delle traduzioni regolari, la capacità di calcolo dei traduttori sequenziali è adeguata al compito da svolgere. Quando, come nell'esempio

precedente, è necessario operare in due passate, è talvolta possibile assorbire la seconda passata, operante sulla stringa intermedia riflessa, nella prima, ricorrendo all'espedito della prospezione, il concetto estesamente sfruttato nei parsificatori. L'idea è di incorporare nel traduttore la possibilità di esplo- rare in avanti la stringa sorgente prima di decidere quale stringa emettere. Il modello dei traduttori con prospezione è praticamente realizzato in strumenti di progetto assai diffusi, come *lex* e *flex*.<sup>7</sup>

## 5.5 Traduzioni sintattiche pure

Le traduzioni regolari finora studiate corrispondono alle trasformazioni svolte da un algoritmo deterministico con memoria finita, che esamina la stringa sorgente in una passata (o due). Certo la finitezza della memoria costituisce un limite inaccettabile per molti tipi di traduzioni che si richiedono nell'informatica.

L'esempio forse più banale è la riflessione d'una stringa. La relazione  $\{(x, x^R) \mid x \in (a \mid b)^*\}$  non è regolare, come si comprende agevolmente con due ragionamenti. Primo, è necessario immagazzinare la stringa sorgente in una memoria illimitata, prima di poter emettere la sua immagine. Secondo, nell'ottica del teorema di Nivat (ramo 3), se si concatenano tra di loro la stringa sorgente  $x$  e la stringa pozzo  $y = (x')^R$  traslitterata nell'alfabeto  $\{a', b'\}$  disgiunto da quello sorgente, l'insieme delle stringhe così ottenute non è regolare ma libero. Questa e altre traduzioni interessanti si possono immediatamente definire con il prossimo modello delle grammatiche o schemi di traduzione.

Quando il linguaggio sorgente è definito da una grammatica è naturale considerare delle traduzioni in cui le strutture sintattiche, cioè i sottoalberi d'una frase, siano tradotte individualmente, le singole traduzioni componendosi poi per produrre l'immagine dell'intera frase. Una siffatta traduzione strutturale sarà ora definita mediante uno schema che mette in corrispondenza le regole della grammatica sorgente con quelle della grammatica pozzo.

**Definizione 5.12.** Una grammatica di traduzione  $G = (V, \Sigma, \Delta, P, S)$  è una grammatica libera avente come alfabeto terminale un insieme  $C \subseteq \Sigma^* \times \Delta^*$  di coppie  $(u, v)$ , solitamente scritte  $\frac{u}{v}$ , di stringhe sorgente e pozzo.

Lo schema di traduzione sintattica associato alla grammatica è un insieme di coppie di regole sintattiche sorgente e pozzo, ottenute eliminando dalle regole della grammatica  $G$  rispettivamente i caratteri dell'alfabeto pozzo e dell'alfabeto sorgente. L'insieme delle regole sorgente costituisce la grammatica sorgente  $G_1$  soggiacente alla traduzione, e analogamente per la grammatica pozzo soggiacente  $G_2$ .

La relazione di traduzione  $\rho(G)$  definita dalla grammatica è

$$\rho(G) = \{(x, y) \mid \exists z \in L(G) \wedge x = h_\Sigma(z) \wedge y = h_\Delta(z)\}$$

<sup>7</sup>Per la base teorica degli automi con prospezione si veda Yang [54].

dove  $h_{\Sigma} : C \rightarrow \Sigma$  e  $h_{\Delta} : C \rightarrow \Delta$  sono le proiezioni dall'alfabeto terminale della grammatica agli alfabeti sorgente e pozzo rispettivamente.

Una traduzione così definita è detta libera (dal contesto) o anche algebrica.<sup>8</sup>

Lo schema e la grammatica di traduzione sono mere varianti notazionali che definiscono la stessa relazione di traduzione. Intuitivamente ogni coppia di stringhe corrispondenti sorgente e pozzo è ottenuta partendo dalla stessa frase  $z$  e proiettandola sui due alfabeti.

*Esempio 5.13.* Grammatica di traduzione per la riflessione.

Una stringa come  $aab$  si traduce nella riflessa  $baa$ . Grammatica di traduzione  $G$ :

$$S \rightarrow \frac{a}{\varepsilon} S \frac{\varepsilon}{a} \mid \frac{b}{\varepsilon} S \frac{\varepsilon}{b} \mid \frac{\varepsilon}{\varepsilon}$$

Lo schema di traduzione associato è:

|   |  |
|---|--|
| <i>Grammatica sorgente <math>G_1</math></i> | <i>Grammatica pozzo <math>G_2</math></i> |
| $S \rightarrow aS$                          | $S \rightarrow Sa$                       |
| $S \rightarrow bS$                          | $S \rightarrow Sb$                       |
| $S \rightarrow \varepsilon$                 | $S \rightarrow \varepsilon$              |

La due colonne contengono le grammatiche soggiacenti, sorgente e pozzo. Ogni riga mostra le *regole corrispondenti*.

Per ottenere una coppia di stringhe della relazione di traduzione si costruisce una derivazione come

$$S \Rightarrow \frac{a}{\varepsilon} S \frac{\varepsilon}{a} \Rightarrow \frac{a}{\varepsilon} \frac{a}{\varepsilon} S \frac{\varepsilon}{a} \frac{\varepsilon}{a} \Rightarrow \frac{a}{\varepsilon} \frac{a}{\varepsilon} \frac{b}{\varepsilon} S \frac{\varepsilon}{b} \frac{\varepsilon}{a} \frac{\varepsilon}{a} \Rightarrow \frac{a}{\varepsilon} \frac{a}{\varepsilon} \frac{b}{\varepsilon} \frac{\varepsilon}{b} \frac{\varepsilon}{a} \frac{\varepsilon}{a} = z$$

e si proietta la frase  $z$  sui due alfabeti:

$$h_{\Sigma}(z) = aab, \quad h_{\Delta}(z) = baa$$

In alternativa, se si usa lo schema di traduzione per costruire una coppia di stringhe corrispondenti, occorre generare una frase sorgente mediante la  $G_1$  e la frase pozzo mediante la  $G_2$ , avendo l'avvertenza di usare a ogni passo della derivazione due regole corrispondenti nello schema.

Il lettore avrà notato che la precedente grammatica di traduzione è quasi identica a quella dei palindromi, la quale, differenziando i caratteri terminali della prima e della seconda metà, si scrive come  $G_p = \{S \rightarrow aSa' \mid bSb' \mid \varepsilon\}$ . Traslitterando  $\frac{a}{\varepsilon}$  in  $a$ ,  $\frac{b}{\varepsilon}$  in  $b$ ,  $\frac{\varepsilon}{a}$  in  $a'$  e  $\frac{\varepsilon}{b}$  in  $b'$ , le due grammatiche coincidono. L'osservazione porta alla seguente proprietà, che sta alle traduzioni libere come il teorema di Nivat a quelle regolari.

*Proprietà 5.14. Linguaggio libero e traduzione libera.*

Le seguenti condizioni sono equivalenti:

---

<sup>8</sup>In passato tali traduzioni erano dette *simple syntax-directed translations*.

1. La relazione di traduzione  $\rho \subseteq \Sigma^* \times \Delta^*$  è definita da una grammatica di traduzione  $G$ .

2. Esiste un alfabeto  $\Omega$ , un linguaggio libero  $L$  su tale alfabeto e due omomorfismi alfabetici  $h_1 : \Omega \rightarrow \Sigma$  e  $h_2 : \Omega \rightarrow \Delta$  tali che

$$\rho = \{(h_1(z), h_2(z)) \mid z \in L\}$$

3. Se i due alfabeti  $\Sigma$  e  $\Delta$  sono disgiunti, esiste un linguaggio libero  $L$  di alfabeto  $\Sigma \cup \Delta$  tale che

$$\rho = \{(h_\Sigma(z), h_\Delta(z)) \mid z \in L\}$$

dove  $h_\Sigma$  e  $h_\Delta$  sono rispettivamente le proiezioni dall'alfabeto  $\Sigma \cup \Delta$  agli alfabeti sorgente e pozzo.

*Esempio 5.15.* Si esemplifica di nuovo con la traduzione della stringa  $x \in (a \mid b)^*$  nella sua riflessa  $y = x^R$ . La traduzione si esprime sfruttando il linguaggio libero di alfabeto  $\Omega = \{a, b, a', b'\}$ , molto simile a quello dei palindromi, seguente:

$$L = \{u(u^R)' \mid u \in (a \mid b)^*\} = \{\varepsilon, aa', \dots, abbb'b'a', \dots\}$$

dove  $(v)'$  denota la copia accentata di  $v$ . Gli omomorfismi alfabetici sono:

|      | $h_1$         | $h_2$         |
|------|---------------|---------------|
| $a$  | $a$           | $\varepsilon$ |
| $b$  | $b$           | $\varepsilon$ |
| $a'$ | $\varepsilon$ | $a$           |
| $b'$ | $\varepsilon$ | $b$           |

Così la stringa  $abb'a' \in L$  si traslittera nella coppia di stringhe

$$(h_1(abb'a'), h_2(abb'a')) = (ab, ba)$$

appartenenti alla relazione di traduzione.

### 5.5.1 Scritture infisse e polacche

Un'applicazione delle traduzioni libere è la conversione delle espressioni aritmetiche, logiche, ecc. tra le diverse rappresentazioni utilizzate nell'informatica, che differiscono per la posizione degli operatori e per la presenza o meno di parentesi e di altri delimitatori.

Il grado d'un operatore o funtore è il numero dei suoi argomenti. Un operatore di grado non fisso è detto variadico. Di particolare interesse sono gli operatori di grado due o binari e uno o unari.

Così il confronto per eguaglianza o diseguaglianza è un operatore binario. L'addizione aritmetica è un'operatore di grado  $\geq 1$ ; però nel linguaggio macchina l'operatore add è binario, perché ha due soli argomenti. Inoltre, se per la

somma vale la proprietà associativa, la somma tra tanti argomenti può essere decomposta in una serie di somme binarie, eseguite ad es. da sinistra a destra. La sottrazione aritmetica è un esempio di operatore binario, mentre il cambiamento di segno è un operatore unario. Se si usa lo stesso segno “–” per i due operatori, si ha un operatore variadic.

Un *operatore* è *prefisso* se precede i suoi argomenti; è *postfisso* se li segue. Un operatore binario è *infisso* se sta tra i due argomenti. Il concetto d'un operatore interposto tra gli argomenti si estende anche agli operatori di grado maggiore di due. Un operatore di grado  $n \geq 2$  è *mistofisso*<sup>9</sup> se la sua rappresentazione è spezzata in  $n + 1$  parti

$$o_0 \ arg_1 \ o_1 \ arg_2 \dots o_{n-1} \ arg_n \ o_n$$

ossia se la lista degli argomenti ha una marca di apertura  $o_0$ , poi ( $n - 1$ ) separatori  $o_i$  eguali tra loro o diversi, e infine una marca di chiusura  $o_n$ . Le marche di apertura o chiusura possono mancare.

Ad es. l'operatore condizionale dei linguaggi di programmazione è mistofisso con grado due o tre, se è presente la parte “else”:

$$\text{if } arg_1 \text{ then } arg_2 \text{ [ else } arg_3]$$

A causa del grado variabile, la rappresentazione risulta ambigua, se il secondo argomento può essere a sua volta un condizionale (come visto a p. 55). In certi linguaggi, come ADA, l'aggiunta della marca di fine “end\_if” toglie l'ambiguità.

L'operazione condizionale binaria è spesso rappresentata nel linguaggio macchina in forma prefissa dall'istruzione

$$\text{jump\_if\_false } arg_1 \ arg_2$$

In effetti ogni istruzione macchina è di tipo prefisso, perché inizia con il codice operativo dell'operazione, e di grado fissato dal numero di campi per gli operandi presenti nell'istruzione.

Una rappresentazione è detta *polacca*<sup>10</sup> se non fa uso di parentesi e se gli operatori sono tutti prefissi o tutti postfissi. La semplicissima grammatica delle espressioni polacche è apparsa a p. 50.

Il prossimo esempio mostra una traduzione frequentemente impiegata nei compilatori allo scopo di eliminare le parentesi, la conversione d'una espressione aritmetica dalla notazione infissa a quella polacca.

La scrittura d'una grammatica di traduzione si alleggerisce se gli alfabeti sorgente e pozzo non sono sovrapposti, perché allora i terminali possono essere scritti semplicemente nelle regole, senza la linea di frazione.

<sup>9</sup>Traduzione di *mixfix*.

<sup>10</sup>Dalla nazionalità del suo inventore, il logico Jan Lukasiewicz che la propose per abbreviare le formule matematiche.

**Esempio 5.16.** Operatori infissi e prefissi.

Del linguaggio sorgente fanno parte le espressioni aritmetiche con addizioni e moltiplicazioni (operatori infissi), parentesi e il terminale  $i$  denotante un identificatore di variabile. La traduzione produce la forma polacca prefissa: gli operatori diventano prefissi, le parentesi scompaiono e gli identificatori si traslitterano in  $i'$ .

Alfabeti:

$$\Sigma = \{+, \times, (,), i\} \quad \Delta = \{\text{add}, \text{mult}, i'\}$$

Grammatica di traduzione:

$$E \rightarrow \text{add } T + E \mid T \quad T \rightarrow \text{mult } F \times T \mid F \quad F \rightarrow (E) \mid ii'$$

(Si noti che  $E \rightarrow \text{add } T + E$  abbrevia  $E \rightarrow \frac{e}{\text{add}} T \frac{e}{+} E$ ).

Lo schema di traduzione associato è:

*Grammatica sorgente  $G_1$*

$$E \rightarrow T + E$$

$$E \rightarrow T$$

$$T \rightarrow F \times T$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow i$$

*Grammatica pozzo  $G_2$*

$$E \rightarrow \text{add } TE$$

$$E \rightarrow T$$

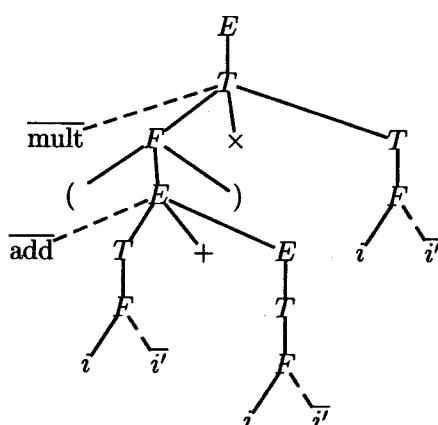
$$T \rightarrow \text{mult } FT$$

$$T \rightarrow F$$

$$F \rightarrow E$$

$$F \rightarrow i'$$

Un esempio di traduzione è mostrato nell'albero sintattico, dove i terminali pozzo sono soprallineati.



Cancellando la parte tratteggiata dell'albero si ottiene l'albero sintattico della frase sorgente  $(i + i') \times i$ , che sarebbe calcolato dal parsificatore usando la grammatica sorgente  $G_1$ . Dualmente, eliminando dall'albero le foglie dell'alfabeto sorgente, si ottiene un albero generato dalla grammatica pozzo per la stringa immagine:  $\text{mult add } i' i' i'$ .

### Costruzione dell'albero sintattico pozzo

Nello schema di traduzione le regole delle grammatiche soggiacenti sono in corrispondenza biunivoca. Si noti che due regole corrispondenti nello schema hanno la stessa parte sinistra e nella parte destra i simboli nonterminali sono nello stesso ordine.

Data una grammatica di traduzione, per calcolare l'immagine d'una frase sorgente  $x$ , si può dunque procedere nel modo seguente. Si esegue l'analisi sintattica, con la grammatica sorgente  $G_1$ , ottenendo l'albero sintattico  $t_x$  di  $x$  (unico se la frase non è ambigua). Si attraversa l'albero  $t_x$ , in un ordine di visita prefissato, ad es. in ordine anticipato (preordine). A ciascun passo della visita, se il nodo corrente dell'albero sorgente usa una certa regola di  $G_1$ , si applica la corrispondente regola della grammatica pozzo  $G_2$ , per costruire un pezzo dell'albero pozzo. Al termine della visita l'albero pozzo è completo.

### Alberi sintattici astratti

Le traduzioni sintattiche sono un buon modo per trasformare gli alberi sintattici del linguaggio sorgente, al fine di togliere quei particolari della sintassi concreta del linguaggio, che sono inutili nella traduzione, attuando così un'astrazione linguistica (p. 25). Un caso è stato esposto: l'eliminazione delle parentesi dalle espressioni aritmetiche. Non è difficile immaginarne altri, come, per le liste a uno o più livelli, la soppressione o sostituzione dei separatori tra gli elementi; o infine l'eliminazione degli operatori mistofissi nelle frasi condizionali if then else end\_if.

#### 5.5.2 Ambiguità della grammatica sorgente e della traduzione

Di solito le grammatiche o schemi di traduzione di interesse pratico sono quelli che definiscono una funzione a un solo valore. Se la grammatica sorgente è ambigua, una frase sorgente ammette due alberi sintattici diversi, a ciascuno dei quali corrisponde un albero sintattico pozzo. La frase avrà dunque due immagini, generalmente diverse.

##### *Esempio 5.17. Parentesizzazione ridondante.*

Un caso di traduzione non univoca si presenta nella traduzione d'una espressione aritmetica dalla forma polacca prefissa (o postfissa) alla forma infissa con parentesi.

Per scrivere lo schema di traduzione, basta considerare la traduzione inversa di quella dell'es. 5.16 di p. 277, ottenuta scambiando le grammatiche sorgente e pozzo.

Presa dunque  $G_2$  come grammatica sorgente, si nota che essa è ambigua, poiché ammette la derivazione circolare

$$E \Rightarrow T \Rightarrow F \Rightarrow E$$

Ad es. la frase sorgente  $i'$  può essere derivata in più modi:

$$E \Rightarrow T \Rightarrow F \Rightarrow i', \quad E \Rightarrow T \Rightarrow F \Rightarrow E \Rightarrow T \Rightarrow F \Rightarrow i', \quad \dots$$

ai quali corrispondono altrettante traduzioni tutte diverse:

$$E \Rightarrow T \Rightarrow F \Rightarrow i, \quad E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (T) \Rightarrow (F) \Rightarrow (i), \quad \dots$$

Il fatto si spiega facilmente perché, nella traduzione da prefisso a infisso, si possono inserire delle parentesi, ma la grammatica di traduzione non ne prescrive il numero, e permette l'aggiunta di parentesi ridondanti.

Viceversa, se la grammatica sorgente non è ambigua, l'albero sintattico di ogni frase sorgente è unico, ma può succedere che la traduzione non sia univoca, quando nella grammatica di traduzione due o più regole si mappano sulla stessa regola della grammatica sorgente. Un esempio è la grammatica di traduzione:

$$S \rightarrow \frac{a}{b} S \mid \frac{a}{c} S \mid \frac{a}{d}$$

dove, pur non essendo  $G_1 = \{S \rightarrow aS \mid a\}$  ambigua, la traduzione  $\tau(aa) = \{bd, cd\}$  ha più valori, perché le prime due regole della grammatica di traduzione portano alla stessa regola sorgente.

*Proprietà 5.18.* Sia  $G$  una grammatica di traduzione tale che

1. la grammatica sorgente  $G_1$  dello schema non è ambigua, e
2. ogni regola di  $G_1$  corrisponde a una sola regola di  $G$ .

Allora la traduzione definita da  $G$  è univoca, ossia è una funzione di traduzione a un solo valore.

Nelle precedenti considerazioni sull'univocità della traduzione, l'ambiguità della grammatica di traduzione non è stata considerata. Ma è immediato osservare che, se  $G$  fosse ambigua, anche la grammatica sorgente soggiacente  $G_1$  lo sarebbe. Infatti, si immagini una frase ambigua  $z \in L(G)$ , con i suoi diversi alberi sintattici. Proiettando gli alberi sull'alfabeto sorgente  $\Sigma$ , si ottengono alberi sorgente diversi della stessa stringa  $h_1(z) \in L(G_1)$ .

Non vale il viceversa: il prossimo esempio mostra una grammatica di traduzione inambigua la cui grammatica sorgente è ambigua.

*Esempio 5.19.* Marca di fine nei condizionali.

La traduzione aggiunge la marca di chiusura "end\_if" in fondo alle istruzioni if ... then ... [ else]. La grammatica di traduzione  $G$

$$S \rightarrow \frac{\text{if } c \text{ then}}{\text{if } c \text{ then end_if}} S \frac{\varepsilon}{\text{else}} \mid \frac{\text{if } c \text{ then}}{\text{if } c \text{ then end_if}} S \frac{\varepsilon}{\text{else}} \mid a$$

non è ambigua, ma la grammatica sorgente, che definisce i condizionali senza marca di chiusura, è nota (da p. 55) per essere ambigua.

In conclusione, nel progetto dei compilatori, si eviterà l'uso di grammatiche di traduzione che possano causare perdita d'univocità della traduzione.

### 5.5.3 Grammatica di traduzione e traduttore a pila

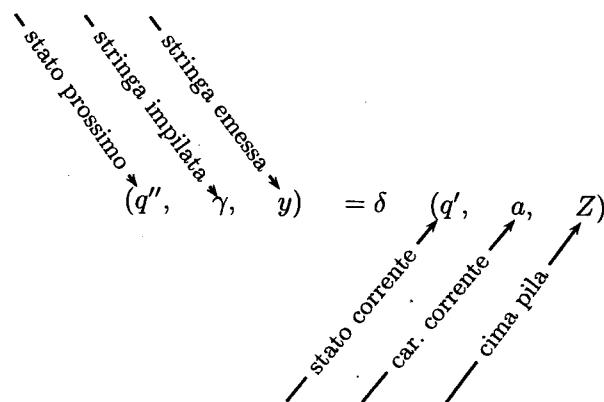
Il concetto astratto d'una *IO-macchina*, che riconosce la stringa sorgente e la traduce nella sua immagine, vale evidentemente anche per le traduzioni libere. Come per il riconoscimento delle frasi dei linguaggi liberi occorre una pila, così per eseguire le traduzioni definite da una grammatica di traduzione, occorre dotare l'automa traduttore d'una memoria illimitata, con accesso LIFO.

Un *traduttore o IO-automa a pila* non è altro che un automa a pila, arricchito della capacità di emettere uno o più caratteri *pozzo* in ogni mossa.

Per definire un *traduttore a pila* occorrono otto entità:

- $Q$ , insieme degli stati;
- $\Sigma$ , alfabeto sorgente;
- $\Gamma$ , alfabeto della pila;
- $\Delta$ , alfabeto pozzo;
- $\delta$ , funzione di transizione e d'uscita;
- $q_0 \in Q$ , stato iniziale;
- $Z_0 \in \Gamma$ , simbolo iniziale della pila;
- $F \subseteq Q$ , insieme degli stati finali.

La funzione  $\delta$  ha come dominio di definizione  $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$  e come codominio<sup>11</sup> l'insieme  $Q \times \Gamma^* \times \Delta^*$ . Essa ha questo significato: se  $(q'', \gamma, y) = \delta(q', a, Z)$ , il traduttore, dallo stato presente  $q'$ , avendo letto  $a$  dal nastro d'ingresso e  $Z$  dalla pila, si porta nello stato  $q''$ , scrive  $\gamma$  in pila e  $y$  in uscita:



Gli stati finali coincidono con l'insieme  $Q$ , nel caso frequente in pratica, che il riconoscimento avvenga a pila vuota.

Come nel caso dei traduttori finiti, l'*automa soggiacente al traduttore* è quello ottenuto semplicemente cancellando dalla definizione l'alfabeto pozzo e la componente di uscita della funzione.

<sup>11</sup>Come nei traduttori finiti sequenziali (p. 270), si potrebbe descrivere l'emissione dei caratteri mediante una funzione separata di uscita. Il codominio della funzione  $\delta$  diverrebbe l'insieme delle parti finite del prodotto cartesiano indicato, se si considerassero automi indeterministici, caso che sarà trattato solo intuitivamente.

La traduzione calcolata da un traduttore a pila si formalizza in modo del tutto analogo al caso dei traduttori finiti, e per brevità non si ripete la definizione.

### Dalla grammatica di traduzione al traduttore a pila

#### GRAMMATICA DI TRADUZIONE

Gli schemi sintattici di traduzione e i traduttori a pila sono due modi di descrivere una relazione di traduzione: il primo è di tipo generativo, il secondo di tipo procedurale. L'equivalenza dei due modelli è affermata nel seguente enunciato.

Proprietà 5.20. Una relazione di traduzione è definita da una grammatica di traduzione (o schema sintattico) se, e solo se, essa è calcolata da un traduttore a pila.

Per brevità si espone soltanto il passaggio dallo schema di traduzione al traduttore, perché il passaggio inverso ha minore interesse.

Si consideri una grammatica di traduzione  $G_t$ . Il traduttore a pila  $T$  si ottiene costruendo con il procedimento noto (alg. 4.1, p. 148) l'automa a pila che riconosce il linguaggio caratteristico  $L(G_t)$ , e poi trasformando la macchina in traduttore mediante una piccola modifica che riguarda i caratteri pozzo. Questi, dopo essere stati inseriti nella pila allo stesso modo dei caratteri sorgente, quando riaffiorano in cima, vanno scritti sul nastro d'uscita (senza confrontarli con il carattere corrente d'ingresso).

#### Normalizzazione delle regole di traduzione

Per semplificare la costruzione del traduttore, senza perdita di generalità, conviene raggruppare le stringhe sorgente e pozzo presenti nelle regole in modo tale che, dove possibile, il primo carattere appartenga sempre all'alfabeto sorgente.

Più precisamente si fanno le seguenti ipotesi sulle coppie  $\frac{u}{v}, u \in \Sigma^*, v \in \Delta^*$ , presenti nelle regole della grammatica di traduzione:

1. In ogni coppia  $\frac{u}{v}$ , è  $|u| \leq 1$ , ossia  $u$  è un singolo carattere  $a \in \Sigma$ , o la stringa vuota.

Ciò non è limitativo: infatti, se vi fosse un elemento  $\frac{a_1 a_2}{v}$ , potrebbe essere sostituito da  $\frac{a_1}{v} \frac{a_2}{\epsilon}$ , senza alterare la traduzione.

2. Una regola non può contenere i diagrammi

$$\frac{\epsilon \quad a}{v_1 \quad v_2} \qquad \frac{\epsilon \quad \epsilon}{v_1 \quad v_2}$$

dove  $v_1, v_2 \in \Delta^*$ . Infatti tali sottostringhe, se presenti, possono essere rispettivamente riscritte come  $\frac{a}{v_1 v_2}$  o come  $\frac{\epsilon}{v_1 v_2}$ , senza che la relazione di traduzione cambi.

Si formalizza ora la corrispondenza tra le regole della grammatica di traduzione  $G_t = (V, \Sigma, \Delta, P, S)$  e le mosse del traduttore.

Algoritmo 5.21. Costruzione del traduttore a pila predittivo indeterministico. Sia  $C$  l'insieme delle coppie dei tipi  $\frac{\epsilon}{v}, v \in \Delta^+$  e  $\frac{b}{w}, b \in \Sigma, w \in \Delta^*$ , presenti nelle regole della grammatica di traduzione. La seguente tabella dà le istruzioni per la costruzione delle mosse dell'automa:

|   | <i>Regola</i>  | <i>Mossa</i>  | <i>Commento</i>  |
|---|--|---|--|
| 1 | $A \rightarrow \frac{\epsilon}{v} BA_1 \dots A_n$<br>$n \geq 0,$<br>$v \in \Delta^+, B \in V,$<br>$A_i \in (C \cup V)$ | if $cima = A$ then<br>write( $v$ );<br>pop;<br>push( $A_n \dots A_1 B$ );   | Emetti la stringa pozzo<br>$v$ e impila la predizione<br>$BA_1 \dots A_n$  |
| 2 | $A \rightarrow \frac{b}{w} A_1 \dots A_n$<br>$n \geq 0,$<br>$b \in \Sigma, w \in \Delta^*,$<br>$A_i \in (C \cup V)$    | if $car\_corr = b \wedge cima = A$ then<br>write( $w$ );<br>pop;<br>push( $A_n \dots A_1$ );<br>avanza testina lettura; | $b$ era il primo carattere<br>atteso ed è stato letto;<br>emetti la stringa pozzo<br>$w$ ; impila la predizione<br>$A_1 \dots A_n$ |
| 3 | $A \rightarrow BA_1 \dots A_n$<br>$n \geq 0, B \in V,$<br>$A_i \in (C \cup V)$   | if $cima = A$ then pop;<br>push( $A_n \dots A_1$ );   | impila la predizione<br>$A_1 \dots A_n$  |
| 4 | $A \rightarrow \frac{\epsilon}{v}$<br>$v \in \Delta^+$   | if $cima = A$ then<br>write( $v$ ); pop;  | emetti la stringa pozzo<br>$v$   |
| 5 | $A \rightarrow \epsilon$   | if $cima = A$ then pop;   |  |
| 6 | per ogni coppia<br>$\frac{\epsilon}{v} \in C$  | if $cima = \frac{\epsilon}{v}$ then<br>write( $v$ ); pop;   | la passata previsione $\frac{\epsilon}{v}$<br>si attua scrivendo $v$   |
| 7 | per ogni coppia<br>$\frac{b}{w} \in C$   | if $car\_corr = b \wedge cima = \frac{b}{w}$ then<br>write( $w$ ); pop; avanza<br>testina lettura;                      | la passata previsione $\frac{b}{w}$<br>si attua leggendo $b$ e<br>scrivendo $w$  |
| 8 | ---  | if $car\_corr = -1 \wedge$ pi-<br>la è vuota then accetta;<br>alt;  | la stringa sorgente è<br>stata scandita per in-<br>tero e non restano<br>obiettivi in agenda                                       |

Le righe 1,2,3,4,5 si applicano quando la cima della pila è un simbolo nonterminale. Nel caso 2 la parte destra inizia con un terminale sorgente, e la mossa è subordinata alla lettura dello stesso. Altrimenti, le righe 1, 3 , 4, 5 danno luogo a mosse spontanee, che non leggono il carattere corrente.

Le righe 6 e 7 si applicano quando una coppia affiora sulla cima della pila. Se la coppia contiene un carattere sorgente (7), questo deve coincidere con il carattere corrente. Se la coppia contiene una stringa pozzo (6,7), la si scrive

in uscita.

Inizialmente la pila contiene soltanto l'assioma  $S$ , e la testina di lettura è posta sul primo carattere della stringa sorgente. A ogni passo l'automa sceglie (indeterministicamente) una delle regole applicabili e esegue la corrispondente mossa. Infine la riga 8 accetta la stringa, se la pila è vuota alla lettura del terminatore.

Si noti che l'automa non usa stati diversi, cioè la pila è l'unica memoria, ma come per i riconoscitori, sarà necessario aggiungere gli stati, al fine di rendere deterministica la macchina.

*Esempio 5.22.* Traduttore a pila indeterministico.

Si estende il riconoscitore (es. 4.7, p. 156) del linguaggio

$$L = \{a^* a^m b^m \mid m > 0\}$$

allo scopo di calcolare la traduzione

$$\tau(a^k a^m b^m) = d^m c^k$$

che ricopia come  $d$  le lettere  $b$  e poi trasforma in  $c$  le lettere  $a$  eccedenti il numero delle  $b$ .

Il traduttore ha le mosse sotto riportate, accanto alle regole della grammatica di traduzione:

|   | <i>Regola</i>   | <i>Mossa</i>   |
|---|---|--|
| 1 | $S \rightarrow \frac{a}{\epsilon} S \frac{\epsilon}{c}$ | if $car\_corr = a \wedge cima = S$ then pop; push( $\frac{\epsilon}{c} S$ ); avanza testina lettura;               |
| 2 | $S \rightarrow A$                                       | if $cima = S$ then pop; push( $A$ );   |
| 3 | $A \rightarrow \frac{a}{d} A \frac{b}{\epsilon}$        | if $car\_corr = a \wedge cima = A$ then pop; write( $d$ ); push( $\frac{b}{\epsilon} A$ ); avanza testina lettura; |
| 4 | $A \rightarrow \frac{a}{d} \frac{b}{\epsilon}$          | if $car\_corr = a$ and $cima = A$ then pop; write( $d$ ); push( $\frac{b}{\epsilon}$ ); avanza testina lettura;    |
| 5 | —   | if $cima = \frac{\epsilon}{c}$ then pop; write( $c$ );   |
| 6 | —   | if $car\_corr = b \wedge cima = \frac{b}{\epsilon}$ then pop; avanza testina lettura;                              |
| 7 | —   | if $car\_corr = - \wedge$ pila è vuota then accetta; alt;  |

La scelta tra le mosse 1 e 2 non è deterministica, e similmente la scelta tra 3 e 4. La mossa 5 scrive un carattere pozzo precedentemente impilato dalla mossa 1. L'automa riconoscitore soggiacente è esattamente quello dell'es. 4.7.

Ricordando che già per le traduzioni regolari non sempre esiste un traduttore finito deterministico capace di realizzarle, si riconferma tale situazione anche per le traduzioni libere: non tutte le relazioni definite da una grammatica di traduzione possono essere calcolate da un traduttore a pila deterministico.

*Esempio 5.23.* Traduzione libera non deterministica.

La funzione di traduzione

$$\tau(u) = u^R u, \quad u \in \{a, b\}^*$$

è facilmente definita dallo schema sintattico:

| <i>Grammatica sorgente <math>G_1</math></i> | <i>Grammatica pozzo <math>G_2</math></i> |
|---|--|
| $S \rightarrow Sa$                          | $S \rightarrow aSa$                      |
| $S \rightarrow Sb$                          | $S \rightarrow bSb$                      |
| $S \rightarrow \epsilon$                    | $S \rightarrow \epsilon$                 |

La traduzione non può essere calcolata da un traduttore a pila deterministico. La giustificazione<sup>12</sup> è che la macchina deve emettere per prima la copia riflessa della stringa sorgente. Per calcolare la riflessione, si deve però impilare la stringa  $\epsilon$ , dopo la lettura del terminatore, disimpilarla sul nastro d'uscita. Ma dopo tale azione, la pila sarà vuota, la macchina avrà perso ogni informazione sulla stringa sorgente e non potrà scriverla in uscita.

Nella pratica, basta studiare il caso deterministico, per il quale si presentano ora gli algoritmi di traduzione, appropriati al tipo di analisi sintattica adottato.

#### 5.5.4 Analisi sintattica e traduzione in linea

Specificata una traduzione mediante uno schema sintattico, si è visto come costruire un traduttore a pila per calcolare la stringa immagine d'una frase sorgente. La macchina ottenuta è quasi sempre indeterministica, anche quando la specifica consentirebbe un calcolo deterministico della traduzione.

Per costruire dei programmi traduttori efficienti, si sfrutterà ora il percorso concettuale che nel cap. 4 ha condotto agli algoritmi di costruzione dei parsificatori deterministici. Si esamina dunque il modo di calcolare la traduzione, trasformando un parsificatore deterministico del linguaggio sorgente in un algoritmo di traduzione.

Data una grammatica o schema di traduzione, si suppone ora che la grammatica sorgente permetta la costruzione d'un parsificatore deterministico. Per calcolare la traduzione, si esegue l'analisi sintattica e via via che si costruisce un sottoalbero se ne emette la traduzione.

È noto che gli analizzatori ascendenti e discendenti differiscono nell'ordine di costruzione dell'albero. Ci si può domandare se l'ordine abbia qualche conseguenza per il calcolo della traduzione. [Si vedrà che l'analisi discendente non pone vincoli sul calcolo della traduzione, mentre l'analisi ascendente porta a una condizione restrittiva sulla forma delle regole della grammatica di traduzione.]

#### 5.5.5 Traduzioni deterministiche discendenti

La grammatica di traduzione  $G_t$  può essere rappresentata con una rete ricorsiva di macchine, allo stesso modo d'una grammatica. Ovviamente, agli effetti

<sup>12</sup>Per una dimostrazione si veda [2, 3].

del determinismo, è la grammatica sorgente dello schema sintattico, quella da considerare. Se la grammatica sorgente è  $LL(k)$ , il parsificatore deterministico del linguaggio sorgente, completato con le azioni di scrittura, può calcolare efficientemente la traduzione.

Come per i parsificatori, le tecniche costruttive sono due: macchina a pila e procedure ricorsive.

La costruzione del traduttore a pila, mostrata per prima, è una semplice modifica di quella dell'analizzatore sintattico.

Per semplificare l'esposizione, conviene supporre che le regole della grammatica di traduzione siano normalizzate (p. 281).

*Algoritmo 5.24.* Costruisce il traduttore a pila deterministico che calcola la traduzione definita dalla grammatica di traduzione  $G_t = (\Sigma, \Delta, V, P, S)$ , la cui grammatica sorgente sia  $LL(1)$ .

L'alfabeto della pila è costituito dai simboli nonterminali, e dalle coppie  $\frac{b}{v} \in C$ , dove  $b$  è un carattere sorgente o la stringa vuota e  $v$  una stringa pozzo (escludendo che entrambe le componenti siano vuote). La proiezione d'una stringa  $z$  sull'alfabeto sorgente è indicata da  $h_\Sigma(z)$ . Il carattere corrente è nella variabile  $cc$ .

1. L'automa parte con  $S$ , l'assioma, sulla pila.
2. Sia  $A$  in cima alla pila. Per ogni regola  $A \rightarrow \frac{b}{w}\beta$ , dove  $b \in \Sigma, w \in \Delta^*, \beta \in \{V \cup C\}^*$ , la macchina, se  $b$  è il  $cc$ , emette la stringa  $w$ , nella pila sostituisce  $A$  con  $\beta^R$  e fa avanzare la testina di ingresso.
3. Sia  $A$  in cima alla pila. Per ogni regola  $A \rightarrow \frac{\epsilon}{v}\beta$ , dove  $v \in \Delta^*$  e  $\beta \in \{V \cup C\}^*$ , l'insieme guida è quello calcolato per la corrispondente regola della grammatica sorgente, ossia è l'insieme  $Gui(A \rightarrow h_\Sigma(\frac{\epsilon}{v}\beta))$ . L'automa, se  $cc$  appartiene a tale insieme guida, emette la stringa  $v$ , poi nella pila sostituisce  $A$  con  $\beta^R$ , senza spostare la testina di ingresso.
4. Con  $\frac{b}{w}, b \in \Sigma$  in cima alla pila e se  $b$  è il  $cc$ , il traduttore scrive  $w$  e legge il prossimo carattere.
5. L'automa termina la traduzione con successo se il carattere corrente è il terminatore e la pila è vuota.

Si noti che il 3. comprende le regole vuote  $A \rightarrow \epsilon$ . In 3. si usano gli insiemi guida per scegliere la strada; la condizione  $LL(1)$  garantisce il determinismo della scelta.

*Esempio 5.25.* (es. 5.13 continuato).

Una stringa è tradotta nella sua riflessa dalla grammatica di traduzione:

$$S \rightarrow \frac{a}{\varepsilon} S \frac{\varepsilon}{a} \mid \frac{b}{\varepsilon} S \frac{\varepsilon}{b} \mid \varepsilon$$

La grammatica sorgente è  $LL(1)$ , con insiemi guida rispettivi  $\{a\}$ ,  $\{b\}$  e  $\{\vdash\}$ . Le mosse dell'automa sono

| pila                    | $cc = a$                                | $cc = b$                                | $cc = \vdash$ | $\varepsilon$ |
|-------------------------|---|---|---------------|---------------|
| $S$                     | pop; push( $\frac{\varepsilon}{a} S$ ); | pop; push( $\frac{\varepsilon}{b} S$ ); | pop;          |               |
| $\frac{\varepsilon}{a}$ |   |   |               | write( $a$ )  |
| $\frac{\varepsilon}{b}$ |   |   |               | write( $b$ )  |

### Realizzazione del traduttore con procedure ricorsive

Si mostra ora la realizzazione del traduttore a pila con procedure ricorsive, nel caso d'una grammatica di traduzione, estesa con espressioni regolari. Si sa costruire il programma del parsificatore ricorsivo (p. 188), che riconosce il linguaggio sorgente, se la grammatica sorgente è  $LL(k)$ . Si ricorda che, per ogni simbolo nonterminale, vi è una procedura che si incarica di riconoscere le sottostringhe da esso generate. La struttura della procedura riproduce il grafo della macchina che definisce il nonterminale.

Per produrre la traduzione, basta inserire un'istruzione di scrittura, in ogni punto della procedura che corrisponda alla comparsa d'un elemento pozzo nel grafo della macchina della rete.

Basta un esempio per spiegare questa semplice modifica della costruzione del parsificatore.

*Esempio 5.26.* Traduttore ricorsivo da infisso a postfisso.

Il linguaggio sorgente contiene le espressioni aritmetiche con operatori a due livelli di precedenza e con parentesi. La traduzione converte le espressioni nella scrittura postfissa, come sotto illustrato:

$$v \times (v + v) \quad \Rightarrow \quad vvv \text{ add mult}$$

Segue la grammatica di traduzione BNF estesa:

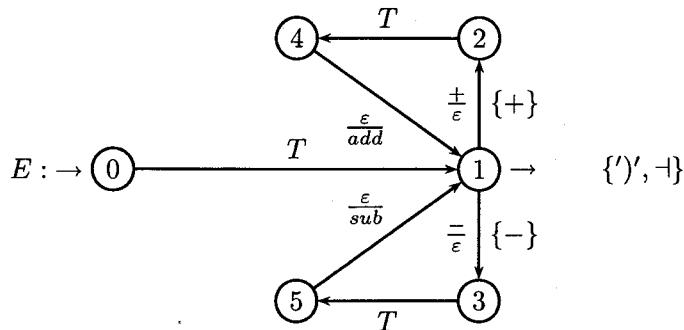
$$E \rightarrow T \left( \frac{+}{\varepsilon} T \frac{\varepsilon}{add} \mid \frac{-}{\varepsilon} T \frac{\varepsilon}{sub} \right)^*$$

$$T \rightarrow F \left( \frac{\times}{\varepsilon} F \frac{\varepsilon}{mult} \mid \frac{\div}{\varepsilon} F \frac{\varepsilon}{div} \right)^*$$

$$F \rightarrow \frac{v}{v} \mid \frac{'('}{\varepsilon} E \frac{')'}{\varepsilon}$$

$$\Sigma = \{+, \times, -, \div, (, ), v\}, \quad \Delta = \{add, sub, mult, div, v\}$$

Si mostra anche una macchina della rete, in cui sono indicati tra graffe gli insiemi guida:



Dal grafo di  $E$  si scrive facilmente la procedura del traduttore.

```

procedure E
call T;
while cc ∈ {+, -}
do
  case
    cc = '+': begin cc := Prossimo; call T; write('add'); end
    cc = '-': begin cc := Prossimo; call T; write('sub'); end
    otherwise Errore
  end case
end do
end ;
  
```

Questo stile di scrittura guadagna in chiarezza sfruttando le istruzioni **while** per realizzare le iterazioni della grammatica.

### 5.5.6 Traduzioni deterministiche ascendenti

Sia dato uno schema di traduzione, tale che la grammatica sorgente consenta la parsificazione ascendente deterministica (non importa se  $LR(k)$  o  $LALR(k)$ ). Si vedrà ora che il parsificatore, arricchito con le azioni di scrittura delle stringhe pozzo, può calcolare la traduzione, ma non in tutti i casi.

La ragione della limitazione è semplice: il riconoscitore opera mediante spostamenti e riduzioni. Uno spostamento inserisce nella pila un macrostato dell'automa pilota, cioè un insieme di stati delle macchine (sorgente) della rete, o, che è lo stesso, un insieme di regole sorgente marcate. Quando dunque esegue uno spostamento, il riconoscitore ancora non sa quale sarà la regola da applicare, perché più candidature sono aperte. Ma due diverse regole marcate, presenti nel macrostato corrente, sono generalmente associate a traduzioni

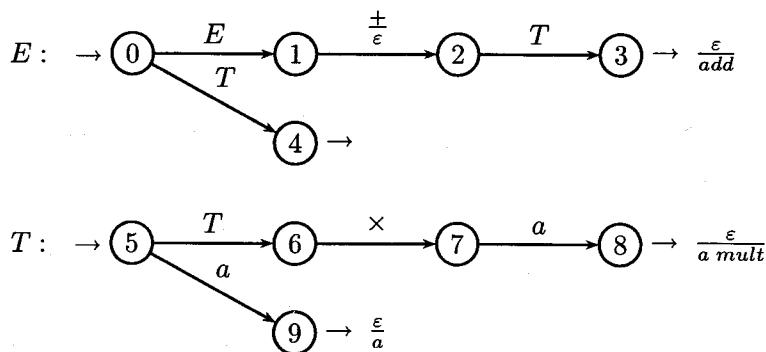
diverse, il che richiederebbe diverse e contradditorie emissioni di caratteri. Questo ragionamento spiega la difficoltà di effettuare l'emissione della traduzione in concomitanza con le mosse di spostamento.

Invece una mossa di riduzione, grazie al determinismo del parsificatore, corrisponde a una, e una sola, regola sorgente (o stato finale d'una macchina), alla quale univocamente corrisponde una regola pozzo. Una mossa di riduzione può dunque scegliere a colpo sicuro la stringa da emettere.

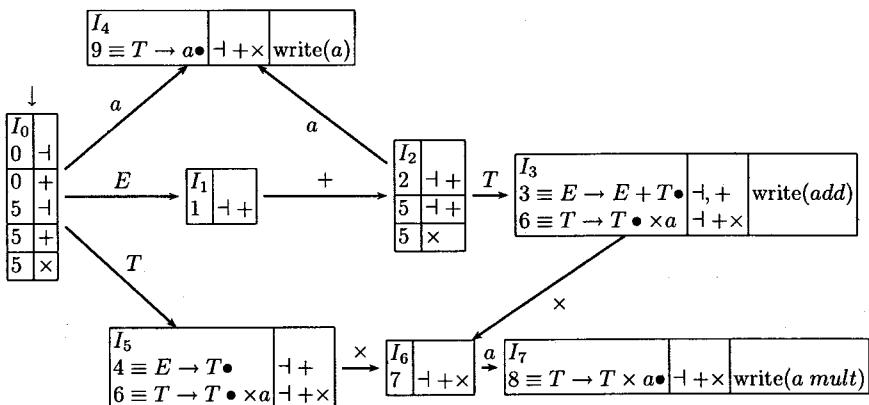
*Esempio 5.27.* Traduzione di espressioni in postfisso.

La grammatica dell'es. 4.57 di p. 218 definiva le espressioni con due operatori infissi, che si vogliono ora tradurre in operatori postfissi. La grammatica di traduzione è sotto mostrata, anche sotto forma di rete di macchine. Essa è stata scritta con l'avvertenza di lasciare i caratteri pozzo in fondo alle regole.

$$E \rightarrow E \frac{+}{\varepsilon} T \frac{\varepsilon}{add} \mid T \quad T \rightarrow T \frac{\times a}{\varepsilon} \frac{\varepsilon}{a mult} \mid \frac{a \varepsilon}{\varepsilon a}$$



La macchina pilota  $LR(1)$  precedentemente costruita si trasforma facilmente in quella del traduttore, inserendo le azioni di stampa associate alle riduzioni, come sotto mostrato.



Il programma del parsificatore deve ora eseguire le azioni di stampa associate alle riduzioni.

Forma normale postfissa  $\rightarrow LR(K)$

Conviene definire un'opportuna forma normale delle grammatiche di traduzione che permette di calcolare la traduzione con azioni di scrittura nelle sole mosse di riduzione.

Definizione 5.28. Grammatica di traduzione postfissa.

Una grammatica di traduzione è postfissa se le regole della grammatica pozzo sono del tipo  $A \rightarrow \gamma w$ , dove  $\gamma \in V^*$  e  $w \in \Delta^*$ .

In parole, non vi possono essere stringhe pozzo all'interno di una regola, ma soltanto alla fine di essa. L'esempio precedente è postfisso, così come l'es. 5.13 (p. 274). Non è postfisso l'es. 5.22 (p. 283). Dal punto di vista delle relazioni di traduzione che si possono definire, le grammatiche di traduzione postfisse sono altrettanto potenti di quelle prive di tale restrizione, anche se talvolta meno comprensibili. Si mostra come trasformare una grammatica nella forma postfissa.

Algoritmo 5.29. Rendere postfissa una grammatica di traduzione.

Si prenda una regola  $A \rightarrow \alpha$  della grammatica di traduzione data  $G_t$ ; essa, se non è già nella forma postfissa, si può scrivere, evidenziando la più lunga stringa pozzo  $v \in \Delta^+$ , posta più a destra, come:

$$A \rightarrow \gamma \frac{\varepsilon}{v} \eta$$

dove  $\gamma$  è qualsiasi, mentre  $\eta$  è una stringa non vuota che non contiene caratteri pozzo. Si sostituisce questa regola con le seguenti:

$$A \rightarrow \gamma \frac{\xi}{v} n \quad A \rightarrow \gamma Y \eta \quad Y \rightarrow \frac{\epsilon}{v}$$

dove  $Y$  è un nuovo nonterminale. La seconda regola è postfissa. Se poi la prima regola viola ancora la condizione postfissa, si individua di nuovo la più lunga stringa pozzo posta più alla destra in  $\gamma$ , e si ripete la stessa trasformazione. Procedendo così per ogni elemento pozzo incontrato in una posizione che violi la condizione postfissa, si ottiene facilmente una grammatica di traduzione postfissa.

Basta un esempio per illustrare il procedimento e constatare che le due grammatiche definiscono la stessa traduzione.

*Esempio 5.30.* Trasformazione nella forma normale postfissa.

La traduzione d'una espressione infissa nella scrittura prefissa è specificata dalla grammatica  $G_t$ , la cui prima regola viola la condizione della forma normale postfissa, come si nota nella grammatica pozzo  $G_2$ .

La versione postfissa della grammatica di traduzione è  $G'_t$ , la cui grammatica pozzo  $G'_1$  soddisfa la condizione.

| $G_t$ originale   | $G_1$                    | $G_2$                           |
|---|--------------------------|---------------------------------|
| $E \rightarrow \frac{\epsilon}{add} E \frac{+a}{a}$       | $E \rightarrow E + a$    | $E \rightarrow add \frac{E}{E}$ |
| $E \rightarrow \frac{a}{a}$                               | $E \rightarrow a$        | $E \rightarrow a$               |
| $G'_t$ postfissa  | $G'_1$                   | $G'_2$                          |
| $E \rightarrow YE \frac{+a}{\epsilon} \frac{\epsilon}{a}$ | $E \rightarrow YE + a$   | $E \rightarrow YEa$             |
| $E \rightarrow \frac{a}{\epsilon} \frac{\epsilon}{a}$     | $E \rightarrow a$        | $E \rightarrow a$               |
| $Y \rightarrow \frac{\epsilon}{add}$                      | $Y \rightarrow \epsilon$ | $Y \rightarrow add$             |

Non è difficile vedere che le traduzioni definite dalle due grammatiche sono eguali.

Si noti che una regola vuota è stata aggiunta alla grammatica sorgente.

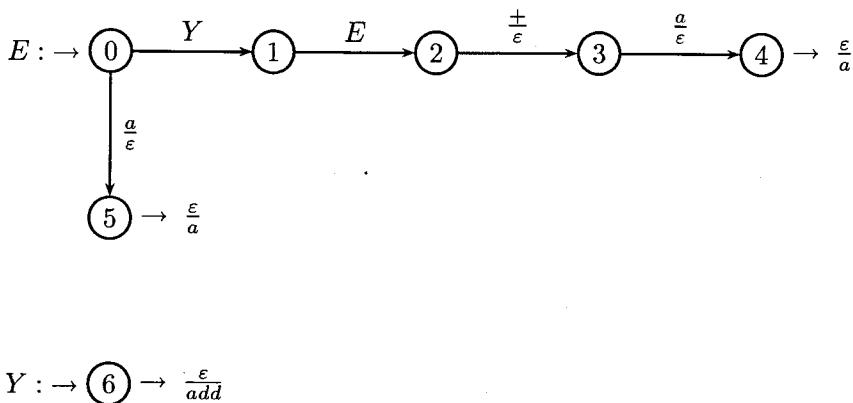
L'introduzione di nonterminali ausiliari come  $Y$  rende meno leggibile la forma postfissa. Inoltre, poiché la normalizzazione aggiunge delle regole vuote alla grammatica sorgente, essa può talvolta perdere la proprietà  $LR(1)$  o richiedere un allungamento della prospezione.

*Proprietà 5.31.* Il calcolo della traduzione definita da una grammatica di traduzione postfissa, tale che la grammatica sorgente sia del tipo  $LR(k)$ , può essere svolto in linea dal parsificatore, effettuando le azioni di scrittura nei soli macrostati di riduzione.

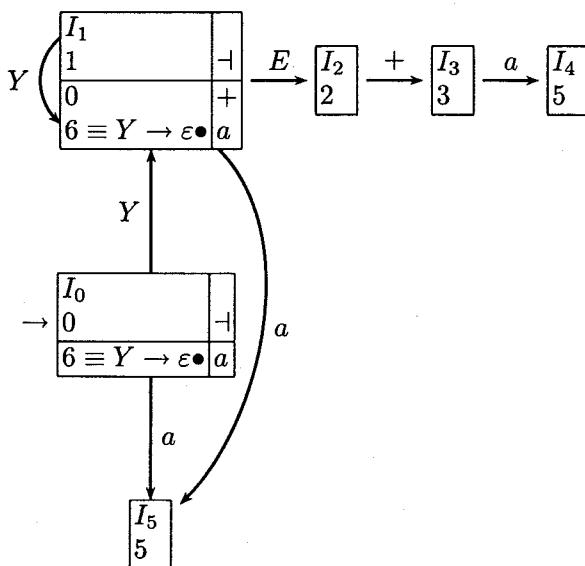
L'interesse della forma postfissa deriva dal fatto che essa rinvia l'emissione della traduzione ai momenti adatti all'analisi sintattica ascendente. Continuando l'es. precedente si sperimenteranno i limiti di questa tecnica di trasformazione della grammatica.

*Esempio 5.32.* Violazione condizione  $LR(1)$  con la forma normale.

La forma normale postfissa  $G'_t$  della grammatica di traduzione dell'es. 5.30 è disegnata come rete di macchine:



La grammatica sorgente originale  $G_1$  soddisfa la condizione  $LR(0)$ , ma l'aggiunta della regola  $Y \rightarrow \varepsilon$  nella grammatica postfissa crea un conflitto spostamento riduzione nei macrostati  $I_0$  e  $I_1$  della macchina pilota sotto disegnata:



In altri casi, per fortuna, la trasformazione della grammatica di traduzione nella forma postfissa non lede il funzionamento deterministico del parsificatore  $LR(1)$ .

### Albero sintattico come traduzione

Un impiego classico della traduzione ascendente o discendente è la costruzione dell'albero sintattico d'una stringa data. Nella programmazione un albero è solitamente rappresentato da una struttura-dati facente uso di puntatori. Per costruirlo non bastano le traduzioni puramente sintattiche, ma occorrono delle azioni semantiche che saranno trattate più avanti. Qui, invece dell'albero in forma di struttura a puntatori, basterà produrre la sequenza delle etichette delle regole sorgente, applicate per costruire l'albero.

Sia data una grammatica sorgente, con le regole numerate per riferimento. Per costruire lo schema di traduzione che stampa la sequenza delle etichette, si fa corrispondere a ciascuna regola sorgente la regola pozzo a fianco indicata:

| etichetta | regola di traduzione   | regola modificata                           |
|-----------|------------------------|---|
| $r_i$     | $A \rightarrow \alpha$ | $A \rightarrow \alpha \frac{\epsilon}{r_i}$ |

La traduzione così definita non è altro che la sequenza delle riduzioni destre. Poiché la grammatica è per ipotesi  $LR$  e lo schema è postfisso, l'analizzatore sintattico può facilmente calcolare la traduzione.

## Confronti

Si ricapitolano le considerazioni sulle tecniche di trasformazione dei parsificatori in traduttori. Il vantaggio della tecnica discendente è quello di permettere la costruzione del traduttore per qualsiasi schema sintattico di traduzione, anche non postfisso, naturalmente a condizione che la grammatica sorgente soddisfi la condizione  $LL(k)$ . Inoltre, la costruzione manuale dei traduttori  $LL(k)$  a discesa ricorsiva è agevole e produce dei programmi leggibili e modificabili.

D'altra parte, il limite imposto alla tecnica ascendente dal vincolo che la grammatica di traduzione sia postfissa è controbilanciato dalla migliore adeguatezza delle grammatiche  $LR(k)$  nella descrizione del linguaggio sorgente. Ricordando che la condizione  $LL(k)$  è più restrittiva di quella  $LR(k)$  e talvolta richiede fastidiosi aggiustamenti della grammatica, si può concludere che le tecniche ascendente e discendente hanno ciascuna pregi e difetti.

### 5.5.7 Proprietà di chiusura rispetto alle traduzioni

Si termina ora lo studio delle traduzioni puramente sintattiche, riassumendo in un quadro l'effetto che la traduzione può avere sulla famiglia cui un linguaggio appartiene. La domanda è la seguente: se si dà un linguaggio  $L$  d'una certa famiglia in ingresso a un traduttore d'un certo tipo, che calcola la funzione di traduzione  $\tau$ , il linguaggio immagine

$$\tau(L) = \{y \in \Delta^* \mid y = \tau(x) \wedge x \in L\}$$

a quale famiglia appartiene? Ad es. se si dà un linguaggio libero come ingresso a un traduttore (*IO-automa*) a pila, il linguaggio prodotto in uscita è ancora libero?

Non si confondano il linguaggio  $L$  e il linguaggio sorgente  $L_1$  del traduttore, anche se entrambi hanno lo stesso alfabeto  $\Sigma$ . Il linguaggio sorgente, come noto, contiene tutte e sole le stringhe riconosciute dall'automa d'ingresso soggiacente al traduttore. Applicando il traduttore a una frase di  $L$ , a seconda che essa appartenga o meno al linguaggio sorgente del traduttore, si ottiene una stringa pozzo o un errore.

Il quadro delle proprietà di appartenenza è il seguente:

| $L$         | <i>Traduttore finito</i>       | <i>Traduttore a pila</i>                    |
|-------------|--------------------------------|---|
| $L \in REG$ | <sup>1</sup> $\tau(L) \in REG$ | <sup>2</sup> $\tau(L) \in LIB$              |
| $L \in LIB$ | <sup>3</sup> $\tau(L) \in LIB$ | <sup>4</sup> $\tau(L)$ non sempre $\in LIB$ |

I casi 1 e 3 sono corollari del Teorema di Nivat (p. 267) e del fatto che sia famiglia *REG* che la famiglia *LIB* sono chiuse rispetto all'intersezione con linguaggi regolari (p. 158). In sostanza la macchina che riconosce il linguaggio  $L$  può essere combinata con il traduttore, ottenendo un nuovo traduttore che ha come linguaggio sorgente l'intersezione  $L \cap L_1$ . Il tipo del traduttore sarà

lo stesso del tipo del riconoscitore di  $L$ : a pila se  $L$  è libero, finito se  $L$  è regolare.

Tale nuovo traduttore può poi essere trasformato nel riconoscitore del linguaggio  $\tau(L)$ , eliminando i caratteri dell'alfabeto sorgente dalle sue mosse e conservando soltanto i caratteri dell'alfabeto pozzo.

Per il caso 2 vale ancora il ragionamento precedente. Un esempio del caso 2 è fornito dalla traduzione d'una stringa  $u \in L = \{a, b\}^*$  nel suo palindromo  $uu^R$ , che chiaramente è un linguaggio libero.

Il caso 4 si differenzia dagli altri perché l'intersezione dei due linguaggi liberi  $L$  e  $L_1$  non sempre è libera. Di conseguenza non è detto che un automa a pila possa riconoscere il linguaggio immagine di  $L$  nella traduzione.

*Esempio 5.33.* Trasduzione a pila d'un linguaggio libero.

Un esempio del caso 4 è la traduzione del linguaggio libero

$$L = \{a^n b^n c^* \mid n \geq 0\}$$

nel linguaggio a tre esponenti (es. 2.81 p. 76)

$$\tau(L) = \{a^n b^n c^n \mid n \geq 0\}$$

che si sa non essere libero.

Tale immagine è definita dalla grammatica di traduzione seguente

$$S \rightarrow \left(\frac{a}{a}\right)^* X \quad X \rightarrow \frac{b}{b} X \frac{c}{c} \mid \varepsilon$$

la quale impone lo stesso numero di  $b$  e di  $c$  in una stringa pozzo.

## 5.6 Traduzioni semantiche

I modelli di traduzione studiati sono assai limitati nelle funzioni che possono calcolare, a causa della semplicità dei dispositivi traduttori: gli *IO*-automi, finiti e a pila. La maggior parte dei problemi di compilazione richiede invece funzioni che superano le capacità di tali modelli. Un primo esempio è la traduzione d'un numero dalla base 2 alla base 10. Un altro esempio, tipico dei linguaggi programmativi, è la compilazione d'una dichiarazione di **record** come

```
LIBRO :
record
AUT: char(8); TIT: char(20); PREZZO: real; QUANT: int;
end
```

in una tabella in cui ogni simbolo possiede più attributi: il tipo, le dimensioni in byte, l'indirizzo di ogni campo (partendo da un indirizzo di base fisso ad es. 3401):

| simbolo | tipo   | dimensione | indirizzo |
|---------|--------|------------|-----------|
| LIBRO   | record | 34         | 3401      |
| AUT     | string | 8          | 3401      |
| TIT     | string | 20         | 3409      |
| PREZZO  | real   | 4          | 3429      |
| QUANT   | int    | 2          | 3433      |

In entrambi gli esempi il risultato della traduzione richiede delle funzioni aritmetiche, che non possono essere calcolate dagli automi considerati.

Per superare la difficoltà, si potrebbe immaginare di passare a modelli più potenti di automi traduttori, come le macchine di Turing, o a schemi di traduzione basati sulle grammatiche del tipo contestuale. Ma si è già argomentato (cap. 2, p. 87) che tali grammatiche sono difficili da progettare e da comprendere, già nel loro impiego per definire la sintassi. A maggior ragione, il loro utilizzo per programmare le funzioni di traduzione porterebbe a formulazioni intricate e praticamente inutilizzabili.

Scartato quindi l'impiego di automi traduttori astratti più potenti, la soluzione adottata è di scrivere le funzioni di traduzione in un linguaggio di programmazione. Per evitare confusione, lo si chiamerà il *linguaggio del compilatore* o anche il *metalinguaggio semantico*.

Al fine di dare ordine e chiarezza al progetto, il programma del compilatore sarà suddiviso in parti corrispondenti alla struttura sintattica del linguaggio sorgente. Pertanto i traduttori progettati con questo approccio sono detti *quidati dalla sintassi*. Essi sono molto più espressivi e generali di quelli puramente sintattici finora studiati, i quali hanno il pregio e il limite di essere completamente formalizzati.

Il salto dai metodi sintattici a quelli semanticici sta appunto nella introduzione

di procedure, che operano sull'albero sintattico della frase sorgente e calcolano certe variabili, dette *attributi semantici*, i cui valori costituiscono la traduzione o, come si dice, il *significato o semantica*, della frase.

I traduttori guidati dalla sintassi non sono un modello formale, poiché le procedure di calcolo degli attributi sono programmi non formalizzati; essi sono meglio classificati come un metodo di ingegneria del software per progettare ordinatamente i compilatori.

Per completare il quadro, è da dire che esistono altri metodi semantici di natura formale, capaci di definire in modo completo e rigoroso il significato dei linguaggi di programmazione, usando approcci appartenenti alla logica; ma il loro studio esula dagli obiettivi di questo libro.<sup>13</sup>

Nella compilazione guidata dalla sintassi, l'elaborazione è concettualmente divisa in due fasi, poste in cascata:

1. Parsificazione o analisi sintattica.
2. Valutazione o analisi semantica

La prima fase è ben nota, e produce un albero sintattico, spesso in un formato astratto, che conserva soltanto le parti che contengono informazioni utili per la traduzione. Di solito gli elementi superficiali (come i delimitatori) del linguaggio sorgente sono cancellati.

La valutazione semantica è guidata dunque dalla sintassi astratta del linguaggio. Questa fase consiste nell'applicazione delle funzioni semantiche, nodo per nodo, in tutto l'albero sintattico, fino a completare il calcolo di tutti gli attributi che contribuiscono alla traduzione.

Il disaccoppiamento tra la parsificazione e la valutazione semantica consente maggiore libertà nella scrittura delle due sintassi, concreta e astratta. La prima deve conformarsi al manuale di riferimento del linguaggio, non può essere ambigua e deve essere adatta all'algoritmo di analisi prescelto. Invece la sintassi astratta sarà la più semplice possibile, compatibilmente con la struttura semantica del linguaggio. La presenza d'ambiguità nella sintassi astratta non provoca la perdita d'univocità nella traduzione, poiché si fa l'ipotesi che il parsificatore passi al valutatore semantico un solo albero astratto per frase.

La compilazione a due passate sopra descritta è la più comune, ma nei traduttori più semplici le due fasi possono essere riunite. Evidentemente in tale caso si userà una sola sintassi, quella concreta del linguaggio sorgente.

### 5.6.1 Grammatiche con attributi

Occorre precisare che cosa s'intenda per significato d'una frase d'un linguaggio libero. Il significato è l'insieme dei valori che sono assegnati da certe funzioni,

<sup>13</sup>I metodi semantici formali sono necessari se si vuole dimostrare la correttezza del processo di traduzione, ossia la proprietà che, per ogni programma sorgente, la corrispondente stringa pozzo esprime esattamente lo stesso significato. Si vedano ad es. [16, 53].

dette semantiche, agli attributi dei simboli nonterminali, nell'albero sintattico della frase. Le funzioni sono associate a ogni regola della grammatica. L'insieme delle regole e delle funzioni costituisce la *grammatica con attributi*.

Per evitare confusioni la grammatica libera sarà chiamata sintassi, riservando il termine grammatica a quella a attributi. Le regole sintattiche saranno chiamate produzioni.

### Esempio introduttivo

Il metodo delle grammatiche con attributi è introdotto con l'esempio del calcolo del valore in base dieci d'un numero binario frazionario.

*Esempio 5.34.* (Knuth<sup>14</sup>).

Il linguaggio sorgente è definito dall'espressione regolare

$$L = \{0, 1\}^+ \bullet \{0, 1\}^+$$

da interpretare come un numero binario, con il punto che separa la parte intera da quella frazionaria. Ad es. il significato della stringa  $1101 \bullet 01$  è il numero 13,25 in base dieci. Si definisce il linguaggio con la sintassi mostrata in prima colonna. L'assioma è  $N$ ,  $D$  sta per una stringa binaria (la parte intera o frazionaria),  $B$  per un bit.

Grammatica con attributi:

| sintassi                    | funzioni semantiche                |
|-----------------------------|------------------------------------|
| $N \rightarrow D \bullet D$ | $v_0 := v_1 + v_2 \times 2^{-l_2}$ |
| $D \rightarrow DB$          | $v_0 := 2 \times v_1 + v_2$        |
| $D \rightarrow B$           | $l_0 := l_1 + 1$                   |
| $B \rightarrow 0$           | $v_0 := 0$                         |
| $B \rightarrow 1$           | $v_0 := 1$                         |

A destra vi sono le definizioni delle funzioni o regole semantiche che calcolano gli attributi:

| attributo       | dominio         | nonterminali aventi l'attributo |
|-----------------|-----------------|---------------------------------|
| $v$ , valore    | numero decimale | $N, D, B$                       |
| $l$ , lunghezza | intero          | $D$                             |

Una funzione semantica è sempre associata a una produzione sintattica che le fa da *supporto*. Il pedice di un attributo, come  $v_0, v_1, v_2, l_2$  alla prima riga della grammatica, specifica a quale simbolo della produzione sia associato l'attributo, usando la seguente convenzione:<sup>15</sup>

<sup>14</sup>Con questo esempio D. Knuth introduce in [29] il metodo delle grammatiche a attributi come sistematizzazione delle tecniche di progetto dei compilatori.

<sup>15</sup>Altre convenzioni possono essere adottate, ad es. , nella prima funzione, si può scrivere  $v$  of  $N$  al posto di  $v_0$ .

$$\underbrace{N}_0 \rightarrow \underbrace{D}_1 \bullet \underbrace{D}_2$$

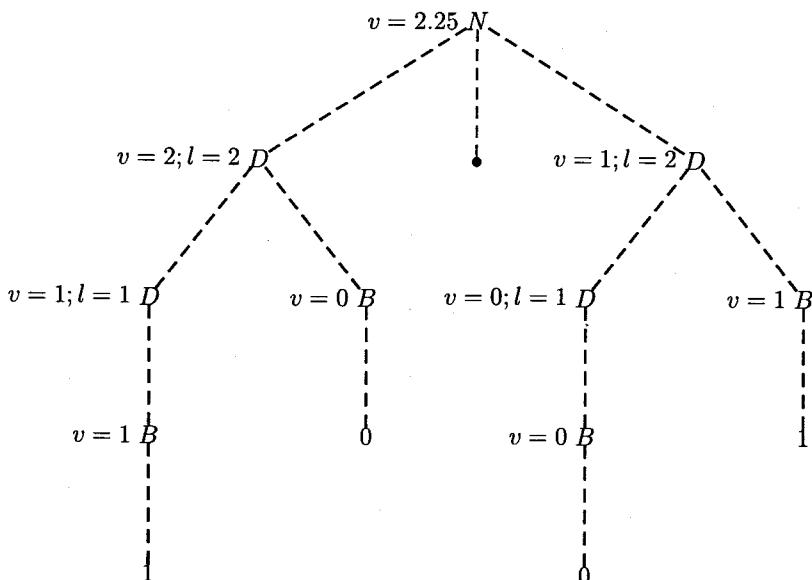
ossia  $v_0$  è associato alla parte sinistra  $N$ ,  $v_1$  al primo nonterminale della parte destra, ecc.. Se in una produzione il simbolo nonterminale non è ripetuto, come avviene con  $N$  nella prima produzione, si può, senza rischio di confusione, scrivere  $v_N$  invece di  $v_0$ .

Tale regola semantica assegna all'attributo  $v_0$  il valore calcolato dall'espressione avente gli attributi  $v_1, v_2, l_2$  come argomenti. In forma funzionale si può scrivere

$$v_0 := f(v_1, v_2, l_2)$$

Data una stringa sorgente, per calcolare la traduzione, si costruisce l'albero sintattico, poi in ogni nodo, si applica una funzione, cominciando dai nodi in cui gli argomenti della funzione sono noti. Il calcolo termina, quando tutti gli attributi sono stati calcolati.

L'albero così *decorato* con i valori degli attributi, costituisce la traduzione della stringa data,  $10 \bullet 01$ :



Più ordini di calcolo sono possibili: tutti quelli che rispettano la condizione che una funzione non può essere eseguita prima delle funzioni che calcolano i suoi argomenti.

L'attributo contenente il risultato finale della traduzione è in questo esempio il valore presente nella radice dell'albero, mentre gli attributi degli altri nodi

sono risultati intermedi. Tale attributo costituisce il significato della frase sorgente.

### 5.6.2 Attributi sinistri e destri

Nella grammatica dell'es. 5.34 precedente, il flusso di calcolo degli attributi ha un orientamento ben preciso dal basso verso l'alto, perché un attributo della parte sinistra (padre) d'una produzione è definito da una funzione che ha come argomenti i soli attributi della parte destra (figli). Gli attributi definiti da funzioni siffatte sono detti sinistri o sintetizzati.

In generale, rispetto alla produzione di supporto, le posizioni del risultato e degli argomenti d'una funzione semantica possono essere diverse.

Da un lato il risultato della funzione può essere l'attributo di un simbolo della parte destra della produzione di supporto. Tale attributo è allora detto destro o ereditato.

D'altro lato, gli argomenti delle funzioni semantiche, oltre che attributi sinistri, possono essere attributi destri.

Per concretizzare il discorso si presenta una grammatica in cui le posizioni degli argomenti e dei risultati delle funzioni sono sia sinistri che destri.

*Esempio 5.35.* Divisione di testo in righe (da Reps).

Si deve suddividere un testo in righe. La sintassi genera delle frasi fatte d'una o più parole separate da uno spazio (scritto  $\perp$ ). Si suppone che le frasi debbano apparire su un video di larghezza limitata a  $W$  caratteri, in modo tale che ogni riga contenga il numero massimo possibile di parole, senza che vi siano parole spezzate su due righe consecutive (nessuna parola è di lunghezza maggiore di  $W$ ). Le colonne sono numerate da 1 a  $W$ .

La grammatica calcola l'attributo *ultimo*, il numero della colonna in cui si trova l'ultima lettera di ogni parola. Così la frase "la torta ha gusto ma la grappa ha forza" è disposta su un video di larghezza  $W = 13$ :

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|----|----|----|----|
| l | a | t | o | r | t | a |   | h | a  |    |    |    |
| g | u | s | t | o | m | a |   | l | a  |    |    |    |
| g | r | a | p | p | a | h | a |   |    |    |    |    |
| f | o | r | z | a |   |   |   |   |    |    |    |    |

La variabile *ultimo* vale 2 per la, 8 per torta, 11 per ha, ..., e 5 per forza.

La sintassi genera liste di parole separate dallo spazio. Il terminale *c* sta per un carattere. Per calcolare la disposizione del testo, si usano i seguenti attributi:

lung<sub>h</sub>, la lunghezza d'una parola (sinistro);

prec, la colonna dell'ultimo carattere della parola precedente (destro);

ultimo, la colonna dell'ultimo carattere della parola (sinistro).

Per calcolare l'attributo *ultimo* d'una parola si deve prima conoscere la colonna dell'ultimo carattere della parola immediatamente precedente; vaibre

indicato da *prec*. Per la prima parola il valore di *prec* è -1.  
Le regole di calcolo sono espresse dalla grammatica seguente:

| sintassi                           | attributi destri                           | attributi sinistri   |
|------------------------------------|--|--|
| 1 $S_0 \rightarrow T_1$            | $prec_1 := -1;$                            |  |
| 2 $T_0 \rightarrow T_1 \sqcup T_2$ | $prec_1 := prec_0$<br>$prec_2 := ultimo_1$ | $ultimo_0 := ultimo_2$   |
| 3 $T_0 \rightarrow V_1$            |  | $ultimo_0 :=$<br>if ( $prec_0 + 1 + lungh_1 \leq W$ )<br>then ( $prec_0 + 1 + lungh_1$ )<br>else $lungh_1$ |
| 4 $V_0 \rightarrow cV_1$           |  | $lungh_0 := lungh_1 + 1$   |
| 5 $V_0 \rightarrow c$              |  | $lungh_0 := 1$   |

La sintassi merita due osservazioni. Primo, i pedici dei simboli nonterminali servono soltanto a chiarire il riferimento per gli argomenti delle funzioni, non differenziano le classi sintattiche, e possono essere omessi.

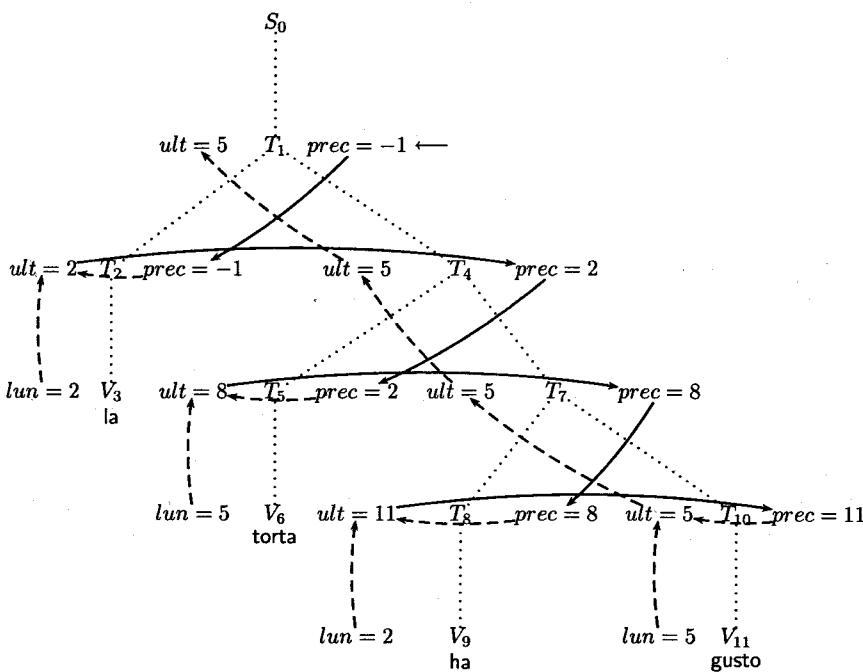
Secondo, la sintassi è ambigua, a causa della regola  $T \rightarrow T \sqcup T$  bilateramente ricorsiva. Ma gli inconvenienti che, in fase di analisi sintattica, derivano dall'ambiguità, qui non sono rilevanti, perché si suppone che al valutatore semantico arrivi dal parsificatore un solo albero sintattico. La versione ambigua della sintassi è stata preferita per la sua brevità.

La lunghezza d'una parola  $V$  è assegnata all'attributo sinistro *lungh* nelle due ultime produzioni. L'attributo *prec* è destro, poiché il valore è assegnato a un simbolo della parte destra, nelle due prime produzioni. L'attributo *ultimo* è sinistro e il suo valore, nei diversi nodi  $T$  d'un albero sintattico, costituisce il risultato finale.

Per scegliere l'ordine di valutazione degli attributi, occorre esaminare le dipendenze tra le istruzioni di assegnamento. Nella figura gli attributi sinistri sono disegnati a sinistra del nodo e quelli destri a destra; i nodi sintattici sono numerati per riferimento. Per evitare intrichi, non sono mostrati i sottoalberi di  $V$ , ma il suo attributo *lungh* è riportato con il suo valore.

Gli attributi dell'albero decorato sono i nodi d'un grafo delle dipendenze tra dati. Ad es. l'arco  $ult(2) \rightarrow prec(4)$  rappresenta la dipendenza del secondo attributo dal primo, portata dalla funzione  $prec_2 := ultimo_1$  della produzione 2. Se una funzione ha tot argomenti, altrettanti sono gli archi delle dipendenze. Si noti che gli archi connettono attributi della stessa produzione.

Ogni ordine di calcolo che soddisfi le precedenze permette di calcolare i valori degli attributi. Si ottiene così l'albero decorato.



È importante osservare che il risultato non dipende dall'ordine di applicazione delle funzioni. Tale proprietà vale per le grammatiche che rispettano certe condizioni che si enunceranno tra breve.

### *Opportunità degli attributi destri*

Questa grammatica usa attributi destri e sinistri, ma si potrebbe esprimere lo stesso calcolo con una grammatica priva di attributi destri? La risposta è affermativa, perché si può calcolare la posizione dell'ultima lettera di ogni parola nel seguente modo. Si calcola l'attributo sinistro *lungh*, poi si costruisce un nuovo attributo sinistro *lista*, avente come dominio una lista ordinata di interi, rappresentanti le lunghezze delle parole. Nella figura precedente il nodo *T* da cui deriva la frase *la torta ha gusto* ha l'attributo *lista* = < 2, 5, 2, 5 >. Il valore di *lista* nella radice dell'albero permette poi di calcolare, conoscendo la larghezza *W*, la posizione dell'ultimo carattere di ogni parola.

Ma questa soluzione ha un difetto fondamentale: il calcolo da fare nella radice sulla lista è sostanzialmente lo stesso del problema iniziale, dunque l'impostazione grammaticale non ha permesso di decomporre il problema in sottoproblemi più semplici.

Un altro inconveniente è che, in assenza di attributi destri come *ultimo*, l'informazione calcolata rimane concentrata nella radice e non può decorare i nodi interni dell'albero.

Infine l'abolizione degli attributi destri ha reso necessario l'uso di attributi non scalari, aventi un dominio complesso, rappresentabili da strutture dati come le liste o gli insiemi.

In definitiva, volendo definire con una grammatica una data traduzione, la soluzione più elegante e efficiente è spesso quella che fa uso di attributi sia destri sia sinistri.

### **5.6.3 Definizione di grammatica con attributi**

È giunto il momento di formalizzare i concetti introdotti negli esempi precedenti.

**Definizione 5.36.** Una grammatica con attributi  $H$  è costituita dalle seguenti entità:

1. Una sintassi libera  $G = (V, \Sigma, P, S)$ , dove  $V$  e  $\Sigma$  sono gli insiemi dei non-terminali e terminali,  $P$  sono le produzioni e  $S$  l'assioma. Spesso conviene imporre che l'assioma non figuri in alcuna parte destra di produzione.
2. Un insieme di simboli, gli attributi (semantici), associati ai simboli non-terminali e terminali.  
L'insieme degli attributi d'un simbolo  $D$  è denotato da  $\text{attr}(D)$ .  
L'insieme degli attributi della grammatica è spartito in due insiemi disgiunti detti attributi sinistri (o sintetizzati) e attributi destri (o ereditati).
3. Per ogni attributo  $\sigma$  è specificato un dominio, l'insieme dei valori che esso può assumere.
4. Un insieme di funzioni (o regole) semantiche. Ogni funzione è associata a una produzione

$$p : D_0 \rightarrow D_1 D_2 \dots D_r, r \geq 0$$

dove  $D_0$  è nonterminale e gli altri simboli sono terminali o non, detta supporto sintattico.

In generale più funzioni possono avere lo stesso supporto.

L'attributo  $\sigma$  associato al (non)terminale  $D_k$  si indica con  $\sigma_k$  o anche con  $\sigma_D$  se il nome del nonterminale è unico nella produzione  $p$  considerata.

Una funzione ha la forma:

$$\sigma_k := f(\text{attr}(\{D_0, D_1, \dots, D_r\}) \setminus \{\sigma_k\})$$

dove  $0 \leq k \leq r$ ; essa assegna un valore all'attributo  $\sigma$  del simbolo  $D_k$  mediante l'applicazione d'una regola di calcolo  $f$ , avente come argomenti certi attributi dei simboli della stessa produzione  $p$ , escluso il risultato della funzione.

Le funzioni semantiche sono funzioni totali nel loro dominio, scritte in

una notazione opportuna, detta metalinguaggio semantico, che può essere un linguaggio programmatico o una specifica di più alto livello, informale come uno pseudocodice, o formale come un linguaggio di specifica del software.

Una funzione  $\sigma_0 := f \dots$  definisce un attributo detto sinistro, del nonterminale  $D_0$  (ossia della parte sinistra o padre).

Una funzione  $\delta_k := f \dots$  con  $k \geq 1$  definisce un attributo detto destro, d'un simbolo della parte destra (o figlio).

È vietato che lo stesso attributo  $\sigma$  sia sinistro per una funzione e destro per un'altra, come detto in 2.

Occorre osservare che, poiché un terminale giace sempre nella parte destra d'una regola, ogni suo attributo è di tipo destro.<sup>16</sup>

Si consideri l'insieme  $\text{fun}(p)$  delle funzioni aventi  $p$  come supporto. Per esso devono valere le seguenti condizioni:

- per ogni attributo sinistro di  $D_0$ ,  $\sigma_0$ , esiste in  $\text{fun}(p)$  una, e una sola, funzione che lo definisce;
- per ogni attributo destro di  $D_0$ ,  $\delta_0$ , non esiste in  $\text{fun}(p)$  alcuna funzione che lo definisce;
- per ogni attributo sinistro  $\sigma_i$ , dove  $i \geq 1$ , non esiste in  $\text{fun}(p)$  alcuna regola che lo definisce;
- per ogni attributo  $\delta_i$ , dove  $i \geq 1$ , esiste in  $\text{fun}(p)$  una, e una sola, regola che lo definisce.

Gli attributi sinistri  $\sigma_0$  e destri  $\delta_i$ ,  $i \geq 1$ , essendo quelli definiti dalle funzioni aventi supporto  $p$ , sono detti interni per tale produzione.

Gli attributi destri  $\delta_0$  e sinistri  $\sigma_i$ ,  $i \geq 1$  sono detti esterni per la produzione  $p$ , in quanto sono definiti da funzioni aventi come supporto un'altra produzione.

5. Alcuni attributi possono essere inizializzati con valori costanti o con valori calcolati da funzioni esterne. Ciò è soprattutto frequente per gli attributi lessicali, cioè quelli dei simboli terminali. Il modo in cui questi valori sono calcolati esula dalla definizione della grammatica.

*Esempio 5.37.* Con riferimento all'esempio 5.35 (p. 299), si ha:

attributi sinistri: *lungh, ultimo*

attributi destri: *prec*

interni/esterni: per la produzione 2 sono interni *prec<sub>1</sub>, prec<sub>2</sub>, ultimo<sub>0</sub>*; sono esterni *prec<sub>0</sub>, ultimio<sub>0</sub>, ultimo<sub>2</sub>* (l'attributo *lungh* è estraneo alla produzione).

Un avvertimento riguarda il *principio di località* delle funzioni. È sbagliato porre come argomento o come risultato d'una funzione semantica, con

<sup>16</sup>Tuttavia, in pratica, gli attributi dei simboli terminali sono spesso definiti non per mezzo di funzioni, ma tramite valori costanti, che sono loro assegnati nella fase di analisi lessicale, a monte del processo di valutazione semantica.

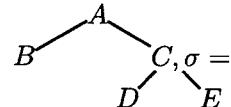
supporto  $p$ , un attributo estraneo alla produzione  $p$  stessa; come avverrebbe modificando le regole 2 nel modo seguente:

| sintassi                           |  |   |
|------------------------------------|--|---|
| 1 $S_0 \rightarrow T_1$            |  | ...   |
| 2 $T_0 \rightarrow T_1 \sqcup T_2$ |  | $prec_1 := prec_0 + \underbrace{lungh_0}_{\text{attr. non locale}}$ |
| 3 ...                              |  |   |

Infatti si viola la condizione di località, che preclude la visibilità degli attributi dei nodi che non siano il padre o i figli.

Conviene approfondire un aspetto della definizione. È fondamentale che ogni attributo dell'albero sintattico sia definito da uno e un solo assegnamento di valore, altrimenti il significato di qualche frase potrebbe non essere univoco. Per tale ragione, uno stesso attributo non può essere sinistro e destro, perché vi sarebbero due assegnamenti, come mostra lo schema:

| supporto             | funzione                      |  |
|----------------------|-------------------------------|--|
| 1 $A \rightarrow BC$ | $\sigma_C := f_1(attr(A, B))$ |  |
| 2 $C \rightarrow DE$ | $\sigma_C := f_2(attr(D, E))$ |  |



Infatti la variabile  $\sigma_C$ , interna in entrambe le produzioni, è destra nella prima e sinistra nella seconda. Il valore da essa assunto nell'albero mostrato dipende allora dall'ordine di applicazione delle funzioni, e la semantica così espressa perde una delle sue qualità più essenziali, il fatto di essere indipendente dalla realizzazione del valutatore degli attributi, ossia dall'ordine in cui esso applica le funzioni semantiche.

#### 5.6.4 Grafo delle dipendenze e valutazione degli attributi

Una grammatica serve a specificare la traduzione, senza l'onere di programmare l'ordine di calcolo della stessa. Infatti la procedura per il calcolo degli attributi d'un dato albero può essere costruita automaticamente, conoscendo le dipendenze funzionali tra gli attributi, come ora mostrato.

Si definisce il *grafo (orientato) delle dipendenze d'una funzione semantica*: i suoi nodi sono gli argomenti e il risultato, e vi è un arco da ogni argomento al risultato. I grafi delle dipendenze di tutte le funzioni, aventi la stessa produzione come supporto, formano il *grafo delle dipendenze della produzione*.

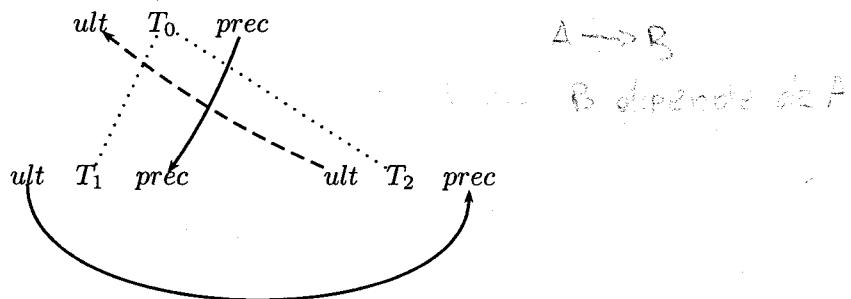
*Esempio 5.38.* Grafi delle dipendenze delle produzioni.

Conviene disegnare il grafo sovrapposto alla produzione di supporto, per evidenziare l'associazione tra gli attributi e i simboli della sintassi.

Per convenienza si risproduce la grammatica dell'es. 5.35:

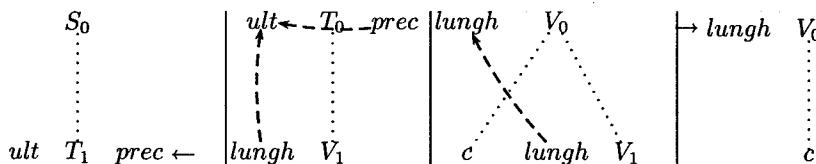
| sintassi                          | attributi destri                           | attributi sinistri   |
|-----------------------------------|--|--|
| 1 $S_0 \rightarrow T_1$           | $prec_1 := -1;$                            |  |
| 2 $T_0 \rightarrow T_1 \perp T_2$ | $prec_1 := prec_0$<br>$prec_2 := ultimo_1$ | $ultimo_0 := ultimo_2$   |
| 3 $T_0 \rightarrow V_1$           |  | $ultimo_0 :=$<br>if $(prec_0 + 1 + lungh_1) \leq W$<br>then $(prec_0 + 1 + lungh_1)$<br>else $lungh_1$ |
| 4 $V_0 \rightarrow cV_1$          |  | $lungh_0 := lungh_1 + 1$   |
| 5 $V_0 \rightarrow c$             |  | $lungh_0 := 1$   |

La figura mostra il grafo delle dipendenze delle funzioni, per la produzione 2:



Le tre funzioni di questa produzione hanno ciascuna un argomento, cioè un arco del grafo.

I grafi delle dipendenze delle altre produzioni sono:



In un grafo i nodi con (senza) archi entranti sono rispettivamente attributi interni (esterni).

Il grafo delle dipendenze d'un albero sintattico (decorato) è ottenuto incollando insieme i grafi delle singole produzioni usate nei nodi dell'albero. Come esempio si veda la figura a p. 301.

### Esistenza e unicità della soluzione

Ogni frase d'un linguaggio tecnico deve sempre avere un ben preciso significato, ossia un solo insieme di valori degli attributi dell'albero, altrimenti si avrebbe un caso sgradito di ambiguità semantica.

I valori sono calcolati da un insieme di assegnamenti, uno, e uno solo, per ogni istanza di attributo dell'albero. Tali assegnamenti compongono un sistema di

equazioni, le cui incognite sono i valori degli attributi. La soluzione delle equazioni è il significato della frase.

Se nel grafo delle dipendenze tra gli attributi dell'albero esiste un cammino (orientato)

$$\sigma_1 \rightarrow \sigma_2 \rightarrow \dots \rightarrow \sigma_{j-1} \rightarrow \sigma_j, \text{ con } j > 1$$

dove i  $\sigma_k$  sono attributi (anche diversi), allora le corrispondenti equazioni sono:

$$\begin{aligned}\sigma_j &= f_j(\dots, \sigma_{j-1}, \dots) \\ \sigma_{j-1} &= f_{j-1}(\dots, \sigma_{j-2}, \dots) \\ &\dots \\ \sigma_2 &= f_2(\dots, \sigma_1, \dots)\end{aligned}$$

Rileggendo gli esempi precedenti, si potrebbe verificare che, presa una frase con il suo albero, i cammini del grafo non formano mai dei circuiti.

Una grammatica è aciclica se, per ogni frase del linguaggio, il grafo delle dipendenze dell'albero<sup>17</sup> sintattico della frase è aciclico.

Al contrario, se nel grafo vi fosse un circuito, cioè se fosse  $\sigma_i = \sigma_k$  per due elementi  $1 \leq i < k \leq j$ , il sistema di equazioni potrebbe avere più d'una soluzione, e vi sarebbe ambiguità semantica.

Proprietà 5.39. Sia data una grammatica con attributi tale da soddisfare le condizioni della def. 5.36. Se il grafo delle dipendenze tra gli attributi d'un albero è aciclico, il sistema delle equazioni, corrispondenti alle funzioni semantiche, ha una e una sola soluzione.

Pertanto se una grammatica è aciclica, ogni albero sintattico ha uno e un solo insieme di valori per i suoi attributi.

Supponendo soddisfatta l'ipotesi di acicità, si mostra come ordinare linearmente le equazioni, in modo che ogni equazione sia calcolata dopo quelle che calcolano i suoi argomenti.

Algoritmo 5.40. Ordinamento topologico.

Sia  $G = (V, E)$  un grafo orientato aciclico, in cui i nodi sono identificati da numeri,  $V = \{1, 2, \dots, |V|\}$ . L'algoritmo calcola un ordine totale dei nodi, detto *topologico*. Il risultato,  $ord[i]$ , è un vettore che dà la posizione del nodo  $i$  nell'ordinamento.

*begin*

*m := 1; -- contatore*

*V<sub>0</sub> := {n ∈ V | il nodo n non ha archi}*

*while V<sub>0</sub> ≠ ∅*

*do*

*togli un nodo n da V<sub>0</sub>; ord[n] := m; m := m + 1*

<sup>17</sup> Al solito si suppone che il parsificatore fornisca un solo albero sintattico per frase.

```

end do
 $V := V \setminus V_0;$ 
while  $V \neq \emptyset$ 
do
   $V_0 := \{n \in V \mid \text{il nodo } n \text{ non ha archi entranti}\}$ 
  while  $V_0 \neq \emptyset$ 
  do
    togli un nodo  $n$  da  $V_0$ ;  $ord[n] := m$ ;  $m := m + 1$ 
  end do
   $V := V \setminus V_0;$ 
   $E := E \setminus \{\text{archi uscenti da nodi di } V_0\};$ 
end do
end

```

Il primo ciclo ordina arbitrariamente i nodi privi di dipendenze.  
In generale, l'ordine topologico non è unico.

*Esempio 5.41.* Applicando l'algoritmo al grafo di p. 301, un ordine topologico è:

$lungh_3, lungh_6, lungh_9, lungh_{11}, prec_1, prec_2, ult_2, prec_4,$   
 $prec_5, ult_5, prec_7, prec_8, ult_8, prec_{10}, ult_{10}, ult_7, ult_4, ult_1.$

Preso il primo nodo nell'ordinamento, l'equazione che lo definisce è necessariamente costante, ossia dà il valore iniziale all'attributo. Si procede poi applicando via via le equazioni in ordine topologico. Si ricorda che le funzioni sono totali, quindi producono sempre un risultato. In questo modo si completa la decorazione dell'albero.

Ma questa via è poco efficiente, perché richiede di applicare l'algoritmo di ordinamento agli attributi dell'albero, prima di procedere all'esecuzione degli assegnamenti. Per ottenere un valutatore più veloce, si vedrà tra breve come predeterminare un ordine fisso di visita, ossia una *schedulazione*, dei nodi, valido per ogni albero, in accordo con le dipendenze tra gli attributi.

Un secondo problema sospeso è quello dell'ipotesi di aciclicità della grammatica data: come verificare che nessun albero possa mai presentare circuiti nel grafo delle dipendenze.

Poiché il linguaggio sorgente è in genere infinito, il test di aciclicità non può essere certo fatto esaminando in modo esaustivo tutte le frasi. Un algoritmo per decidere se una grammatica con attributi è aciclica esiste ma è complesso<sup>18</sup>, e non necessario in pratica. Infatti sono più utili delle condizioni sufficienti, che, data la grammatica, permettono di verificare facilmente che non sia ciclica e di costruire allo stesso tempo la schedulazione usata dal valutatore degli attributi. Esse sono presentate nel seguito.

<sup>18</sup>L'algoritmo [29], [30] ha complessità asintotica  $NP$ -completa rispetto alle dimensioni della grammatica.

### 5.6.5 Valutazione semantica con una scansione

Un valutatore semantico molto efficiente dovrebbe visitare l'albero passando una sola volta su ogni nodo (o al peggio un piccolo numero di volte), calcolando gli attributi ivi pertinenti.

Un buon ordine di percorso è la visita in profondità d'un albero.

Detto  $N$  un nodo dell'albero, siano  $N_1, \dots, N_r$  i figli, e si indica con  $t_i$  il sottoalbero avente la radice in  $N_i$ .

La visita in profondità inizia dalla radice dell'intero albero. Poi la visita del sottoalbero  $t_{N_1}$ , avente la sua radice nel generico nodo  $N$ , procede così. L'algoritmo visita in profondità i sottoalberi  $t_1, \dots, t_r$ , in un ordine che non è necessariamente quello naturale  $1, 2, \dots, r$ , ma può essere una diversa permutazione.

Questa procedura di visita è ora applicata nell'algoritmo di valutazione semantica detto a una scansione<sup>19</sup>. Il calcolo degli attributi si svolge secondo il seguente schema:

- ① prima di visitare e valutare il sottoalbero  $t_N$ , si calcolano gli attributi destri del nodo  $N$ ;
- ② al termine della visita del sottoalbero  $t_N$  si calcolano gli attributi sinistri di  $N$ .

Non tutte le grammatiche consentono di valutare gli attributi con una scansione, perché certe dipendenze funzionali possono richiedere più visite dei nodi. Affinché una sola visita in profondità dell'albero permetta il calcolo degli attributi, si danno certe condizioni, che si possono verificare individualmente e facilmente sul grafo delle dipendenze  $dip_p$  di ciascuna produzione  $p$  della grammatica.

L'esperienza dice che nel progetto d'una grammatica con attributi è spesso abbastanza agevole rispettare tali condizioni, in modo da permettere la costruzione d'un valutatore semantico efficiente a una scansione.

#### Grammatica a una scansione

Per ogni produzione

$$p : D_0 \rightarrow D_1 D_2 \dots D_r, r \geq 0$$

è utile definire una relazione binaria tra i simboli  $D_i$ , rappresentata in un grafo, detto grafo dei fratelli frat<sub>p</sub>. I suoi nodi sono i simboli della parte destra della produzione  $\{D_1, D_2, \dots, D_r\}$ . Il grafo dei fratelli contiene l'arco

$$D_i \rightarrow D_j, i \neq j, i, j \geq 1$$

se esiste nel grafo delle dipendenze  $dip_p$  un arco  $\sigma_i \rightarrow \delta_j$ , tra un attributo del simbolo  $D_i$  e un attributo del simbolo  $D_j$ .

<sup>19</sup>One sweep.

QUALSiasi

Si noti che il grafo dei fratelli non ha gli stessi nodi del grafo delle dipendenze della produzione, perché i suoi nodi sono i simboli della sintassi, non gli attributi. Tutti gli attributi di  $dip_p$  aventi lo stesso pedice  $j$  si fondono nel nodo  $D_j$  di  $frat_p$ , quindi tra i due grafi vi è una relazione di omomorfismo.

#### **Definizione 5.42. Grammatica a una scansione.**

Una grammatica è detta a una scansione se, per ogni produzione  $p : D_0 \rightarrow D_1D_2 \dots D_r, r \geq 0$  valgono le condizioni:

1. nel grafo  $dip_p$  delle dipendenze, non esiste un circuito;
2. nel grafo  $dip_p$  delle dipendenze, non esiste un cammino

$$\sigma_i \rightarrow \dots \rightarrow \delta_i, i \geq 1$$

- da un attributo sinistro  $\sigma_i$  a un attributo destro  $\delta_i$  dello stesso simbolo  $D_i$  della parte destra della produzione;
3. nel grafo  $dip_p$  delle dipendenze, non esiste un arco  $\sigma_0 \rightarrow \delta_i, i \geq 1$ , da un attributo sinistro del padre  $D_0$  a un attributo destro d'un figlio  $D_i$ ;
4. il grafo dei fratelli  $frat_p$  è privo di circuiti.

Seguono alcune spiegazioni punto per punto.

1. Questa è condizione necessaria affinché la grammatica sia aciclica.
2. Se vi fosse un cammino  $\sigma_i \rightarrow \dots \rightarrow \delta_i, i \geq 1$ , l'attributo destro  $\delta_i$  non potrebbe essere calcolato prima di visitare il sottoalbero  $t_i$ , perché il valore dell'attributo sinistro  $\sigma_i$  sarebbe noto soltanto al termine della visita del sottoalbero. Ciò è in contrasto con la schedulazione di visita adottata.
3. Come al punto precedente, il valore dell'attributo  $\delta_i$  non sarebbe disponibile, quando inizia la visita del sottoalbero  $t_i$ .
4. Questa condizione permette di ordinare topologicamente i fratelli, ossia i sottoalberi, e così di organizzare la visita dei sottoalberi  $t_1, \dots, t_r$  in un ordine che vada bene per tutte le dipendenze di  $dip_p$ . Se il grafo dei fratelli fosse ciclico, vorrebbe dire che vi sono esigenze contrastanti sull'ordine di visita dei fratelli, e non esisterebbe una schedulazione valida per tutti gli attributi della parte destra di  $p$ .

#### **Algoritmo 5.43. Costruzione del valutatore a una scansione.**

Vi è una procedura per ogni nonterminale, i cui argomenti sono il sottoalbero e gli attributi destri della sua radice. La procedura visita il sottoalbero e al termine calcola gli attributi sinistri della radice. Per ogni produzione

$$p : D_0 \rightarrow D_1D_2 \dots D_r, r \geq 0$$

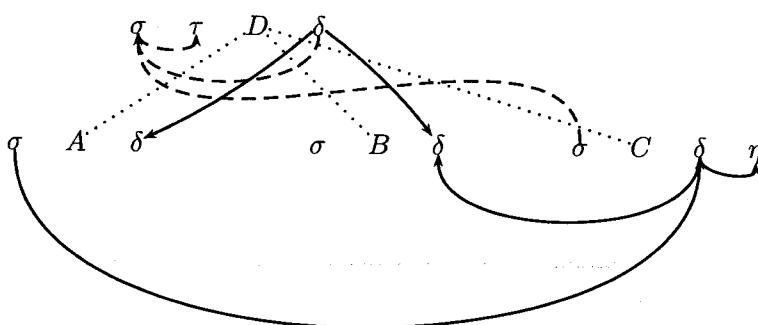
1. Costruisci un ordine topologico, detto  $OTF$ , dei nonterminali  $D_1, D_2 \dots D_r$  risp. al grafo dei fratelli  $frat_p$ .
2. Per ogni simbolo  $D_i, 1 \leq i \leq r$ , costruisci un ordine topologico, detto  $OTD$ , degli attributi destri del simbolo  $D_i$ .

3. Costruisci un ordine topologico, detto  $OTS$ , degli attributi sinistri del nonterminale  $D_0$ .

I tre ordinamenti  $OTF, OTD, OTS$  determinano la sequenza delle istruzioni della procedura semantica, mostrata nel prossimo esempio.

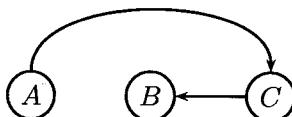
*Esempio 5.44.* Procedura semantica a una scansione.

La produzione  $D \rightarrow ABC$  ha il grafo delle dipendenze  $dip$



Non è difficile verificare che il grafo delle dipendenze soddisfa le condizioni 1., 2. e 3. della definizione 5.42:

1. non ci sono circuiti;
2. né cammini da un attributo sinistro come  $\sigma_B$  a un attributo destro, come  $\delta_B$ , dello stesso nodo;
3. né archi dagli attributi sinistri  $\sigma_D, \tau_D$  a qualche attributo destro di  $A, B, C$ ;
4. il grafo dei fratelli *frat* è aciclico:



Gli archi del grafo sono così ricavati:

$$\begin{aligned} A \rightarrow C &\text{ da } \sigma_A \rightarrow \delta_C, \\ C \rightarrow B &\text{ da } \delta_C \rightarrow \delta_B. \end{aligned}$$

I tre ordinamenti topologici possono essere così scelti:

grafo dei fratelli:  $OTF = A, C, B$ ;

attributi destri di ogni figlio: per  $A$  e  $B$  vi è un solo attributo; per  $C$  si ha  $OTD = \delta, \eta$ ;

attributi sinistri di  $D$ :  $OTS = \sigma, \tau$ .

$\stackrel{1}{S} \stackrel{2}{\sim} \stackrel{3}{S}$

Segue la procedura semantica di questa produzione, con le istruzioni scritte in un ordine allineato con gli ordinamenti topologici sopraindicati.

```

procedure D(in t, δD; out σD, τD)
begin
  δA := f1(δD)           1
  {   - le funzioni astratte sono denotate f1, f2, ecc.;

    A(tA, δA; σA)      2
    - chiamata di A per decorare il sottoalbero tA;
    δC := f2(σA)          3
    ηC := f3(δC)          4
    C(tC, δC, ηC; σC)  5
    - chiamata di C per decorare il sottoalbero tC;
    δB := f4(δD, δC)    6
    B(tB, δB; σB)      7
    - chiamata di B per decorare il sottoalbero tCB;
    σD := f5(δD, σB, σC) 8
    τD := f6(σD)          9
end

```

In conclusione, con questo metodo è semplice costruire un efficiente valutatore semantico ricorsivo, se la grammatica soddisfa la condizione per essere a una scansione.

### 5.6.6 Altri metodi di valutazione

I valutatori a una scansione sono semplici e efficienti, ma non vanno bene per tutte le grammatiche. Altre condizioni più generali sono state inventate e applicate al progetto dei valutatori<sup>20</sup>.

Un metodo piuttosto intuitivo consiste nel decomporre la valutazione in più stadi, ciascuno a una scansione, operanti in cascata sullo stesso albero sintattico.

Focalizzando il caso di due stadi, l'insieme degli attributi  $Attr$  della grammatica deve essere spartito dal progettista in due insiemi disgiunti  $Attr_1 \cup Attr_2 = Attr$ , che saranno valutati rispettivamente nel primo e nel secondo stadio. A ogni insieme sono associate le corrispettive funzioni semantiche, che costituiscono una sottogrammatica con attributi.

Occorre verificare che la prima sottogrammatica soddisfi le condizioni della definizione 5.36 (p. 302) e quelle per la valutazione a una scansione 5.42 (p. 309). Evidentemente gli attributi di  $Attr_1$  non devono dipendere dai rimanenti

<sup>20</sup>Diverse classi di valutatori sono stati studiati, tra cui quelli a più scansioni, a più visite, quelli per grammatiche ordinate OAG, e per grammatiche assolutamente acicliche. Una rassegna dei principali metodi è in [18] o anche in [13].

$Attr_2$ , affinché il primo stadio possa valutarli.

Si può allora costruire il primo stadio come un valutatore a una scansione che produce un albero decorato con i valori del primo insieme di attributi, mentre i secondi attributi restano da calcolare.

Per il secondo stadio, occorre verificare che anche la seconda sottogrammatica soddisfi le condizioni della definizione e la condizione per la valutabilità a una scansione.

Si noti che, per il secondo valutatore, gli attributi  $Attr_1$  sono delle costanti note, e quindi le dipendenze intercorrenti tra due elementi di  $Attr_1$  oltre che tra un elemento di  $Attr_1$  e un elemento di  $Attr_2$  non vanno considerate ai fini della verifica della condizione 5.42; ossia soltanto le dipendenze tra gli attributi del secondo insieme devono soddisfare la condizione.

Il secondo stadio, partendo dall'albero decorato con i primi attributi, in una scansione calcola gli attributi  $Attr_2$ .

Il punto cruciale per applicare questo metodo è di trovare una buona partizione degli attributi nei due (o più) sottoinsiemi. Poi la costruzione procede con lo stesso metodo della valutazione a una scansione.

Poiché la progettazione dell'analizzatore semantico di un linguaggio di grandi dimensione è un compito complesso, la partizione della valutazione in più stadi va incontro all'esigenza di modularizzazione del progetto. In pratica molti compilatori dividono l'analisi semantica in più stadi o fasi, per ridurne la complessità. Per esempio il primo stadio analizza le dichiarazioni delle varie entità (variabili, tipi, classi, ecc.) e il secondo stadio analizza le istruzioni eseguibili del linguaggio.

### 5.6.7 Analisi sintattica e semantica integrate

Una tecnica molto efficiente si offre quando la valutazione degli attributi può essere svolta direttamente da un parsificatore deterministico, evitando il passo separato di costruzione dell'albero sintattico astratto.

Tre sono le situazioni da considerare, a seconda della natura del linguaggio sorgente:

- linguaggio sorgente regolare: analisi lessicale con attributi lessicali;
- sintassi  $LL(k)$ : parsificatore a discesa ricorsiva con attributi;
- sintassi  $LR(k)$ : parsificatore a spostamento e riduzione con attributi.

Si esamineranno ora le condizioni che permettono queste modalità più dirette di calcolo degli attributi.

#### Analisi lessicale con valutazione di attributi

L'analizzatore lessicale (o scansore) ha lo scopo di segmentare il testo sorgente negli elementi lessicali o *lessemi*, ad es. identificatori, costanti intere o reali, commenti, ecc. Essi sono le più piccole sottostringhe cui può essere associata qualche proprietà semantica. Ad es. nel linguaggio Pascal la parola chiave

*begin* ha la proprietà di aprire una frase composta, mentre una sua sottostringa come *egin* non ha alcun significato.

Ogni linguaggio tecnico possiede un insieme finito di *classi lessicali*, come quelle citate. Ogni classe lessicale è un linguaggio formale regolare, come il caso noto degli identificatori (es. 2.27) definiti dalla espressione regolare di p. 28. Un lessema della classe identificatore è allora una stringa appartenente a tale linguaggio.

Nella definizione d'un linguaggio, sopra al livello lessicale della descrizione sta quello sintattico; la sintassi prende per disponibili i lessemi, trattandoli come simboli atomici del proprio alfabeto terminale. Alcuni dei lessemi hanno però anche un attributo semantico, il cui valore è calcolato dall'analizzatore lessicale.

### Classi lessicali

Esaminando più da vicino le classi lessicali, alcune sono linguaggi di cardinalità finita. Ad es. le parole chiave riservate d'un linguaggio programmatico formano una classe finita contenente in particolare

*{begin, end, if, then, else, while, do, ...}*

Similmente gli operatori aritmetici, logici e relazionali formano una classe lessicale finita.

In contrasto, gli identificatori, le costanti intere, i commenti sono esempi di classi lessicali infinite.

L'analizzatore lessicale è in sostanza un traduttore finito che vede la stringa sorgente come una sequenza di lessemi. Tra un lessema e l'altro ci possono essere, a seconda delle loro classi, spazi o altri separatori (come new-line). Il traduttore calcola lessema per lessema la traduzione, e elimina i separatori.

Nella stringa pozzo ogni lessema è di solito trascritto come una coppia di elementi: il nome (o un identificativo) della classe lessicale di appartenenza, e un attributo semantico detto *lessicale*.

Gli attributi lessicali cambiano da classe a classe e mancano per certe classi.

Alcuni casi tipici sono:

- costante decimale: l'attributo è il valore della costante in base 10;
- identificatore: l'attributo è una chiave che permetterà al compilatore di ricercare rapidamente l'identificatore in una tabella;
- commento: un commento può non avere alcun attributo, se l'ambiente di compilazione non mantiene la documentazione originale presente nel programma sorgente; altre volte invece i commenti sono mantenuti, e il loro attributo lessicale specifica dove ritrovarli;
- parola chiave: è priva di attributo lessicale; ha soltanto un codice che la identifica.

### Segmentazione univoca

In un linguaggio tecnico ben progettato le definizioni del lessico devono garantire che la segmentazione in lessemi sia unica, per ogni testo sorgente. Una cautela è necessaria nel caso in cui il concatenamento di due o più classi lessicali causi ambiguità. Ad es. la stringa *beta237* è segmentabile in più modi: come concatenamento dei lessemi *beta* di classe *identifier* e *237* di classe *integer*; o come concatenamento di *beta2* e di *37*, e ancora in altri modi.

In pratica l'ambiguità è spesso eliminata imponendo all'analizzatore la *regola del massimo prefisso riconosciuto*: essa comanda di segmentare una stringa  $x = uv$  in due lessemi  $u \in \text{identifier}$  e  $v \in \text{integer}$  in modo tale che  $u$  sia il più lungo prefisso di  $x$  appartenente alla classe *identifier*.

Nell'esempio, la regola assegna l'intera stringa *beta237* alla classe *identifier*. In tal modo la traduzione risulta univoca e può essere calcolata da un traduttore finito deterministico, arricchito con certe azioni per il calcolo degli attributi semantici.

### Attributi lessicali

Si è detto che l'attributo semantico d'un lessema è correlato alla sua classe lessicale, e quindi la funzione semantica che lo calcola è diversa da classe a classe.

Tuttavia è conveniente unificare fin dove possibile il trattamento lessicale degli attributi, associando a un lessema di qualsiasi classe uno stesso attributo *ss* di tipo stringa: la sottostringa sorgente che è stata riconosciuta come lessema. Ad es. l'attributo *ss* dell'identificatore *beta237* non è altro che la stringa '*beta237*' (o un puntatore ad essa).

Il traduttore finito, quando riconosce il lessema *beta237*, produce la traduzione

$$(\text{classe} = \text{identifier}, \text{ss} = 'beta237')$$

La coppia prodotta contiene tutte le informazioni necessarie per applicare in un secondo tempo la funzione semantica, specifica della classe identificatore. Tale funzione cerca la stringa *ss* nella tabella dei simboli del programma da compilare, se è assente la inserisce, e restituisce come attributo la posizione nella tabella. In accordo con la struttura scelta per il compilatore, tale funzione semantica farà parte della grammatica con attributi che descrive la traduzione guidata dalla sintassi.

### Parsificatore a discesa ricorsiva con attributi

Se la sintassi è adatta all'analisi discendente deterministica, e la grammatica con attributi è a una scansione, il calcolo degli attributi può procedere di pari passo con la parsificazione, a condizione che le dipendenze funzionali tra gli attributi soddisfino alcune restrizioni supplementari ora mostrate.

Si ricorda che l'algoritmo a una scansione (p. 309) visitava in profondità l'albero sintattico, percorrendo i sottoalberi  $t_1, \dots, t_r$  associati alla produzione  $D_0 \rightarrow D_1 \dots D_r$  in un ordine, anche diverso da quello naturale. L'ordine era scelto, mediante l'ordinamento topologico, in modo da rispettare le dipendenze funzionali tra gli attributi dei nodi  $1, \dots, r$ .

Poiché il parsificatore costruisce l'albero nell'ordine naturale, il sottoalbero  $t_j$  sarà costruito dopo i sottoalberi  $t_1, \dots, t_{j-1}$ . È dunque necessario vietare quelle dipendenze funzionali che imporrebbero una visita dei sottoalberi in un ordine che fosse una permutazione diversa da  $1, \dots, r$ .

#### Definizione 5.45. Condizione $L$ .

Una grammatica soddisfa la condizione  $L^{21}$  se, per ogni produzione  $p : D_0 \rightarrow D_1 \dots D_r$ :

1. la condizione 5.42 (p. 309) per essere a una scansione è soddisfatta;
2. nel grafo dei fratelli  $\text{frat}_p$  non vi sono archi  $D_j \rightarrow D_i, j > i \geq 1$ .

Si osservi che la seconda condizione vieta che un attributo destro del nodo  $D_i$  dipenda da un attributo (destro o sinistro) di un nodo  $D_j$  posto alla sua destra. In sostanza tale condizione fa sì che l'ordine naturale  $1, \dots, r$  soddisfi i vincoli di precedenza che condizionano la visita dei sottoalberi.

Proprietà 5.46. Se una grammatica con attributi è tale che

- la sintassi soddisfa la condizione  $LL(k)$ , e
- le regole semantiche soddisfano la condizione  $L$

allora è possibile costruire un parsificatore deterministico, detto analizzatore sintattico-semantico, che calcola gli attributi durante l'analisi sintattica.

La costruzione combina facilmente i concetti del parsificatore a discesa ricorsiva e del valutatore ricorsivo a una scansione, come mostra il seguente esempio.

Esempio 5.47. Analizzatore sintattico-semantico a discesa ricorsiva.

Si riformula l'es. 5.34 (p. 297) in una nuova grammatica che converte un numero frazionario minore di 1 dalla base due alla base 10. Il linguaggio sorgente è definito dall'espressione regolare

$$L = \bullet(0 \mid 1)^*$$

Il significato della stringa  $\bullet01$  è il numero 0,25 in base dieci.

Grammatica con attributi:

| sintassi                  | attributi sinistri | attributi destri                  |
|---------------------------|--------------------|-----------------------------------|
| $N \rightarrow \bullet D$ | $v_0 := v_1$       | $l_1 := 1$                        |
| $D \rightarrow BD$        | $v_0 := v_1 + v_2$ | $l_1 := l_0 \quad l_2 := l_0 + 1$ |
| $D \rightarrow B$         | $v_0 := v_1$       | $l_1 := l_0$                      |
| $B \rightarrow 0$         | $v_0 := 0$         |                                   |
| $B \rightarrow 1$         | $v_0 := 2^{-l_0}$  |                                   |

<sup>21</sup>La lettera  $L$  sta per left-to-right.

Gli attributi sono:

| <i>attributo</i> | <i>tipo</i> | <i>nonterminali associati</i> |
|------------------|-------------|-------------------------------|
| $v$ , valore     | sinistro    | $N, D, B$                     |
| $l$ , lunghezza  | destro      | $D, B$                        |

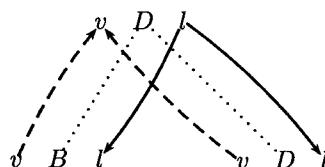
Si noti che il valore d'un bit è pesato con un esponente negativo pari alla sua distanza dal punto frazionale.

La sintassi risulta deterministica  $LL(2)$ . La verifica della condizione  $L$  segue, produzione per produzione.

$N \rightarrow \bullet D$ : Il grafo *dip* delle dipendenze ha soltanto l'arco  $v_1 \rightarrow v_0$ , pertanto:

- nel grafo non esiste un circuito;
- nel grafo non esiste un cammino dall'attributo sinistro  $v$  all'attributo destro  $l$  dello stesso figlio;
- nel grafo non esiste un arco dall'attributo  $v$  del padre a un attributo destro  $l$  d'un figlio;
- il grafo dei fratelli *frat* è privo di archi.

$D \rightarrow BD$ : Il corrispondente grafo delle dipendenze



- non ha circuiti;
- non presenta un cammino dall'attributo sinistro  $v$  all'attributo destro  $l$  di uno stesso simbolo (un figlio nella produzione);
- non ha un arco dall'attributo sinistro  $v$  del padre a un attributo destro  $v$  d'un figlio;
- il grafo dei fratelli è privo di archi.

$D \rightarrow B$ : idem come sopra.

$B \rightarrow 0$ : il grafo delle dipendenze è privo di archi.

$B \rightarrow 1$ : il grafo delle dipendenze ha l'unico arco  $l_0 \rightarrow v_0$  che è compatibile con una scansione. Non ci sono fratelli.

Il valutatore si compone, come il parsificatore, di tre procedure  $N, D, B$ , che hanno come argomenti gli attributi destri del padre e come risultati gli attributi sinistri del padre. Per sfruttare la prospezione, le procedure mantengono in due variabili il carattere corrente  $cc1$  e quello successivo  $cc2$ . La funzione 'leggi' aggiorna entrambe le variabili. La variabile  $cc2$  guida la scelta tra le alternative sintattiche di  $D$ .

*procedure N(in Ø; out v<sub>0</sub>)*  
*begin*

*if cc1 = '•' then leggi else errore end if*  
*l<sub>1</sub> := 1*  
     – inizializza var. locale contenente attr. destro di D;  
*D(l<sub>1</sub>; v<sub>0</sub>)*  
     – chiamata di D per costruire sottoalbero e calcolare v<sub>0</sub>;

*end*

*procedure D(in l<sub>0</sub>; out v<sub>0</sub>)*  
*begin*

*case cc2 of*

*'0,1': begin*  
     – caso D → BD  
*B(l<sub>0</sub>; v<sub>1</sub>)*  
*l<sub>2</sub> := l<sub>0</sub> + 1*       $\rightarrow$   $l_1 \in L_0$   
*D(l<sub>2</sub>; v<sub>2</sub>)*  
*v<sub>0</sub> := v<sub>1</sub> + v<sub>2</sub>*  
*end*  
*'¬': begin*  
     – caso D → B  
*B(l<sub>0</sub>; v<sub>1</sub>)*  
*v<sub>0</sub> := v<sub>1</sub>*  
*end*

*otherwise error*

*end*

*procedure B(in l<sub>0</sub>; out v<sub>0</sub>)*  
*begin*

*case cc1 of*

*'0': v<sub>0</sub> := 0*  
     – caso B → 0  
*'1': v<sub>0</sub> := 2<sup>-l<sub>0</sub></sup>*  
     – caso B → 1

*otherwise error*

*end*

Per lanciare il programma si chiama la procedura associata all'assioma.

Questo è lo schema generale delle procedure, alle quali si potrebbero applicare vari miglioramenti ovvii per un programmatore.

### Parsificatore ascendente con attributi

Si supponga che la sintassi sia LR(1). Volendo combinare la valutazione degli attributi con la costruzione dell'albero in ordine ascendente, si devono considerare diversi problemi: come garantire che le dipendenze tra gli attributi

siano compatibili con l'ordine di costruzione dell'albero, quando calcolare gli attributi e dove memorizzarli.

Iniziando dal problema di quando applicare le funzioni per il calcolo degli attributi, prima si espone l'impossibilità di valutare gli attributi destri, anche se la grammatica soddisfa la condizione  $L$  che consente l'analisi discendente. Si sa che, nella costruzione dell'albero, il parsificatore ascendente sceglie la produzione da applicare al momento della riduzione, quando si trova in un macrostato contenente la produzione marcata  $D_0 \rightarrow D_1 \dots D_r \bullet$ . Soltanto allora, e non prima, esso può scegliere le funzioni semantiche da applicare.

Il successivo problema da esaminare sono le dipendenze tra gli attributi. Subito prima della riduzione la pila contiene  $r$  elementi dalla cima, in corrispondenza con i simboli sintattici della parte destra. Supponendo che i valori degli attributi di  $D_1 \dots D_r$  siano disponibili, l'algoritmo può applicare le funzioni per calcolare i valori degli attributi sinistri di  $D_0$ .

Ma una difficoltà sorge per la valutazione degli attributi destri di  $D_1 \dots D_r$ . Si immagini che l'algoritmo stia per costruire e decorare il sottoalbero di  $D_1$ . In accordo con la visita a una scansione, un attributo destro  $\eta_{D_1}$  deve essere noto prima di valutare il sottoalbero di  $D_1$ . Ma esso può dipendere da un attributo destro  $\eta_0$  del padre  $D_0$ , di valore ignoto non esistendo ancora nell'albero sintattico la parte contenente il padre.

Il modo più semplice di superare la difficoltà è di imporre che la grammatica sia priva di attributi destri. Infatti gli attributi sinistri d'un nodo, avendo allora dipendenze dai solo attributi sinistri dei figli, sono facilmente calcolabili al momento della riduzione.

Venendo alla questione della memorizzazione, gli attributi possono essere collocati nella stessa pila usata dell'automa parsificatore, accanto alle informazioni sui nomi dei macrostati della macchina pilota. Ogni elemento della pila diviene così un record, contenente un campo sintattico e uno o più campi semanticici per gli attributi (o per dei puntatori verso gli attributi se essi sono memorizzati altrove). Ciò è mostrato nel prossimo esempio.

#### *Esempio 5.48. Calcolatrice, attributi solo sinistri.*

La sintassi dell'es. 4.57 (p. 218) di certe espressioni aritmetiche soddisfa la condizione  $LR(1)$ .

La seguente grammatica calcola il valore  $v$  dell'espressione o avverrà il predicato  $o$  (overflow) in caso di traboccamento. Il terminale  $a$  ha l'attributo lessicale  $v$  inizializzato con il valore della costante intera  $a$ . Entrambi gli attributi sono sinistri.

| sintassi                   | funzioni semantiche  |
|----------------------------|--|
| $E \rightarrow E + T$      | $o_0 := o_1 \text{ or } o_1 \text{ or } (v_1 + v_2 > maxint)$<br>$v_0 := \text{if } o_0 \text{ then } nil \text{ else } v_1 + v_2$ |
| $E \rightarrow T$          | $o_0 := o_1$<br>$v_0 := v_1$   |
| $T \rightarrow T \times a$ | $o_0 := o_1 \text{ or } (v_1 \times v_2 > maxint)$<br>$v_0 := \text{if } o_0 \text{ then } nil \text{ else } v_1 \times v_2$       |
| $T \rightarrow a$          | $o_0 := false$<br>$v_0 := valore(a)$   |

dove  $maxint$  è il massimo intero rappresentabile.

Una traccia del funzionamento dell'automa a pila esteso con i campi semantici è sotto mostrata per la frase  $a_3 + a_5$ , dove il pedice delle costanti è il loro valore.

| Pila        |                 | Stringa                |              |  |  |  |
|-------------|-----------------|------------------------|--------------|--|--|--|
| $I_0$       | $a_3$           | $+ a_5 \vdash$         |              |  |  |  |
| $I_0$       | $a_3$           | $I_4 + a_5 \vdash$     |              |  |  |  |
| $T$         |                 |                        |              |  |  |  |
| $I_0$       | $v = 3$         | $I_5 + a_5 \vdash$     |              |  |  |  |
| $o = false$ |                 |                        |              |  |  |  |
| $E$         |                 |                        |              |  |  |  |
| $I_0$       | $v = 3$         | $I_1 + a_5 \vdash$     |              |  |  |  |
| $o = false$ |                 |                        |              |  |  |  |
| $E$         |                 |                        |              |  |  |  |
| $I_0$       | $v = 3$         | $I_1 + I_2 a_5 \vdash$ |              |  |  |  |
| $o = false$ |                 |                        |              |  |  |  |
| $E$         |                 |                        |              |  |  |  |
| $I_0$       | $v = 3$         | $I_1 + I_2 v = 5$      | $I_4 \vdash$ |  |  |  |
| $o = false$ |                 |                        |              |  |  |  |
| $E$         |                 |                        |              |  |  |  |
| $I_0$       | $v = 3 + 5 = 8$ |                        |              |  |  |  |
| $o = false$ |                 |                        |              |  |  |  |

Al termine dell'analisi sintattica, la pila contiene gli attributi  $v$  e  $o$  della radice dell'albero.

Occorre osservare che in generale il divieto di usare gli attributi destri lede gravemente l'espressività della grammatica. La progettazione della grammatica, tenendo conto di questo limite, risulta spesso meno diretta e naturale, anche se, dal punto di vista teorico, ogni traduzione può essere riformulata senza fare uso di attributi destri.

Attributi destri senza dipendenze dal padre

Per migliorare un poco l'espressività, si possono riammettere gli attributi destri, limitando però le loro dipendenze nel modo seguente.

**Definizione 5.49. Condizione A per la valutabilità ascendente.**

Per ogni produzione  $p : D_0 \rightarrow D_1 \dots D_r$

1. la condizione L (p. 315) per la valutazione discendente è soddisfatta;
2. nessun attributo destro  $\eta_{D_k}$ ,  $1 \leq k \leq r$  d'un figlio dipende da un attributo destro  $\gamma_{D_0}$  del padre.

In positivo si può riformulare la condizione nel modo seguente.

Un attributo destro  $\eta_{D_k}$ ,  $1 \leq k \leq r$  può dipendere soltanto dagli attributi destri o sinistri dei simboli  $D_1 \dots D_{k-1}$ .

Se la grammatica soddisfa la condizione A, al momento della riduzione, essendo disponibili gli attributi sinistri dei nonterminali  $D_1, \dots, D_r$ , si possono calcolare nell'ordine:

1. gli attributi destri degli stessi nonterminali, nell'ordine  $1, 2, \dots, r$ ;
2. gli attributi sinistri del padre  $D_0$ .

Quest'ordine di calcolo degli attributi differisce da quello discendente perché gli attributi destri sono calcolati più tardi, in fase di riduzione.

Inoltre tale fatto consentirebbe maggiore libertà nella valutazione degli attributi destri in un ordine, diverso da quello naturale, purché in accordo con l'ordinamento topologico del grafo dei fratelli (p. 309), come è stato esposto nei valutatori a una scansione.

**5.6.8 Applicazioni tipiche delle grammatiche con attributi**

La traduzione guidata dalla sintassi è il modo di progettazione dei compilatori più comune. Le grammatiche con attributi possono essere usate vantaggiosamente per specificare in modo astratto le numerose operazioni che costituiscono l'analisi semantica, invece di codificarle direttamente nel compilatore. Non potendo qui esporre in modo completo la gamma delle operazioni che costituiscono l'analisi semantica d'un tipico linguaggio di programmazione, conviene schematizzare e illustrare alcune parti interessanti e rappresentative. Del resto, nell'analisi semantica dei linguaggi tecnici, molte parti sono ripetitive e sarebbe tedioso esporle per esteso.

I casi selezionati riguardano i controlli semanticci, la generazione del codice e l'impiego di informazioni semantiche per rendere deterministica l'analisi sintattica.

**Controlli semanticci**

Il linguaggio formale  $L_F$  definito dalla sintassi è soltanto una grossolana approssimazione per eccesso del linguaggio tecnico  $L_T$  da compilare, ossia vale

l'inclusione  $L_F \supset L_T$ . Il primo è un linguaggio libero dal contesto, mentre il secondo è il linguaggio definito informalmente nel manuale di riferimento. Esso, formalmente parlando, sta in una famiglia più complessa, quella contestuale. Senza qui ripetere le ragioni esposte alla fine del cap. 3, le sintassi contestuali non sono utilizzabili in pratica e ci si deve accontentare dell'approssimazione fornita da quelle libere.

Per meglio comprendere il senso delle approssimazioni, si pensi a un linguaggio  $L_T$  di programmazione. Le frasi di  $L_F$  sono sintatticamente corrette, ma possono violare molte prescrizioni del manuale del linguaggio, ad es. la compatibilità tra i tipi degli operandi d'una espressione, la corrispondenza tra i parametri formali e attuali d'una procedura, la corrispondenza tra una dichiarazione di variabile e il suo uso, ecc..

Il controllo di tali prescrizioni può essere fatto mediante opportune regole semantiche, che calcolano degli attributi booleani, detti *predicati semanticici*. La valutazione semantica del testo sorgente produce un predicato falso, se una prescrizione è violata. Si dice anche che è stato scoperto un *errore semantico statico*.

In generale i predicati semanticici dipendono funzionalmente da altri attributi che rappresentano certe proprietà del testo sorgente. Si pensi alla concordanza tra la dichiarazione d'una variabile e il suo uso in un assegnamento. Poiché nel testo la dichiarazione della variabile è distante dagli assegnamenti che la usano, il tipo con cui è dichiarata la variabile deve essere messo in un attributo, detto *tabella dei simboli* o *ambiente*, che sarà propagato sull'albero per raggiungere i punti in cui la variabile è usata negli assegnamenti o in altri costrutti. Tale propagazione può parere inefficiente, ma è soltanto concettuale, perché in pratica nel compilatore la tabella dei simboli è realizzata come una struttura dati (o un oggetto) globale, visibile da tutte le funzioni semantiche. Il seguente esempio schematizza la creazione della tabella dei simboli e il suo impiego nel controllo delle variabili usate in un assegnamento.

#### *Esempio 5.50.* Tabella dei simboli e controllo dei tipi.

L'esempio tratta le dichiarazioni di variabili scalari e di vettori, che possono essere usate negli assegnamenti. Per la correttezza semantica, valgono le seguenti prescrizioni:

1. una variabile non può essere dichiarata due o più volte;
2. una variabile non può essere usata prima della sua dichiarazione;
3. sono permessi assegnamenti soltanto tra variabili scalari e tra vettori della stessa dimensione.

La sintassi, in forma astratta, si accontenta di distinguere le dichiarazioni delle variabili dagli usi delle stesse. La tabella dei simboli ha come chiave d'accesso il nome  $n$  della variabile e contiene, per ogni variabile dichiarata, il descrittore  $descr$ , con il tipo (scalare o vettore) e, se del caso, la dimensione del vettore. Durante la sua costruzione, la tabella è rappresentata dall'attributo  $t$ . Il predicato  $dd$  denuncia una doppia dichiarazione; il predicato  $ai$  la incompatibilità

tra la parte sinistra e destra d'un assegnamento.

L'attributo *t* è propagato in tutto il programma per i necessari controlli.

Attributi:

| <i>attributo</i>                        | <i>tipo</i> | <i>simb. associati</i> |
|---|-------------|------------------------|
| <i>n</i> , nome d'una variabile         | sinistro    | <i>id</i>              |
| <i>v</i> , valore d'una costante        | sinistro    | <i>const</i>           |
| <i>dd</i> , bool., doppia dichiarazione | sinistro    | <i>D</i>               |
| <i>ai</i> , bool., incompatibilità      | sinistro    | <i>D</i>               |
| <i>descr</i> , descrittore              | sinistro    | <i>D, L, R</i>         |
| <i>t</i> , tabella dei simboli          | destro      | <i>A, P</i>            |

La semantica delle dichiarazioni *D* rende vero il predicato *dd* se la variabile dichiarata è già presente nella tabella. Altrimenti il descrittore della variabile è costruito e passato al nodo padre insieme al nome della variabile.

La parte sinistra *L* e destra *R* d'un assegnamento *A* hanno lo stesso attributo *descr*, che descrive il termine ivi presente: variabile con o senza indice o costante. Se invece un nome non è trovato in tabella, per convenzione il descrittore denuncia l'errore.

La semantica dell'assegnamento esegue il controllo di compatibilità, calcolando il predicato *ai*.

| sintassi                  | Grammatica  |
|---------------------------|---|
|                           | <i>funzioni semantiche</i>  |
| $S \rightarrow P$         | $t_1 := \emptyset$ – tabella inizialmente vuota   |
| $P \rightarrow DP$        | $t_2 := inserisci(t_0, n_1, descr_1)$<br>– aggiunge nome e descrittore alla tabella   |
| $P \rightarrow AP$        | $t_1 := t_0$<br>$t_2 := t_0$<br>– propaga la tabella nei due sottoalberi  |
| $P \rightarrow \epsilon$  |   |
| $D \rightarrow id$        | -- dichiarazione di variabile scalare<br>$dd_0 := presente(t_0, n_{id})$<br>if $\neg dd_0$ then $descr_0 := 'sca'$<br>$n_0 := n_{id}$   |
| $D \rightarrow id[const]$ | -- dichiarazione di variabile vettoriale<br>$dd_0 := presente(t_0, n_{id})$<br>if $\neg dd_0$ then $descr_0 := ('vet', v_{const})$<br>$n_0 := n_{id}$   |
| $A \rightarrow L := R$    | $aio_0 := \neg \langle descr_1 \text{ è compatibile con } descr_2 \rangle$  |
| $L \rightarrow id$        | $descr_0 := < \text{tipo di } n_{id} \text{ in } t_0 >$   |
| $L \rightarrow id[id]$    | if $\langle \text{tipo di } n_{id_1} \text{ in } t_0 \rangle = 'vet' \wedge \langle \text{tipo di } n_{id_2} \text{ in } t_0 \rangle = 'sca'$ then<br>$descr_0 := \langle \text{descr di } n_{id_1} \text{ in } t_0 \rangle$ else <i>errato</i>                                   |
| $R \rightarrow id$        | -- uso di variabile scalare o vettoriale<br>$descr_0 := \langle \text{tipo di } n_{id} \text{ in } t_0 \rangle$   |
| $R \rightarrow const$     | -- uso d'una costante<br>$descr_0 := 'sca'$   |
| $R \rightarrow id[id]$    | -- uso di variabile con indice<br>if $\langle \text{tipo di } n_{id_1} \text{ in } t_0 \rangle = 'vet' \wedge \langle \text{tipo di } n_{id_2} \text{ in } t_0 \rangle = 'sca'$ then<br>$descr_0 := \langle \text{descr di } n_{id_1} \text{ in } t_0 \rangle$ else <i>errato</i> |

Il controllo della compatibilità tra parte sinistra e destra dell'assegnamento è specificato in pseudocodice: le condizioni che falsificano il predicato sono quelle elencate sopra ai punti 2. e 3.

Nel testo sintatticamente corretto:

$$\begin{array}{cccccccc}
 D_1 & D_2 & D_3 & A_4 & A_5:ai=true & D_6 & D_7:dd=true & A_8:ai=true \\
 \overbrace{a[10]} & \overbrace{i} & \overbrace{b} & \overbrace{i := 4} & \overbrace{c := a[i]} & \overbrace{c[30]} & \overbrace{i} & \overbrace{a := c}
 \end{array}$$

sono stati riconosciuti vari errori semanticci negli assegnamenti  $A_5$ ,  $A_8$  e nella dichiarazione  $D_7$ .

Diversi perfezionamenti e completamenti sarebbero necessari in un compilatore reale, tra i quali in particolare i seguenti. Per rendere più precisa la diagnostica, si possono discriminare i generi di errori (variabile indefinita, tipo non compatibile, dimensione errata, ...). Si deve certamente comunicare al programmatore l'indicazione del punto (numero di linea) in cui l'errore si

è manifestato. Arricchendo la grammatica di altri attributi e funzioni si può facilmente perfezionare la diagnostica. In particolare ciascun predicato, calcolato in un punto dell'albero decorato, unitamente alla coordinata del punto, può essere propagato verso la radice, dove un'opportuna funzione emetterà la diagnostica in maniera coerente e comprensibile.

Altri errori o difetti non sono trattati nell'esempio, come il fatto che una variabile sia priva di valore, o che la stessa variabile riceva valore in due assegnamenti senza che il primo valore sia utilizzato. Nel compilatore questo tipo di controlli è di solito impostato con un metodo diverso e più preciso delle grammatiche con attributi, la cosiddetta *analisi statica del programma*, che concluderà il capitolo e il libro.

Infine il programma, anche se ha passato tutti i controlli statici, può provocare degli *errori dinamici* durante la sua esecuzione. Si veda, nel frammento

```
array a[10]; ... read(i); a[i] := ...
```

l'istruzione di lettura che può assegnare alla variabile *i* un valore esterno all'intervallo 1...10, evenienza certo non scopribile durante la compilazione.

### 5.6.9 Generazione del codice

Poiché lo scopo della compilazione è la traduzione del programma sorgente in quello pozzo, una parte essenziale del processo è la generazione delle istruzioni del secondo. Le situazioni possibili si differenziano, a seconda della natura dei due linguaggi e della distanza tra i loro costrutti. Se le differenze sono piccole, la traduzione può essere fatta direttamente dal parsificatore, come si è visto in 5.5.1 per le conversioni tra le rappresentazioni infisse e polacche delle espressioni aritmetiche.

Ben più difficile è la traduzione d'un linguaggio di alto livello come Java in linguaggio macchina, perché la distanza tra i due è tanto grande da rendere necessario un processo di traduzione a più stadi. Ogni stadio traduce un linguaggio intermedio in un altro. Il primo stadio ha Java come linguaggio sorgente, e l'ultimo ha come pozzo il linguaggio macchina del processore. La gamma dei linguaggi intermedi impiegati dai compilatori è piuttosto ampia: rappresentazioni testuali magari in forma prefissa o postfissa, alberi o grafi, rappresentazioni simili al codice assemblatore d'una macchina, tabelle, ecc. Il primo stadio è un traduttore guidato dalla sintassi del linguaggio Java. Gli ultimi stadi scelgono in modo ottimale le istruzioni macchina, allo scopo di massimizzare la velocità o minimizzare il consumo d'energia elettrica del programma compilato.<sup>22</sup> sono descritti ad es. in [4].

I primi stadi del compilatore sono indipendenti dalle caratteristiche della macchina e sono chiamati il tronco o il fronte.<sup>23</sup>

<sup>22</sup> Esso è di solito progettato con algoritmi specializzati, di riconoscimento di forme nell'albero del linguaggio intermedio. Tali algoritmi di *tree pattern matching*

<sup>23</sup> front-end.

Gli ultimi stadi dipendono dalla macchina e sono chiamati il *retro*<sup>24</sup> del compilatore. Esso comprende minimalmente il generatore di codice e l'assegnatore dei registri fisici della macchina. Uno stesso tronco può interfacciarsi con più retrocompilatori, orientati verso macchine diverse.

Un piccolissimo saggio delle problematiche della generazione del codice è offerto dal prossimo esempio di traduzione dei costrutti di controllo di alto livello in istruzioni di salto.

*Esempio 5.51.* Traduzione delle istruzioni di controllo in salti.

Le strutture di controllo governano l'ordine e la scelta delle istruzioni da eseguire. I costrutti *if then else*, *while do*, ecc. vanno convertiti in istruzioni di salto, condizionale e non. L'istruzione condizionale 'jump-if-false' ha due argomenti: un registro *rc* contenente l'esito della condizione, e l'etichetta bersaglio cui saltare.

Il nonterminale *L* sta per una lista di frasi. Per ogni costrutto, il traduttore ha bisogno di nuove etichette di arrivo dei salti, diverse dalle etichette usate in altri costrutti. Ciò si può ottenere invocando una funzione *nuovo* che, a ogni invocazione, assegna all'attributo *n* un diverso intero.

La traduzione d'un costrutto è posta nell'attributo *tr*, concatenando (segno •) le traduzioni delle diverse parti e inserendo le istruzioni di salto con le nuove etichette generate. Queste hanno la forma *e397, f397, i23...*, dove il suffisso numerico è quello generato come detto sopra. Il registro *rc* è un attributo di *cond*.

### Istruzioni condizionali

La grammatica delle frasi condizionali *I* è la seguente:

| <i>sintassi</i>   | <i>funzioni semantiche</i>  |
|---|---|
| $F \rightarrow I$   | $n_1 := \text{nuovo}$   |
| $I \rightarrow \text{if } cond$<br>then $L_1$<br>else $L_2$ | $tr_0 := tr_{cond} \bullet \text{'jump-if-false'} \ rc_{cond} \ ', e' \bullet n_0 \bullet$<br>$tr_{L_1} \bullet \text{'jump'} \ 'f' \bullet n_0 \bullet$<br>$'e' \bullet n_0 \bullet \text{':}' \ tr_{L_2} \bullet$<br>$'f' \bullet n_0 \bullet \text{':}'$ |

Si suppone che la traduzione della condizione booleana *cond* e delle altre frasi del linguaggio, qui non specificate, stiano in altre parti della grammatica.

Si mostra la traduzione d'un frammento di programma, supponendo  $n = 7$ :

|                    |   |
|--------------------|---|
| if $a > b$         | $tr(a > b);$  |
| then $a := a - 1;$ | $\text{jump-if-false } rc, e7 ; tr(a := a - 1); \text{ jump } f7 ;$ |
| else $a := b;$     | $e7 : tr(a := b);$<br>$f7 : \text{-- seguito del progr.}$           |

<sup>24</sup>back-end.

Si ricordi che  $tr(\dots)$  sta per la traduzione d'un costrutto nel codice macchina. Il registro  $rc$  è scelto dal compilatore quando traduce l'espressione booleana  $a > b$ .

### Istruzioni iterative

La grammatica del costrutto *while do* è simile alla precedente:

| <i>sintassi</i>  | <i>funzioni semantiche</i>   |
|--|--|
| $F \rightarrow W$                                      | $n_1 := \text{nuovo}$  |
| $W \rightarrow \text{while } cond$<br>do<br>$L$<br>end | $tr_0 := 'i' \bullet n_0 \bullet ':' \bullet tr_{cond} \bullet$<br>$'\text{jump-if-false}' \bullet rc_{cond} \bullet, f' \bullet n_0 \bullet$<br>$tr_L \bullet '\text{jump}' \bullet 'i' \bullet n_0 \bullet ;$<br>$f' \bullet n_0 \bullet :'$ |

Traduzione d'un frammento di programma (supponendo che la funzione *nuovo* restituisca il valore 8):

|  |   |
|--|---|
| $\text{while } (a > b) \text{ do}$<br>$a := a - 1;$<br>end while | i8: $tr(a > b);$<br>$\text{jump-if-false } rc, f8;$<br>$tr(a := a - 1);$<br>$\text{jump i8;}$<br>   f8: - - seguito del prog. |
|--|---|

In modo simile si trattano le altre strutture di controllo iterative o condizionali.

Le traduzioni così prodotte sono inefficienti e devono solitamente essere migliorate ad opera dell'ottimizzatore.<sup>25</sup> Un esempio banale è la condensazione d'una cascata di più salti incondizionali in una sola istruzione di salto.

### 5.6.10 Analisi sintattica guidata dalla semantica

Nello schema classico della compilazione, l'analisi sintattica è un passo separato, che costruisce gli alberi sintattici sui quali opererà l'analisi semantica. Ma vi sono circostanze, quando la sintassi è ambigua, in cui la parsificazione non può essere condotta con successo da sola, perché produrrebbe molti alberi sintattici diversi, tra i quali l'analisi semantica non saprebbe come scegliere. In campo tecnico tale evenienza è rara, perché di solito i linguaggi sono progettati in modo da rendere deterministica la sintassi, ma nel trattamento dei testi in lingua naturale la situazione si ribalta, perché la sintassi da sola presenta quasi ovunque un elevato grado di ambiguità.

Un'organizzazione diversa del rapporto tra analisi sintattica e semantica si

<sup>25</sup>L'ottimizzatore è la parte più complessa e articolata d'un moderno compilatore; si può vedere [4], [36] e [1].

offre come alternativa alla precedente, quando la sintassi di riferimento del linguaggio sorgente presenta delle situazioni indeterministiche, o addirittura ambigue, che non possono essere risolte neanche con una prospezione illimitata del parsificatore.

Per un linguaggio tecnico ben progettato si può sempre ritenere che le frasi non siano ambigue semanticamente, cioè che abbiano un solo significato. Sotto quest'ipotesi, l'incertezza tra i vari alberi sintattici d'una frase può essere spesso eliminata, già durante la parsificazione, sfruttando delle informazioni di natura semantica, via via che divengono disponibili al compilatore.

Riferendosi all'analisi sintattica discendente, quando il parsificatore non saprebbe scegliere tra due produzioni alternative, perché i loro insiemi guida  $LL(k)$  sono sovrapposti (p. 180), esso può risolvere il dubbio consultando il valore d'un attributo semantico, che funge da *predicato guida*. Tale attributo deve essere stato calcolato prima di tale momento, dall'analizzatore sintattico-semantic.

Quest'organizzazione ha dei punti di contatto con il metodo di valutazione degli attributi in più stadi (p. 311).

La totalità degli attributi è ripartita in due insiemi: i predicati guida e gli attributi da cui essi dipendono devono essere valutati nel primo stadio durante la parsificazione. I restanti attributi possono essere valutati dopo che l'albero sintattico, unico, è stato costruito. Affinché gli attributi del primo insieme siano calcolabili durante la parsificazione, essi, come si ricorda, devono soddisfare la condizione  $L$  (p. 315).

Di conseguenza il predicato guida sarà calcolabile quando va fatta la scelta tra due alternative per espandere il nonterminale  $D_i, 1 \leq i \leq r$ , nella produzione  $D_0 \rightarrow D_1 \dots D_i \dots D_r$ . Poiché il parsificatore opera in profondità da sinistra a destra, l'albero sintattico sarà stato costruito dalla radice fino ai sottoalberi  $D_1 \dots D_{i-1}$ . Per la condizione  $L$  il predicato guida può dipendere dagli attributi destri di  $D_0$  e dagli attributi destri o sinistri dei simboli che, nella parte destra della produzione, precedono la radice del sottoalbero  $D_i$  da costruire. Il prossimo esempio illustra l'uso dei predicati guida.

*Esempio 5.52.* Un linguaggio senza interpunkzione.

Nel linguaggio PLZ-SYS<sup>26</sup> mancano le virgolette e gli altri segni di interpunkzione, con la conseguente ambiguità nella lista dei parametri d'una procedura. Sono elencate tre possibili interpretazioni degli argomenti (parametri formali) della procedura  $P$ . Ogni argomento in questo linguaggio è specificato con il suo tipo, e più argomenti possono condividere il tipo.

$$P \text{ proc } (X Y T1 Z T2) \left\{ \begin{array}{l} 1. X \text{ ha il tipo } Y; T1, Z \text{ hanno il tipo } T2; \\ 2. X, Y \text{ hanno il tipo } T1; Z \text{ ha il tipo } T2; \\ 3. X, Y, T1, Z \text{ hanno il tipo } T2. \end{array} \right.$$

Per fortuna le dichiarazioni dei tipi nel linguaggio PLZ-SYS devono precedere le dichiarazioni delle procedure. Se ad es. le dichiarazioni di tipo, che nel testo

<sup>26</sup> Progettato negli anni 1970 per un microprocessore a 8 bit di poche risorse.

stanno prima della dichiarazione di  $P$ , sono

type  $T_1$  record ... end type  $T_2$  = record ... end

la possibilità 1. cade perché  $Y$  non è un tipo, mentre  $T_1$  non è una variabile. Similmente la possibilità 3 si può escludere e l'ambiguità è eliminata.

Resta da precisare come sfruttare la conoscenza delle precedenti dichiarazioni, al fine di guidare la scelta dell'analizzatore sintattico-semantico tra i casi 1, 2 e 3.

All'interno della parte dichiarativa  $D$  del linguaggio PLZ-SYS, le parti rilevanti della sintassi sono due:  $T$ , le dichiarazioni di tipo e  $I$ , l'intestazione d'una procedura (non interessa qui studiare il corpo della procedura). La semantica inserisce i descrittori dei tipi dichiarati nella tabella dei simboli  $t$ , gestita come un attributo sinistro. Al solito  $n$  è il nome o chiave di un identificatore.

Al termine dell'analisi delle dichiarazioni di tipo, la tabella deve essere propagata verso le successive parti del programma, tra le quali le intestazioni che qui interessano. L'attributo sinistro  $t$  è copiato nell'attributo destro  $td$  per la propagazione verso il basso dell'albero. Il descrittore  $descr$  del tipo di ogni identificatore consente al parsificatore di scegliere la produzione corretta.

Per non ingrandire troppo l'esempio si fanno varie semplificazioni: l'ambiente di visibilità degli oggetti dichiarati è unico, ogni tipo è dichiarato come un record non ulteriormente specificato; si omette il controllo sulle doppie dichiarazioni, si omette l'inserimento delle dichiarazioni di procedura (e dei relativi argomenti) nella tabella dei simboli.

| sintassi  | funzioni semantiche   |
|---|---|
| - parte dichiarativa<br>$D \rightarrow TI$  | $td_I := t_T$<br>-- la tabella dei simboli è copiata                                      |
| - dichiarazioni di tipo<br>$T \rightarrow \text{type } id = \text{record} \dots \text{end}$ | $t_0 := \text{inserisci}(t_2, n_{id}, 'type')$<br>-- inserimento del descr. nella tabella |
| $T \rightarrow \epsilon$  | $t_0 := \emptyset$  |
| - intestazione di proc.<br>$I \rightarrow id \text{ proc } (L) I$                           | $td_L := td_0$<br>$td_3 := td_0$<br>-- la tabella è passata a $L$ e a $I \equiv 3$        |
| $I \rightarrow \epsilon$  |   |
| - lista di parametri<br>$L \rightarrow V \text{ type\_id } L$                               | $td_V := td_0$<br>$td_3 := td_0$  |
| $L \rightarrow \epsilon$  |   |
| - lista di var. (stesso tipo)<br>$V \rightarrow var\_id V$                                  | $td_1 := td_0$<br>$td_2 := td_0$  |
| $V \rightarrow var\_id$   | $td_1 := td_0$  |
| $type\_id \rightarrow id$   |   |
| $var\_id \rightarrow id$  |   |

La coppia di produzioni di  $V$  viola la condizione  $LL(2)$  poiché gli identificatori di tipo e di variabile sono sintatticamente indistinguibili:

$V \rightarrow var\_id V$  : inizia con  $var\_id var\_id$  ossia con  $id id$   
 $V \rightarrow var\_id$  : inizia con  $var\_id$  ossia  $id$ , seguito da  $type\_id$  ossia  $id$

Il parsificatore deve ricorrere a un test semantico per scegliere l'alternativa, nel modo sotto specificato.

Siano  $cc_1, cc_2$  il carattere terminale (o meglio il lesema) corrente e quello successivo.

| produzione                  | predicato guida   |
|-----------------------------|---|
| 1 $V \rightarrow var\_id V$ | ( il descr. di $cc_2$ nella tabella $td_0$ ) \neq 'type' \wedge<br>( il descr. di $cc_1$ nella tabella $td_0$ ) \neq 'type' |
| 1' $V \rightarrow var\_id$  |   |
| 2 $V \rightarrow var\_id$   | ( il descr. di $cc_2$ nella tabella $td_0$ ) = 'type' \wedge<br>( il descr. di $cc_1$ nella tabella $td_0$ ) \neq 'type'    |
| 2' $V \rightarrow var\_id$  |   |

Le clausole 1 e 2, mutuamente esclusive, sono i predicati guida per la scelta tra le alternative 1 e 2. Le clausole 1' e 2' sono un predicato semantico che controlla che l'identificatore associato a  $var\_id$  non sia di tipo.

Anche alla produzione  $L \rightarrow V \text{ type\_id } L$  si può associare un predicato semantico per controllare che nella tabella il tipo di  $type\_id \equiv cc_2$  valga 'type'.

In questo modo il parsificatore può costruire deterministicamente l'albero, facendosi guidare dai valori via via disponibili degli attributi semantici.

## 5.7 Analisi statica dei programmi

In quest'ultima parte del libro si studia una tecnica di analisi e ottimizzazione dei programmi, impiegata in tutti i compilatori che traducono un linguaggio di programmazione in un codice macchina.

Il primo stadio, il tronco del compilatore, traduce un programma in una rappresentazione intermedia che si avvicina al linguaggio macchina. Il programma intermedio è poi analizzato da altri stadi che possono avere diverse finalità a seconda delle circostanze:

- verifica*, per esaminare ulteriormente la correttezza del programma;
- ottimizzazione*, per trasformare il programma onde renderlo più efficiente, ad es. assegnando in modo ottimale i registri del processore alle variabili del programma;
- schedulazione*, per cambiare l'ordine delle istruzioni in modo da meglio sfruttare le "pipeline" e le unità funzionali del processore, evitando che tali risorse siano in certi momenti inattive e in altri momenti troppo contese.

Questi compiti hanno in comune la rappresentazione del programma sotto forma d'un grafo detto *di controllo (del flusso)*<sup>27</sup>, simile allo schema di flusso o a blocchi del programma, grafo che conviene concettualizzare come un automa. Ma la prospettiva è del tutto diversa da quella della traduzione guidata dalla sintassi, perché tale automa non definisce l'intero linguaggio sorgente della traduzione, bensì un ben preciso programma, sul quale si fissa l'attenzione. Una stringa riconosciuta dall'automa è la sequenza temporale delle operazioni di calcolo che tale programma può eseguire, ossia una traccia della sua esecuzione.

L'*analisi statica* consiste nello studio di certe proprietà del grafo di controllo d'un programma, mediante i metodi della teoria degli automi, della logica o della statistica. Nella seguente breve esposizione soltanto i primi sono considerati.

### 5.7.1 Il programma come automa

Nel grafo di controllo d'un programma ogni nodo è un'istruzione. Di solito le istruzioni che si considerano sono più semplici di quelle d'un linguaggio programmatico, perché sono quelle prodotte dalla compilazione nella rappresentazione intermedia. Gli operandi delle istruzioni sono variabili semplici e costanti (non ci sono tipi di dati aggregati). Le istruzioni tipiche sono assegnamenti a variabili, e semplici espressioni aritmetiche, relazionali e logiche,

<sup>27</sup>Control-flow graph.

di solito con un solo operatore.

In questo libro l'analisi è *intraprocedurale*, ossia il grafo rappresenta un solo sottoprogramma alla volta, ma, nello studio più avanzato, si considerano anche le chiamate di sottoprogrammi, e l'analisi del programma è detta *interprocedurale*.

Se nell'esecuzione l'istruzione  $p$  può essere immediatamente seguita dall'istruzione  $q$ , il grafo ha un arco orientato da  $p$  a  $q$ . In altre parole un arco rappresenta la relazione di precedenza tra due istruzioni:  $p$  è il *predecessore* e  $q$  il *successore*.

La prima istruzione eseguita del programma è il suo punto d'*entrata*, ossia il *nodo iniziale*, che è senza predecessori; un'istruzione priva di nodi successori è un punto d'*uscita* del programma o *nodo finale*.

Le istruzioni non condizionali hanno al più un successore. Quelle condizionali hanno due successori (più di due nel caso dei test a più vie come l'istruzione switch del linguaggio C).

Un'istruzione avente più predecessori è un punto di *confluenza* di più rami del programma.

Il grafo di controllo non è però una rappresentazione fedele e completa del programma, ma soltanto un'astrazione sufficiente per analizzare certe proprietà che interessano. Infatti diverse informazioni del programma scompaiono dal grafo di controllo.

- Il valore vero o falso che fa scegliere il successore di un'istruzione condizionale non è rappresentato.
- L'operazione (aritmetica, di lettura o scrittura, ecc.) compiuta da un'istruzione è sostituita dalla seguente astrazione:
  - si dice che un assegnamento di valore a una variabile, mediante un'istruzione di assegnamento o di lettura, *definisce* quella variabile;
  - si dice che una comparsa della variabile in un'espressione, nella parte destra d'un assegnamento o in una scrittura, *usa* quella variabile;
  - nel grafo il nodo che rappresenta un'istruzione  $p$  è associato a due insiemi: l'insieme  $def(p)$  delle variabili definite e l'insieme  $usa(p)$  delle variabili usate dall'istruzione.

Invece la particolare operazione svolta può spesso essere trascurata in questo modello.

Così l'istruzione  $p : a := a \oplus b$ , dove  $\oplus$  è un generico operatore, è nel grafo di controllo un nodo associato all'informazione:

$$def(p) = \{a\}, usa(p) = \{a, b\}$$

In tale astrazione le istruzioni  $read(a)$  e  $a := 7$  sono associate alla stessa informazione:  $def = \{a\}$ ,  $usa = \emptyset$ .

Per chiarire i concetti e le applicazioni di questo metodo di analisi, conviene appoggiarsi a un esempio completo.

*Esempio 5.53.* Schema a blocchi e grafo di controllo.

Si mostra un (sotto)programma, lo schema a blocchi e il grafo di controllo.

*programma:*

$a := 1$

e1 :  $b := a + 2$

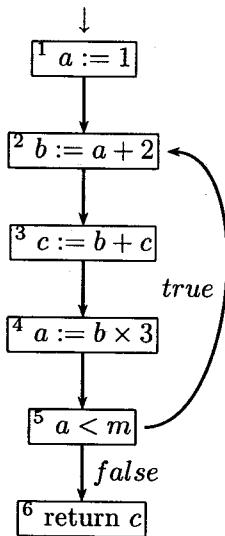
$c := b + c$

$a := b \times 3$

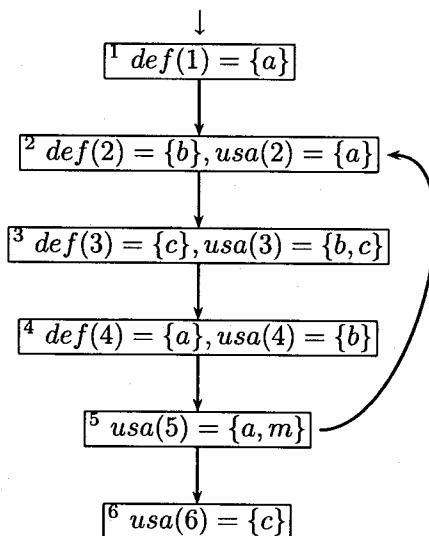
if  $a < m$  goto e1

return c

*schemà a blocchi:*



*grafo di controllo A:*



Per brevità nel grafo di flusso le istruzioni non sono riportate, ma soltanto gli insiemi delle variabili definite e usate.

L'istruzione 1 non ha predecessori, è l'entrata del (sotto)programma, e l'istruzione 6, senza successori è l'uscita del programma (o nodo finale). Il nodo 5 ha due successori, mentre il nodo 2 è la confluenza di due predecessori.

L'insieme  $usa(1)$  è vuoto, così come  $def(5)$ .

**Definizione 5.54.** *Linguaggio del grafo di controllo.*

*L'automa finito A, rappresentato dal grafo di controllo, ha come alfabeto terminale l'insieme I delle istruzioni, ciascuna schematizzata da una terna come  $\langle 2, def = \{b\}, usa = \{a\} \rangle$ ; per brevità spesso si prenderà soltanto la prima componente della terna, il numero o etichetta che identifica l'istruzione.*

*Gli stati dell'automa (come in un grafo sintattico p. 172) non hanno un nome esplicito.*

*Lo stato iniziale è la freccia entrante nel nodo d'entrata.*

*Gli stati finali sono quelli privi di successori.*

*Il linguaggio formale  $L(A)$  riconosciuto dall'automa contiene le stringhe di alfabeto I che etichettano un cammino dall'entrata all'uscita del grafo. Ta-*

*le cammino rappresenta una sequenza di istruzioni che la macchina potrebbe eseguire quando si lancia il programma.*

Poiché ogni nodo porta un'etichetta distinta, il linguaggio formale  $L(A)$  appartiene alla famiglia dei linguaggi locali (p. 127), una sottofamiglia della famiglia dei linguaggi regolari  $REG$ .

Nell'esempio l'alfabeto terminale è  $I = \{1 \dots 6\}$ . Un cammino riconosciuto è:

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \equiv 1234523456$$

L'insieme dei cammini è il linguaggio  $1(2345)^+6$ .

#### *Approssimazione cautelativa*

Anche questa, a meglio vedere, è soltanto una approssimazione, perché non è sempre vero che ogni cammino del grafo sia eseguibile dal programma, a causa del fatto che il grafo di controllo trascura le condizioni booleane che determinano la scelta dei successori d'un nodo. Un esempio banale è il programma

1 : if  $a ** 2 \geq 0$  then  $istr_2$  else  $istr_3$

in cui il linguaggio formale accettato dall'automa contiene i due cammini  $\{12, 13\}$ , ma il cammino 13 non è eseguibile, perché il quadrato non può essere negativo.

L'analisi statica, a causa di questa approssimazione, può portare a conclusioni pessimistiche, nel senso che potrebbe scoprire delle anomalie spurie in certi cammini che in realtà non possono essere mai eseguiti dal programma.

D'altra parte è in generale indecidibile se un cammino del grafo di controllo d'un programma possa o meno essere eseguito; ciò equivarrebbe infatti a decidere se esistono certi valori delle variabili d'ingresso del programma, che causano l'esecuzione di quel cammino, ma il problema è riconducibile a quello dell'alt d'una macchina di Turing.

Non potendo dunque sapere in generale quali cammini siano eseguibili e quali no, sarebbe più grave se l'analisi statica trascurasse certi cammini, perché allora non scoprirebbe gli errori che potrebbero manifestarsi in esecuzione.

In conclusione l'esame di tutti i cammini (dall'entrata all'uscita) è un'approssimazione cautelativa allo studio del programma, nel senso che può portare a diagnosi di errori inesistenti o all'assegnazione prudenziale di risorse in realtà non necessarie, ma non trascura mai le reali condizioni di errore.

Infine nell'analisi statica si fa l'ipotesi che l'automa sia pulito (p. 102), cioè che ogni istruzione giaccia su un cammino che dall'entrata (per ipotesi unica) porta a un'uscita. In caso contrario, il programma potrebbe presentare uno dei seguenti difetti: per certe esecuzioni non raggiungere la fine; contenere delle istruzioni che non possono essere mai eseguite (cosiddetto *codice irraggiungibile*).

### 5.7.2 Intervalli di vita delle variabili

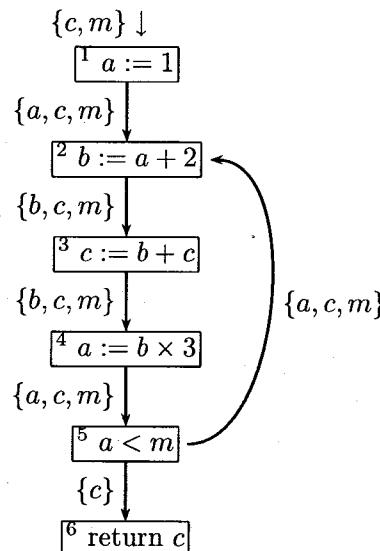
Un compilatore di buona qualità ripassa più volte sul programma intermedio per migliorarlo. Un'analisi molto redditizia ai fini di vari miglioramenti del programma è lo studio degli intervalli di vita<sup>28</sup> delle variabili.

**Definizione 5.55.** Una variabile  $a$  è viva in un punto  $p$  del programma se nel grafo esiste un cammino da  $p$  a un altro nodo  $q$  tale che

- il cammino non passa per un'istruzione  $r, r \neq q$  che definisce  $a$ , ossia tale che  $a \in \text{def}(r)$   $\wedge$
- l'istruzione  $q$  fa uso di  $a$ , ossia  $a \in \text{usa}(q)$ .

In sostanza una variabile è viva in un certo punto se qualche istruzione che potrebbe essere eseguita successivamente fa uso del valore che la variabile ha in quel punto.

Per capire l'utilità di questo concetto si immagini che l'istruzione  $p$  sia un assegnamento  $a := b \oplus c$  e si voglia sapere se qualche istruzione usa il valore di  $a$  definito in  $p$ , ovvero se  $a$  è viva sull'arco uscente dal nodo  $p$ . In caso negativo, l'assegnamento è *inutile* e può essere cancellato dal programma. Se poi nessuna delle variabili usate in  $p$  fosse usata in altre istruzioni diverse da  $p$ , anche le istruzioni che definiscono tali variabili potrebbero divenire inutili. La prossima figura riporta le variabili vive nei punti, ossia sugli archi del grafo, del programma di p. 331.



<sup>28</sup>Liveness.

La figura riporta le variabili vive all'entrata e all'uscita delle istruzioni. Ad es. la variabile  $c$  è viva all'ingresso del nodo 1 perché esiste il cammino 123, tale che  $c \in usa(3)$  e né 1 né 2 definiscono  $c$ .

La variabile  $a$  è viva negli intervalli (ossia cammini del grafo) 12 e 452; non è viva negli intervalli 234 e 56, e così via.

Si dice che una variabile è viva all'uscita d'un nodo, se è viva su uno degli archi uscenti dal nodo. Similmente è viva all'entrata d'un nodo se è viva su uno degli archi entranti nel nodo.

Così all'uscita di 5 sono vive le variabili  $\{a, c, m\} \cup \{c\}$ .

### Calcolo degli intervalli di vita

Sia  $I$  l'insieme delle istruzioni. Siano  $D(a)$  e  $U(a)$  gli insiemi delle istruzioni che rispettivamente definiscono e usano la variabile  $a$ .

La proprietà che  $a$  sia viva all'uscita del nodo  $p$  è equivalente alla seguente condizione sul linguaggio accettato dall'automa:

esiste nel linguaggio  $L(A)$  una frase  $x = upvqw$  tale che

$$u \in I^* \wedge p \in I \wedge v \in (I \setminus D(a))^* \wedge q \in U(a) \wedge w \in I^*$$

La differenza insiemistica contiene tutte le istruzioni che non definiscono la variabile  $a$ .

L'insieme di tutte le frasi  $x$  che soddisfano la condizione è un linguaggio regolare  $L_p \subseteq L(A)$ , che può essere definito come l'intersezione

$$L_p = L(A) \cap R_p \quad (5.1)$$

dove il linguaggio  $R_p$ , pure regolare, è definito come segue.

$$R_p = I^* p (I \setminus D(a))^* U(a) I^*$$

L'espressione 5.1 prescrive che il carattere  $p$  sia seguito da un carattere  $q$  scelto tra  $U(a)$  e che gli eventuali caratteri interposti tra  $p$  e  $q$  non appartengano a  $D(a)$ .

Per sapere dunque se  $a$  sia viva all'uscita da  $p$ , basta vedere se il linguaggio  $L_p$  non è vuoto.

Gli algoritmi del cap. 3 permetterebbero di costruire il riconoscitore del linguaggio  $L_p$ , ossia la macchina prodotto cartesiano che riconosce l'intersezione. Se in tale macchina non vi sono cammini dallo stato iniziale a uno stato finale, il linguaggio  $L_p$  è vuoto. Ma tale procedimento, tenuto conto delle dimensioni dei programmi da analizzare, rischia di essere troppo pesante.

Conviene allora sviluppare un metodo più efficiente, che inoltre ha il pregio di calcolare simultaneamente tutte le variabili vive in ogni punto del grafo. A tale scopo, si devono esaminare i cammini che dal punto considerato vanno a qualche istruzione che fa uso d'una variabile.

Per rendere più ordinato il calcolo, si scrivono certe equazioni dette di flusso

che, nodo per nodo, correlano le variabili vive all'uscita  $vive_{out}(p)$  e all'ingresso  $vive_{in}(p)$  del nodo medesimo e dei suoi successori.

Si indica con  $succ(p)$  l'insieme dei nodi successori (immediati) di  $p$  e con  $var(A)$  l'insieme delle variabili del programma  $A$ .

**Equazioni di flusso:**

per ogni nodo  $p$  finale:

$$vive_{out}(p) = \emptyset \quad (5.2)$$

per ogni altro nodo  $p$ :

$$vive_{in}(p) = usa(p) \cup (vive_{out}(p) \setminus def(p)) \quad (5.3)$$

$$vive_{out}(p) = \bigsqcup_{q \in succ(p)} vive_{in}(q) \quad (5.4)$$

**Commenti:**

- Nella eq. 5.2 nessuna variabile è viva all'uscita del grafo.<sup>29</sup>.
- Per la 5.3, una variabile è viva all'ingresso di  $p$  se essa è usata in  $p$ , o se essa è viva all'uscita di  $p$  ma non è definita in  $p$  stesso.

Si consideri l'istruzione  $4 : a := b \times 3$  del programma di p. 331. All'uscita di 4 sono vive  $a, m, c$  poiché esistono cammini che raggiungono gli usi di tali variabili senza incontrare delle definizioni delle stesse. All'entrata di 4 sono vive:  $b$  perché è usata in 4;  $c, m$  perché sono vive all'uscita e non definite in 4; invece  $a$ , pur essendo viva all'uscita di 4, non lo è all'ingresso di 4, poiché è definita in 4.

- Per l'equazione 5.4, il nodo 5 ha due successori 2 e 6; le variabili vive all'uscita di 5 sono quelle ( $a, c, m$ ) vive all'ingresso di 2 e quella ( $c$ ) viva all'ingresso di 6.

### Soluzione delle equazioni di flusso

Dato il grafo di controllo, è facile scrivere le equazioni di flusso, istruzione per istruzione. Per un grafo di  $|I| = n$  nodi si ottiene un sistema di  $2 \times n$  equazioni nelle  $2 \times n$  incognite  $vive_{in}(p), vive_{out}(p), p \in I$ . La soluzione del sistema è un vettore di  $2 \times n$  insiemi.

Il sistema si risolve iterativamente assegnando l'insieme vuoto come valore iniziale a ogni incognita:

$$\forall p : vive_{in}(p) = \emptyset; vive_{out}(p) = \emptyset$$

Questa è l'iterazione iniziale  $i = 0$ . Si sostituiscono nelle equazioni del sistema 5.2 i valori dell'iterazione corrente  $i$  e si ricavano i valori dell'iterazione  $i + 1$ . Se almeno uno di essi differisce dal valore dell'iterazione  $i$  si continua allo stesso modo, altrimenti si termina e i valori dell'iterazione  $i + 1$  costituiscono

<sup>29</sup>Si trascurano gli eventuali parametri d'uscita del sottoprogramma, che sono normalmente usati anche dopo il termine del sottoprogramma.

la soluzione del sistema.

Questa soluzione è dunque il *primo punto fisso* della trasformazione che produce un nuovo vettore partendo da quello dell'iterazione precedente.

Per dimostrare che il calcolo converge dopo un numero finito di iterazioni, si osservi che:

- ogni insieme  $vive_{in}(p)$  e  $vive_{out}(p)$  ha una cardinalità limitata superiormente dal numero di variabili del programma;
- l'iterazione non può togliere elementi da nessun insieme, ma soltanto aggiungerli o lasciarlo immutato;
- se l'iterazione lascia immutati tutti gli insiemi, l'algoritmo termina.

*Esempio 5.56.* Calcolo iterativo delle variabili vive.

Gli insiemi delle istruzioni che definiscono e usano le variabili sono di calcolo immediato:

|   | D    | U    |
|---|------|------|
| a | 1, 4 | 2, 5 |
| b | 2    | 3, 4 |
| c | 3    | 3, 6 |
| m | ∅    | 5    |

Le equazioni per il programma dell'es. 5.53 sono scritte abbreviando i nomi delle incognite:  $in(p)$  invece di  $vive_{in}(p)$  e  $out(p)$  invece di  $vive_{out}(p)$ :

$$\begin{array}{ll} 1 | in(1) = out(1) \setminus \{a\} & out(1) = in(2) \\ 2 | in(2) = \{a\} \cup (out(2) \setminus \{b\}) & out(2) = in(3) \\ 3 | in(3) = \{b, c\} \cup (out(3) \setminus \{c\}) & out(3) = in(4) \\ 4 | in(4) = \{b\} \cup (out(4) \setminus \{a\}) & out(4) = in(5) \\ 5 | in(5) = \{a, m\} \cup out(5) & out(5) = in(2) \cup in(6) \\ 6 | in(6) = \{c\} & out(6) = \emptyset \end{array}$$

Le approssimazioni, calcolate partendo dagli insiemi tutti vuoti, sono sotto tabulate, supponendo che ogni iterazione calcoli prima le  $in$  e poi le  $out$ :

|   | $in = out$ | $in$ | $out$ | $in$    | $out$   | $in$    | $out$   | $in$    | $out$   | $in$    | $out$   |
|---|------------|------|-------|---------|---------|---------|---------|---------|---------|---------|---------|
| 1 | ∅          | ∅    | a     | ∅       | a, c    | c       | a, c    | c       | a, c, m | c, m    | a, c, m |
| 2 | ∅          | a    | b, c  | a, c    | b, c    | a, c    | b, c, m | a, c, m | b, c, m | a, c, m | b, c, m |
| 3 | ∅          | b, c | b     | b, c    | b, m    | b, c, m | b, c, m | b, c, m | b, c, m | b, c, m | b, c, m |
| 4 | ∅          | b    | a, m  | b, m    | a, c, m | b, c, m | a, c, m | b, c, m | a, c, m | b, c, m | a, c, m |
| 5 | ∅          | a, m | a, c  | a, c, m | a, c    | a, c, m | a, c    | a, c, m | a, c, m | b, c, m | a, c, m |
| 6 | ∅          | c    | ∅     | c       | ∅       | c       | ∅       | c       | ∅       | c       | ∅       |

Il punto fisso si raggiunge dopo 5 iterazioni: infatti un'ulteriore iterazione non modificherebbe il risultato.

Si noti che la soluzione non dipende dall'ordine in cui si esaminano i nodi del grafo, ma la rapidità di convergenza dell'algoritmo dipende dall'ordine.

Si potrebbe derivare la complessità di calcolo dell'algoritmo nel caso peggiore, e si troverebbe  $(O)(n^4)$ , dove  $n$  è il numero di nodi del grafo. Tuttavia in pratica la complessità di calcolo è poco più che lineare.

## Applicazione del risultato

Si mostrano ora due tipiche applicazioni dell'analisi precedente: l'assegnazione della memoria alle variabili e la scoperta delle istruzioni inutili.

### Assegnazione della memoria

L'analisi della vita serve per decidere se due variabili possono stare nella stessa cella di memoria (o nello stesso registro della macchina). È evidente che se  $a$  e  $b$  sono vive in uno stesso punto, in quel punto entrambi i valori vanno conservati per degli usi futuri, e non possono stare nella stessa cella di memoria. Si dice allora che le due variabili *interferiscono*.

Se due variabili non interferiscono, ossia non sono mai vive contemporaneamente, ad essi si può assegnare lo stesso indirizzo nella memoria o lo stesso registro.

#### *Esempio 5.57. Interferenza e assegnazione dei registri.*

Nel grafo di controllo di p. 334 si vede che le coppie  $a, c$  e  $c, m$ , essendo presenti nell'insieme  $in(2)$  interferiscono. Anche le coppie  $b, c$  e  $b, m$  interferiscono nell'insieme  $in(3)$ . Al contrario nessun insieme contiene la coppia  $a, b$ . Tenuto conto del vincolo che due variabili interferenti devono stare in celle diverse, le variabili  $c$  e  $m$  devono stare ciascuna nella propria cella, mentre le variabili  $a$  e  $b$  possono stare nella stessa cella, diversa dalle due precedenti. In conclusione bastano tre celle per le quattro variabili del programma.

Nei moderni compilatori, l'assegnazione dei registri alle variabili si fa con metodi euristici che sfruttano la relazione di interferenza.

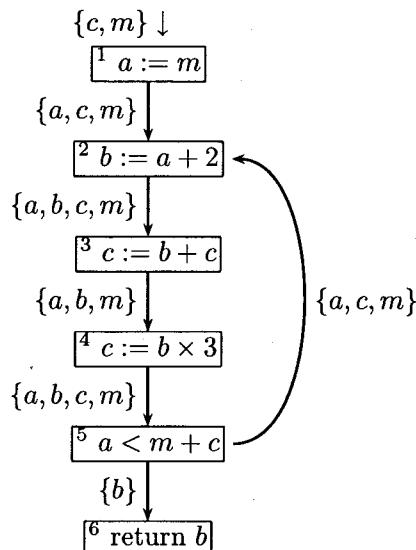
### Definizioni inutili

Un'istruzione che definisce una variabile è inutile, se la variabile non è viva all'uscita dell'istruzione. La ricerca delle definizioni inutili si riduce al controllo che ogni istruzione  $p$  che definisce una variabile  $a$  contenga la variabile nell'insieme  $out(p)$ .

Nel programma di p. 334 nessuna definizione di variabile è inutile, a differenza dal prossimo esempio.

#### *Esempio 5.58. Definizioni inutili.*

Si consideri il programma seguente.



Nella figura sono riportate le variabili vive all'entrata e all'uscita delle istruzioni. La variabile  $c$  all'uscita di 3 non è viva, e l'istruzione 3 è inutile. Eliminando l'istruzione 3, non solo si accorcia il programma, ma si causa la scomparsa di  $c$  dagli insiemi  $in(1), in(2), in(3), out(5)$ . Ciò diminuisce le interferenze tra le variabili e può ridurre il numero di registri necessari.

Questo esempio illustra le ottimizzazioni a catena che spesso si offrono quando si applica un miglioramento a un programma.

### 5.7.3 Definizioni raggiungenti

Si vedrà ora un'altra importante analisi statica, il calcolo delle definizioni che raggiungono i vari punti del programma.

Se un assegnamento che definisce la variabile  $a$  ha come parte destra una costante, il compilatore esamina il programma per vedere se la medesima costante può essere sostituita alla variabile nelle istruzioni che usano  $a$ . Il guadagno della sostituzione è molteplice: primo, un'operazione con un argomento costante è spesso più veloce; secondo, la sostituzione può rendere costante quell'espressione e consentire al compilatore di valutarla senza dover generare codice. Infine la sostituzione elimina uno o più usi di  $a$  e perciò accorcia gli intervalli di vita, riducendo così le interferenze con altre variabili e la pressione sull'uso della memoria o dei registri.

Questa ottimizzazione è detta *propagazione delle costanti*. Per svilupparla è prima necessario definire alcuni concetti, utili anche per altre ottimizzazioni e verifiche dei programmi.

Si consideri un'istruzione  $p : a := b \oplus c$  che definisce la variabile  $a$ . Per brevità essa sarà indicata come  $a_p$ , mentre si indica con  $D(a)$  l'insieme di tutte le definizioni di  $a$  nel programma.

**Definizione 5.59.** Si dice che la definizione di  $a$  in  $q$ ,  $a_q$ , raggiunge l'ingresso di un'istruzione  $p$  (non necessariamente distinta da  $q$ ) se esiste un cammino da  $q$  a  $p$  che non passa per un nodo, diverso da  $q$ , che definisce  $a$ .

In tale caso l'istruzione  $p$  potrà usare il valore di  $a$  definito in  $q$ .

Con riferimento all'automa  $A$  ossia al grafo di controllo del programma, la condizione si riformula così:

esiste nel linguaggio  $L(A)$  una frase  $x$  tale che

$$x = uqvpw$$

dove

$$u \in I^*, q \in D(a), v \in (I \setminus D(a))^*, p \in I, w \in I^*$$

Si noti che  $p$  e  $q$  possono coincidere.

Ad es. nel programma di p. 331 (riprodotto anche a p. 341), la definizione  $a_1$  raggiunge l'ingresso delle istruzioni 2,3,4 ma non l'ingresso di 5. La definizione  $a_4$  raggiunge l'ingresso di 5,6,2,3,4.

### Equazioni di flusso per le definizioni raggiungenti

Il calcolo delle definizioni raggiungenti i vari punti del programma sarà ora formulato come soluzione di certe equazioni di flusso.

Se il nodo  $p$  definisce la variabile  $a$ , si dice che ogni altra definizione  $a_q$ ,  $q \neq p$ , della stessa variabile è *soppressa* da  $p$ . L'insieme delle definizioni sopprese da  $p$  è:

$$\begin{cases} sop(p) = \{a_q \mid q \in I \wedge q \neq p \wedge a \in def(q) \wedge a \in def(p)\}, & \text{se } def(p) \neq \emptyset \\ sop(p) = \emptyset, & \text{se } def(p) = \emptyset \end{cases}$$

Si noti che l'insieme  $def(p)$  contiene più d'una variabile, nel caso di istruzioni come la lettura "read(a,b,c)".

Gli insiemi delle definizioni che raggiungono l'entrata e l'uscita del nodo  $p$  sono scritti  $in(p)$  e  $out(p)$ . I nodi predecessori (immediati) di  $p$  formano un insieme scritto come  $pred(p)$ .

Seguono le equazioni di flusso:

Per il nodo 1 iniziale:

$$in(1) = \emptyset \quad (5.5)$$

Per ogni altro nodo  $p \in I$ :

$$out(p) = def(p) \cup (in(p) \setminus sop(p)) \quad (5.6)$$

$$in(p) = \bigsqcup_{\forall q \in pred(p)} out(q) \quad (5.7)$$

Commenti:

La eq. 5.5 ipotizza per semplicità che non ci siano variabili passate come parametri d'entrata al sottoprogramma. Altrimenti  $in(1)$  contiene le definizioni, esterne al sottoprogramma, di tali parametri.

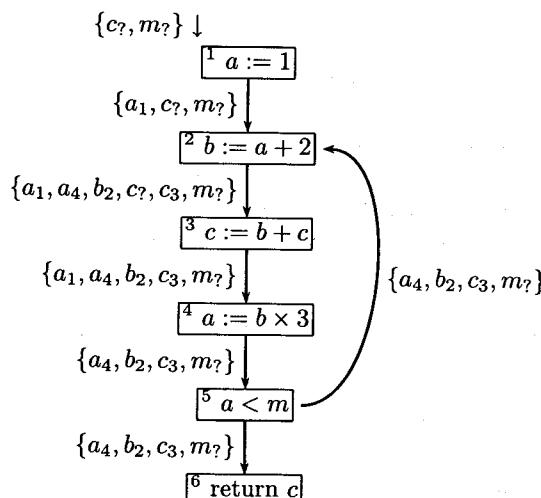
La eq. 5.6 pone nell'uscita di  $p$  le definizioni proprie di  $p$  e quelle raggiungenti l'ingresso di  $p$ , purché non sopprese da  $p$ .

La eq. 5.7 dice che confluiscono all'ingresso di  $p$  le definizioni raggiungenti le uscite dei nodi predecessori.

Il sistema di equazioni si risolve, come quello delle variabili vive, mediante successive iterazioni che convergono al primo punto fisso. Nell'iterazione iniziale tutti gli insiemi sono vuoti.

*Esempio 5.60.* Definizioni raggiungenti.

La prossima figura riproduce il grafo di controllo di p. 331 e riporta le definizioni raggiungenti i punti del programma, calcolate come spiegato nel seguito. Le variabili  $c$  e  $m$ , essendo parametri del sottoprogramma, sono definite all'esterno in un punto ignoto, indicato come  $c_?$  e  $m_?$ .



Ad es. si osservi che la definizione  $c_?$  di  $c$ , esterna, non raggiunge l'ingresso di 4, perché è soppressa da 3.

Prima di scrivere le equazioni di flusso, si elencano i termini costanti:

| nodo                | def         | sop         |
|---------------------|-------------|-------------|
| 1 $a := 1$          | $a_1$       | $a_4$       |
| 2 $b := a + 2$      | $b_2$       | $\emptyset$ |
| 3 $c := b + c$      | $c_3$       | $c_?$       |
| 4 $a := b \times 3$ | $a_4$       | $a_1$       |
| 5 $a < m$           | $\emptyset$ | $\emptyset$ |
| 6 return $c$        | $\emptyset$ | $\emptyset$ |

Seguono le equazioni di flusso:

$$\begin{aligned}
 in(1) &= \{c_?, m_?\} \\
 out(1) &= \{a_1\} \cup (in(1) \setminus \{a_4\}) \\
 in(2) &= out(1) \cup out(5) \\
 out(2) &= \{b_2\} \cup (in(2) \setminus \emptyset) = \{b_2\} \cup in(2) \\
 in(3) &= out(2) \\
 out(3) &= \{c_3\} \cup (in(3) \setminus \{c_?\}) \\
 in(4) &= out(3) \\
 out(4) &= \{a_4\} \cup (in(4) \setminus \{a_1\}) \\
 in(5) &= out(4) \\
 out(5) &= \emptyset \cup (in(5) \setminus \emptyset) = in(5) \\
 in(6) &= out(5) \\
 out(6) &= \emptyset \cup (in(6) \setminus \emptyset) = in(6)
 \end{aligned}$$

All'iterazione 0 tutti gli insiemi sono vuoti. Dopo poche iterazioni si ottiene la soluzione mostrata nella figura precedente.

### Propagazione delle costanti

Continuando l'es. precedente, si considera ora la possibilità di sostituire a una variabile un valore costante.

Un esempio è la domanda: è lecito sostituire alla variabile  $a$  nella 2 la costante 1, assegnata dall'istruzione 1 (definizione  $a_1$ )? La risposta è negativa perché si vede che l'insieme  $in(2)$  contiene anche un'altra definizione di  $a$ , la  $a_4$ , il che significa che in certi calcoli il programma potrebbe usare per  $a$  il valore definito in 4 e la sostituzione produrrebbe un programma non equivalente a quello dato.

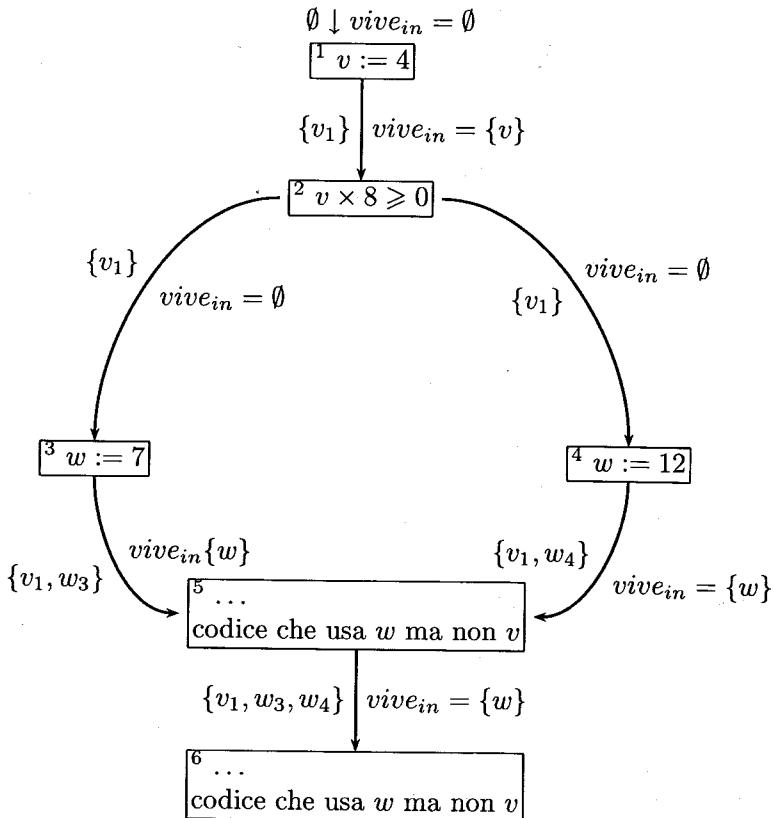
Da questo ragionamento è facile formulare una condizione generale. È lecito sostituire nella istruzione  $p$ , al posto della variabile  $a$  ivi usata, la costante  $k$  se

1. esiste un'istruzione  $q : a := k$ ,  $k$  costante, tale che la definizione  $a_q$  raggiunge l'ingresso di  $p$   $\wedge$
2. nessun'altra definizione  $a_r, r \neq q$  di  $a$  raggiunge l'ingresso di  $p$ .

Il prossimo programma mostra il miglioramento ottenuto con la propagazione delle costanti e altre semplificazioni indotte.

*Esempio 5.61.* Propagazione delle costanti.

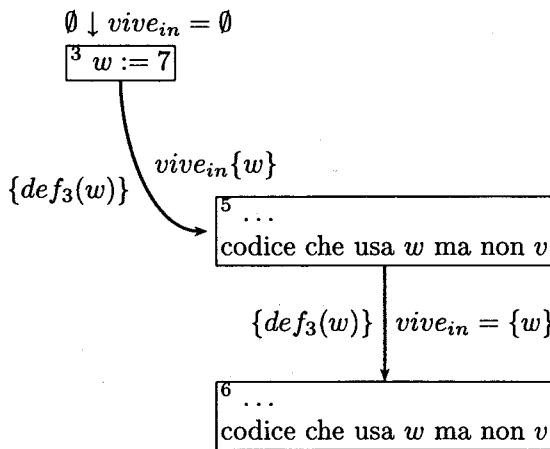
Nella prossima figura sono indicate le definizioni raggiungenti e le variabili vive nei punti del programma.



Poiché l'unica definizione di  $v$  raggiungente l'ingresso di 2 è la  $v_1$ , è permesso sostituire la costante 4 al posto di  $v$  nel test 2, che si trasforma nell'espressione costante  $4 \times 8 \geq 0$ .

Ora la variabile  $v$  cessa di essere viva all'uscita dell'assegnamento 1, che diviene inutile e può essere cancellato.

Ma le semplificazioni non terminano qui. La conoscenza del valore dell'espressione booleana costante  $32 \geq 0 = \text{true}$  consente al compilatore di determinare quale dei due rami del condizionale 2 vada eseguito, mettasi quello di sinistra, e di eliminare l'altro. Infatti l'istruzione 4 diventa allora un codice irraggiungibile dall'entrata del programma. Anche il test 2, ormai superfluo, è eliminato. Il programma si è così semplificato:



L'analisi potrebbe poi continuare esaminando la possibilità di propagare la costante  $w = 7$  nel resto del programma.

### Disponibilità delle variabili e inizializzazioni

Una verifica di correttezza richiesta al compilatore è il controllo delle inizializzazioni: ogni variabile usata in un'istruzione deve avere un valore al momento dell'esecuzione, ottenuto tramite un assegnamento valido (o comunque una definizione). In caso contrario la variabile è detta *indisponibile* e si ha un errore.

Riprendendo il grafo di controllo dell'es. a p. 341, si vede che il nodo 3 usa la variabile  $c$  ma nel cammino 123 nessuna istruzione eseguita prima di 3 assegna un valore a  $c$ . Questo non è necessariamente un errore:  $c$  potrebbe essere un parametro d'ingresso del sottoprogramma e ricevere il valore al momento della chiamata. Se così fosse, la variabile  $c$  avrebbe un valore all'ingresso del nodo 3. Lo stesso vale per la variabile  $m$ .

La variabile  $b$  è disponibile all'ingresso di 3 perché ha ricevuto un valore in 2 per mezzo d'un assegnamento che usa la variabile  $a$ , a sua volta disponibile all'ingresso di 2, essendo stata inizializzata in 1.

Per procedere, è necessario precisare il concetto di disponibilità. Per semplicità si suppone che il programma non abbia parametri d'entrata.

**Definizione 5.62.** Una variabile  $a$  è disponibile all'ingresso del nodo  $p$ , cioè subito prima della sua esecuzione, se nel grafo di controllo ogni cammino dal nodo iniziale 1 all'ingresso di  $p$ , contiene una definizione di  $a$ .

Confrontando questa condizione con quella di definizione raggiungente (p. 340), si nota una differenza nella quantificazione dei cammini. Se una definizione  $a_q$  di  $a$  raggiunge l'ingresso di  $p$ , certamente esiste un cammino da 1 a  $p$

passante per il punto di definizione  $q$ . Ma non si può escludere che esista anche un altro cammino da  $1$  a  $p$ , non passante per  $q$  né per un'altra definizione di  $a$ .

Pertanto il concetto di disponibilità è più restrittivo di quello di definizione raggiungente.

Per calcolare quali variabili siano disponibili all'ingresso del nodo  $p$ , si devono esaminare più da vicino le definizioni raggiungenti le uscite dei predecessori di  $p$ . Se per ogni nodo  $q$ , predecessore di  $p$ , l'insieme  $out(q)$  delle definizioni raggiungenti l'uscita di  $q$  contiene una definizione della variabile  $a$ , essa risulta disponibile all'ingresso di  $p$ . Si dice allora che qualche definizione di  $a$  *raggiunge sempre il nodo  $p$* .

Ora si può rendere operativo il test sull'inizializzazione delle variabili. Per un'istruzione  $q$  si indica con  $out'(q)$  l'insieme delle definizioni raggiungenti l'uscita di  $q$ , private dei pedici. Ad es. se è  $out(q) = \{a_1, a_4, b_3, c_6\}$ , si ha  $out'(q) = \{a, b, c\}$ .

#### *Non buona inizializzazione*

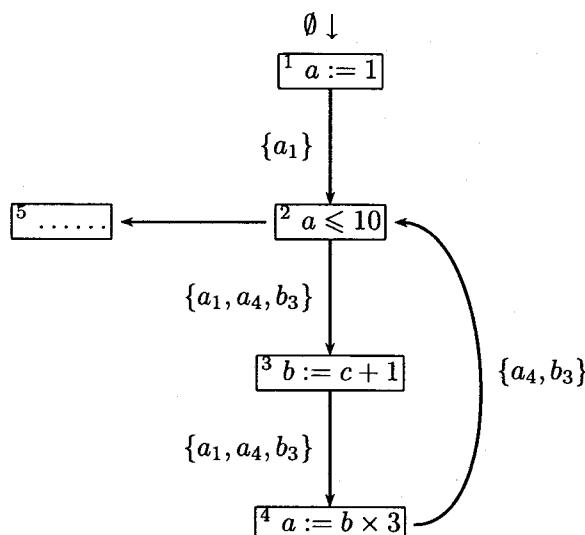
Un'istruzione  $p$  non è ben inizializzata se vale il predicato:

$$\exists q \in pred(p) \text{ tale che } usa(p) \not\subseteq out'(q) \quad (5.8)$$

La condizione afferma l'esistenza d'un predecessore  $q$  di  $p$ , le cui definizioni raggiungenti non comprendono tutte le variabili usate in  $p$ . Di conseguenza, quando il calcolo segue il cammino che passa per  $q$ , una o più variabili usate in  $p$  saranno prive di valore.

*Esempio 5.63.* Scoperta di variabili non inizializzate.

Si mostra un programma con gli insiemi delle variabili disponibili.



La condizione 5.8 di non buona inizializzazione è falsa nel nodo 2, poiché ogni suo predecessore (1 e 4) contiene nel proprio insieme *out'* una definizione di *a*, la sola variabile usata in 2. Invece la condizione è vera in 3, perché non vi sono definizioni di *c* raggiungenti l'uscita di 2.

L'analisi ha così trovato un errore nel programma: un'istruzione fa uso d'una variabile non inizializzata.

Volendo continuare la ricerca di altri eventuali errori di inizializzazione, si procede nel seguente modo. Si cancella l'istruzione erronea 3 trovata, si aggiorna il calcolo degli insiemi delle definizioni raggiungenti e si rivaluta la condizione 5.8. Così facendo si troverebbe che l'istruzione 4 non è ben inizializzata, perché la definizione *b*<sub>3</sub> di *out*(3) non è in realtà disponibile, a causa dell'errore trovato in 3. Cancellando la 3 e proseguendo nello stesso modo, non si trovano altri errori.

L'analisi precedente ha permesso di scoprire degli errori, che le fasi di parsificazione e di valutazione semantica non avevano potuto diagnosticare. Si evita così di lanciare un programma, che, cadendo in errore durante l'esecuzione, potrebbe causare imprevedibili conseguenze.

In conclusione, l'analisi statica dei programmi non si limita allo studio delle proprietà di vita e di raggiungimento sopra considerate, ma è un modo generale di analizzare tante altre proprietà dei programmi, sia per ottimizzarli, sia per verificarne la correttezza.<sup>30</sup>

---

<sup>30</sup>Un testo dedicato alla teoria dell'analisi statica è [38]. Per le applicazioni alla compilazione si veda [1] o [4].