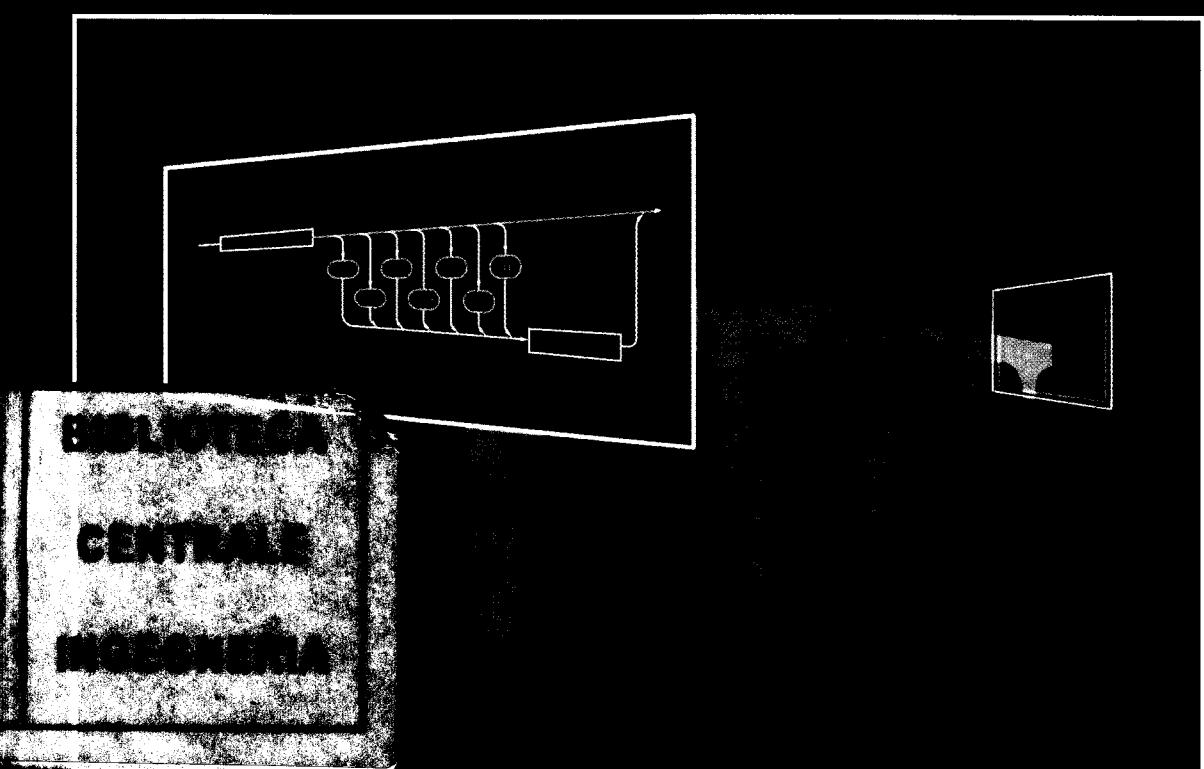


Stefano Crespi Reghizzi

# LINGUAGGI FORMALI E COMPILAZIONE

 Pitagora Editrice Bologna



---

# Indice

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Introduzione</b>   | <b>1</b> |
| 1.1      | Destinazione dell'opera                                     | 1        |
| 1.2      | Parti del compilatore e concetti corrispondenti             | 2        |
| <b>2</b> | <b>Sintassi</b>   | <b>5</b> |
| 2.1      | Introduzione  | 5        |
| 2.1.1    | Linguaggi artificiali e formali, sintassi e semantica       | 5        |
| 2.1.2    | Tipi di linguaggi   | 6        |
| 2.1.3    | Scaletta  | 7        |
| 2.2      | Teoria dei linguaggi formali                                | 8        |
| 2.2.1    | Alfabeto e linguaggio                                       | 8        |
| 2.2.2    | Operazioni sui linguaggi                                    | 12       |
| 2.2.3    | Operazioni insiemistiche                                    | 13       |
| 2.2.4    | Stella e croce  | 14       |
| 2.2.5    | Quoziente   | 17       |
| 2.3      | Espressioni e linguaggi regolari                            | 17       |
| 2.3.1    | Definizione di espressione regolare                         | 18       |
| 2.3.2    | Derivazione e linguaggio                                    | 20       |
| 2.3.3    | Altri operatori   | 23       |
| 2.3.4    | Chiusura della famiglia <i>REG</i> rispetto alle operazioni | 24       |
| 2.4      | Astrazione linguistica                                      | 25       |
| 2.4.1    | Liste astratte e concrete                                   | 26       |
| 2.5      | Grammatiche generative libere dal contesto                  | 30       |
| 2.5.1    | Limiti dei linguaggi regolari                               | 30       |
| 2.5.2    | Introduzione alle grammatiche libere                        | 30       |
| 2.5.3    | Rappresentazioni convenzionali delle grammatiche            | 33       |
| 2.5.4    | Derivazioni e linguaggio generato                           | 35       |
| 2.5.5    | Grammatiche erronee e regole inutili                        | 37       |
| 2.5.6    | Ricorsione delle regole e infinitezza del linguaggio        | 39       |
| 2.5.7    | Alberi sintattici e derivazioni canoniche                   | 40       |
| 2.5.8    | Linguaggi a parentesi                                       | 43       |

|          |   |           |
|----------|---|-----------|
| 2.5.9    | Composizione regolare di linguaggi liberi . . . . .                           | 45        |
| 2.5.10   | Ambiguità . . . . .   | 47        |
| 2.5.11   | Catalogo di forme ambigue e rimedi . . . . .                                  | 49        |
| 2.5.12   | Equivalenza debole e strutturale . . . . .                                    | 57        |
| 2.5.13   | Trasformazioni delle grammatiche e forme normali . . . . .                    | 60        |
| 2.6      | Le grammatiche dei linguaggi regolari . . . . .                               | 67        |
| 2.6.1    | Dalla espressione regolare alla grammatica libera . . . . .                   | 68        |
| 2.6.2    | Grammatiche lineari . . . . .   | 69        |
| 2.6.3    | Equazioni lineari del linguaggio . . . . .                                    | 71        |
| 2.7      | Linguaggi regolari e liberi a confronto . . . . .                             | 73        |
| 2.7.1    | Limiti dei linguaggi liberi dal contesto . . . . .                            | 76        |
| 2.7.2    | Proprietà di chiusura di REG e LIB . . . . .                                  | 78        |
| 2.7.3    | Trasformazioni alfabetiche . . . . .  | 80        |
| 2.7.4    | Grammatiche libere estese con espressioni regolari . . . . .                  | 83        |
| 2.8      | Grammatiche e famiglie di linguaggi più generali . . . . .                    | 86        |
| 2.8.1    | Classificazione di Chomsky . . . . .  | 87        |
| <b>3</b> | <b>Automi finiti e riconoscimento dei linguaggi regolari . . . . .</b>        | <b>93</b> |
| 3.1      | Introduzione . . . . .  | 93        |
| 3.2      | Algoritmi di riconoscimento e automi . . . . .                                | 94        |
| 3.2.1    | Automa generale . . . . .   | 95        |
| 3.3      | Introduzione agli automi finiti . . . . .                                     | 98        |
| 3.4      | Automa finito deterministico . . . . .  | 100       |
| 3.4.1    | Stato di errore e completamento dell'automa . . . . .                         | 101       |
| 3.4.2    | Automa pulito . . . . .   | 102       |
| 3.4.3    | Automa minimo . . . . .   | 103       |
| 3.4.4    | Dall'automa alla grammatica . . . . .   | 106       |
| 3.5      | Automi indeterministici . . . . .   | 107       |
| 3.5.1    | Motivazioni dell'indeterminismo . . . . .                                     | 108       |
| 3.5.2    | Riconoscimento indeterministico . . . . .                                     | 110       |
| 3.5.3    | Automi con mosse spontanee . . . . .  | 112       |
| 3.5.4    | Corrispondenza tra automa e grammatica . . . . .                              | 113       |
| 3.5.5    | Ambiguità dell'automa . . . . .   | 115       |
| 3.5.6    | Grammatica lineare a sinistra e automa . . . . .                              | 115       |
| 3.6      | Dall'automa all'espressione regolare direttamente: il metodo<br>BMC . . . . . | 116       |
| 3.7      | Eliminazione dell'indeterminismo . . . . .                                    | 118       |
| 3.7.1    | Costruzione delle parti finite raggiungibili . . . . .                        | 120       |
| 3.8      | Dall'espressione regolare all'automa riconoscitore . . . . .                  | 123       |
| 3.8.1    | Metodo strutturale o di Thompson . . . . .                                    | 123       |
| 3.8.2    | Algoritmo di Glushkov, Mc Naughton e Yamada . . . . .                         | 126       |
| 3.8.3    | Costruzione del riconoscitore deterministico di Berry<br>e Sethi . . . . .    | 133       |
| 3.9      | Espressioni regolari con complemento e intersezione . . . . .                 | 136       |
| 3.9.1    | Prodotto di automi . . . . .  | 138       |

|  |     |
|--|-----|
| 3.10 Riepilogo: relazioni tra linguaggi regolari, automi e grammatiche | 142 |
| <b>4 Riconoscimento e parsificazione delle frasi</b>                   | 145 |
| 4.1 Introduzione   | 145 |
| 4.1.1 Automi a pila  | 146 |
| 4.1.2 Dalla grammatica all'automa a pila                               | 147 |
| 4.1.3 Definizione dell'automa a pila                                   | 150 |
| 4.2 Linguaggi liberi e automi a pila: una sola famiglia                | 155 |
| 4.2.1 Intersezione di linguaggi liberi e regolari                      | 158 |
| 4.2.2 Automi a pila e linguaggi deterministici                         | 160 |
| 4.3 Analisi sintattica   | 168 |
| 4.3.1 Analisi discendente e ascendente                                 | 168 |
| 4.3.2 La grammatica come rete di automi finiti                         | 170 |
| 4.3.3 Procedura ricorsiva indeterministica di riconoscimento           | 174 |
| 4.4 Analisi sintattica discendente deterministica                      | 176 |
| 4.4.1 Condizioni per la costruzione del riconoscitore $LL(1)$          | 177 |
| 4.4.2 Come ottenere grammatiche $LL(1)$                                | 190 |
| 4.4.3 Allungamento della prospezione                                   | 194 |
| 4.5 Analisi sintattica ascendente deterministica                       | 199 |
| 4.5.1 Analisi $LR(0)$  | 199 |
| 4.5.2 Grammatiche $LR(0)$  | 201 |
| 4.5.3 Analizzatore a spostamento e riduzione                           | 206 |
| 4.5.4 Analisi sintattica con prospezione $LR(k)$                       | 209 |
| 4.5.5 Algoritmo di parsificazione $LR(1)$                              | 222 |
| 4.5.6 Proprietà delle sottofamiglie deterministiche e confronti        | 223 |
| 4.5.7 Come ottenere grammatiche $LR(1)$                                | 226 |
| 4.5.8 Analisi sintattica $LR(1)$ con grammatiche estese                | 229 |
| 4.6 Un algoritmo generale di analisi sintattica                        | 239 |
| 4.7 Scelta del parsificatore   | 253 |
| <b>5 Traduzione semantica e analisi statica</b>                        | 257 |
| 5.1 Introduzione   | 257 |
| 5.1.1 Contenuti  | 258 |
| 5.2 Relazione e funzione di traduzione                                 | 260 |
| 5.3 Traslitterazioni   | 262 |
| 5.4 Traduzioni regolari  | 263 |
| 5.4.1 Automa riconoscitore a due ingressi                              | 264 |
| 5.4.2 Funzione di traduzione e automa traduttore                       | 268 |
| 5.5 Traduzioni sintattiche pure  | 273 |
| 5.5.1 Scritture infisse e polacche                                     | 275 |
| 5.5.2 Ambiguità della grammatica sorgente e della traduzione           | 278 |
| 5.5.3 Grammatica di traduzione e traduttore a pila                     | 280 |
| 5.5.4 Analisi sintattica e traduzione in linea                         | 284 |
| 5.5.5 Traduzioni deterministiche discendenti                           | 284 |
| 5.5.6 Traduzioni deterministiche ascendenti                            | 287 |

## VIII Indice

|   |            |
|---|------------|
| 5.5.7 Proprietà di chiusura rispetto alle traduzioni .....    | 293        |
| 5.6 Traduzioni semantiche .....                               | 295        |
| 5.6.1 Grammatiche con attributi .....                         | 296        |
| 5.6.2 Attributi sinistri e destri .....                       | 299        |
| 5.6.3 Definizione di grammatica con attributi .....           | 302        |
| 5.6.4 Grafo delle dipendenze e valutazione degli attributi .. | 304        |
| 5.6.5 Valutazione semantica con una scansione .....           | 308        |
| 5.6.6 Altri metodi di valutazione .....                       | 311        |
| 5.6.7 Analisi sintattica e semantica integrate .....          | 312        |
| 5.6.8 Applicazioni tipiche delle grammatiche con attributi .. | 320        |
| 5.6.9 Generazione del codice .....                            | 324        |
| 5.6.10 Analisi sintattica guidata dalla semantica.....        | 326        |
| 5.7 Analisi statica dei programmi .....                       | 330        |
| 5.7.1 Il programma come automa.....                           | 330        |
| 5.7.2 Intervalli di vita delle variabili .....                | 334        |
| 5.7.3 Definizioni raggiungenti .....                          | 339        |
| <b>Riferimenti bibliografici.....</b>                         | <b>347</b> |
| <b>Indice analitico .....</b>                                 | <b>351</b> |

## Sintassi

### 2.1 Introduzione

#### 2.1.1 Linguaggi artificiali e formali, sintassi e semantica

Molti secoli dopo l'emergenza spontanea del linguaggio nella comunicazione naturale, l'uomo ha consapevolmente progettato altri sistemi di comunicazione, i linguaggi artificiali, rivolti a obiettivi molto specifici. Anche se non mancano gli esempi antichi come le proposizioni della logica aristotelica e la notazione musicale di Guittone d'Arezzo, il numero dei linguaggi artificiali è cresciuto enormemente con l'avvento dell'informatica, a partire dagli anni 1950. Molti di essi sono rivolti alla comunicazione tra l'uomo e la macchina, che deve essere istruita a compiere un lavoro: l'esecuzione di un calcolo, la scrittura d'un documento, la ricerca di un'informazione in una base-dati, o la manovra d'un robot.

Altri linguaggi servono alla comunicazione tra apparati, come il linguaggio PostScript, scritto da un elaboratore per comandare le mosse d'una stampante.

Tutti i linguaggi inventati possono essere detti artificiali, ma non tutti sono formalizzati: i linguaggi di programmazione, come Java, sono linguaggi formalizzati, ma l'Esperanto, altrettanto artificiale essendo stato progettato da un uomo, certamente non lo è.

Si dirà formale un linguaggio se la forma delle frasi (o sintassi) e il loro significato (o semantica) sono definiti in modo preciso e algoritmico; ovvero se risulta possibile progettare una procedura informatica che verifichi la correttezza grammaticale delle frasi e ne calcoli il significato.

Evitando di addentrarsi nell'ardua definizione di che cosa sia il significato, ci si limita a dire che esso è la traduzione del linguaggio in un altro linguaggio formale, che risulta noto alla macchina o all'operatore. Ad es. il significato d'un programma Java è la sua traduzione nel linguaggio o codice macchina, composto di istruzioni che il microprocessore sa eseguire.

Escludendo la semantica, vi è un altro senso assai più limitato in cui si usa

il termine di linguaggio formale nell'ambito della sintassi, ed è questo che si adotterà. Un linguaggio formale è una struttura matematica, analoga a quelle studiate dalla teoria dei numeri, costruita a partire da un alfabeto, per mezzo di certe regole assiomatiche (grammatiche formali) o di certe macchine matematiche o automi (come quelle famose di A. Turing).

La teoria dei linguaggi formali si occupa della forma o sintassi delle frasi, ma non del loro significato. Una stringa o testo è semplicemente classificato come valido o non valido, ossia come appartenente o estraneo al linguaggio formale. Tale teoria non è però fine a se stessa, ma serve come impalcatura su cui la semantica del linguaggio potrà essere costruita con altri metodi.

### 2.1.2 Tipi di linguaggi

Un linguaggio è in questo libro un mezzo di comunicazione unidimensionale, formato da sequenze di elementi simbolici, i caratteri d'un alfabeto.

Tuttavia nel parlare comune il termine di linguaggio è anche usato in senso lato per designare non solo i linguaggi testuali, ma quasi ogni forma di comunicazione, più o meno formalizzata tramite un insieme di regole.

I linguaggi iconici trattano i segnali stradali o le icone del video del PC.

Il linguaggio musicale descrive la successione delle note, il ritmo e l'armonia.

L'architettura e il design si interessano alle forme dei manufatti e chiamano linguaggio lo studio delle relazioni tra di esse.

I disegni d'un bimbo sono le frasi d'un linguaggio comunicativo pittorico che è in piccola parte formalizzabile sulla base dei modelli psicologici del suo sviluppo mentale.

L'accostamento formale allo studio della sintassi dei linguaggi testuali, affrontato in questo capitolo, potrebbe avere interesse anche per i linguaggi intesi in senso lato.

Nell'informatica il termine di linguaggio si applica non solo ai testi, che sono insiemi di caratteri totalmente ordinati da sinistra a destra, ma anche a più complesse strutture discrete di varia costituzione: alberi, grafi, immagini discretizzate come tabelle bidimensionali di pixel, ecc.. Anche per tali linguaggi non testuali esistono teorie formali simili a quella esposta nel libro, ma meno consolidate e diffuse.<sup>1</sup>

Tornando al caso principale dei testi, nell'informatica è frequentemente necessario definire o specificare un linguaggio artificiale, allo scopo di documentare chi dovrà servirsene (manuale del linguaggio), di costituire un riferimento ufficiale (documenti standard), e di indicare ai progettisti del traduttore, compilatore o interprete di quel linguaggio, quali siano le frasi da accettare e con quale significato.

Non è facile definire un linguaggio in modo completo e rigoroso: infatti è impossibile elencare tutte le frasi valide, perché esse sono in numero infinito

---

<sup>1</sup>Si menzionano due esempi di tali teorie: la teoria dei linguaggi di alberi [20] e la teoria dei linguaggi bidimensionali [22, 15].

essendo a priori illimitata la loro lunghezza. Infatti il parlatore d'una lingua, così come il progettista d'un programma, non è mai limitato nella lunghezza delle frasi che può produrre. Il problema di rappresentare un numero infinito di casi mediante una descrizione finita si risolve ricorrendo a un algoritmo di enumerazione. Un algoritmo è, come si sa, un insieme finito di regole di calcolo; l'esecuzione dell'algoritmo produce l'enumerazione delle frasi del linguaggio, senza fine se il linguaggio ne contiene un numero illimitato.

Nell'ambito del libro si studiano certi modi semplici e pratici di scrivere le regole dell'algoritmo di enumerazione, sotto forma d'una grammatica (o sintassi) generativa. Lo studio delle grammatiche è l'argomento di questo capitolo.

### 2.1.3 Scaletta

L'esposizione inizia con i concetti elementari della teoria: alfabeto, stringa, operazioni insiemistiche, di concatenamento e di ripetizione, applicate alle stringhe e agli insiemi di stringhe.

Il capitolo prosegue con la definizione della famiglia dei linguaggi regolari e lo studio delle sue proprietà algebriche.

Segue lo studio delle liste, una struttura sintattica presente in ogni genere di linguaggio. Lo studio delle varianti delle liste introduce le astrazioni linguistiche, un potente mezzo di ragionamento per ricondurre le forme dei linguaggi a un numero ridotto di stereotipi.

Considerate le limitazioni dei linguaggi regolari, il capitolo continua con le grammatiche libere dal contesto, di cui si studiano le definizioni e le proprietà, in particolare quelle strutturali come la ricorsione e l'ambiguità.

I più importanti esempi di stereotipi linguistici sono esemplificati nel corso del capitolo: liste anche gerarchiche, strutture a parentesi, notazione polacca e linguaggi a operatori con precedenze. La combinazione d'un ridotto numero di stereotipi astratti produce la varietà delle forme presenti nei linguaggi artificiali.

Lo studio delle forme più comuni di ambiguità e dei rimedi fornisce indicazioni pratiche per il progetto delle grammatiche.

Varie trasformazioni delle grammatiche (forme normali) sono poi introdotte, per acquisire la capacità di trasformare le regole, senza alterare il linguaggio definito, allo scopo di adattarle alle esigenze applicative.

Il capitolo riprende poi lo studio dei linguaggi regolari in un'ottica grammaticale, che rende evidenti le ragioni della maggiore capacità descrittiva delle grammatiche rispetto alle espressioni regolari.

Il confronto tra le famiglie dei linguaggi regolari e liberi prosegue con lo studio delle operazioni che preservano l'appartenenza a una e all'altra famiglia. Tra queste vi sono le trasformazioni alfabetiche, che saranno riprese nel capitolo 5. Sono anche enunciate le proprietà che caratterizzano le ripetizioni inevitabili di certe sottostringhe nelle frasi più lunghe del linguaggio.

Il capitolo termina con la classificazione di Chomsky dei tipi di grammatiche

e un cenno alle grammatiche dipendenti dal contesto, più potenti ma poco comprensibili e raramente impiegate.

## 2.2 Teoria dei linguaggi formali

Si presentano ora gli elementi fondamentali della teoria dei linguaggi formali. Partendo dall'alfabeto, si introducono le operazioni che costruiscono e manipolano le stringhe di caratteri e poi i linguaggi più complessi, partendo da quelli elementari.

### 2.2.1 Alfabeto e linguaggio

Un *alfabeto* è un insieme finito di elementi, detti *simboli* o *caratteri terminali*. Sia  $\Sigma = \{a_1, a_2, \dots, a_k\}$  un alfabeto di  $k$  simboli; ovvero la sua *cardinalità*  $|\Sigma|$  vale  $k$ . Una *stringa* (o anche *parola*) è una sequenza (ossia un insieme ordinato con eventuali ripetizioni) di caratteri.

*Esempio 2.1.* Sia  $\Sigma = \{a, b\}$  l'alfabeto terminale. Allora  $aaba, aaa, abaa, b$  sono stringhe.

Un *linguaggio* è un insieme di stringhe di un dato alfabeto.

*Esempio 2.2.* Sia ancora  $\Sigma = \{a, b\}$  l'alfabeto. Ecco tre linguaggi aventi lo stesso alfabeto:

$$L_1 = \{aa, aaa\}$$

$$L_2 = \{aba, aab\}$$

$$L_3 = \{ab, ba, aabb, abab, \dots, aaabbb, \dots\} = \text{insieme delle stringhe contenenti eguale numero di } a \text{ e di } b$$

Si osserva che la struttura insiemistica d'un linguaggio formale ha due livelli: al primo vi è un insieme non ordinato di elementi non atomici (stringhe), al secondo vi sono degli insiemi ordinati di elementi atomici (caratteri terminali).

Una stringa appartenente a un linguaggio ne è una *frase*. Così  $bbaa \in L_3$  è una frase di  $L_3$ , mentre  $abb \notin L_3$  non è una frase ma una *stringa scorretta*.

La *cardinalità* d'un linguaggio è il numero delle sue stringhe: ad es.  $|L_2| = |\{aba, aab\}| = 2$ .

Un linguaggio avente cardinalità finita è detto *finito*. In caso contrario, ossia se non esiste un limite finito al numero delle frasi del linguaggio, esso è detto *infinito*.

Così  $L_1$  e  $L_2$  sono finiti, mentre  $L_3$  è infinito.

Un linguaggio finito, talvolta chiamato *vocabolario*, non è altro che un elenco

finito di parole<sup>2</sup>. Un caso particolare di linguaggio finito è l'insieme o *linguaggio vuoto*  $\emptyset$ , che non contiene alcuna frase,  $|\emptyset| = 0$ .

Per semplicità di scrittura, se un linguaggio contiene una sola frase, si possono omettere le parentesi graffe, scrivendo ad es.  $abb$  invece di  $\{abb\}$ .

Il *numero di caratteri*  $b$  presenti in una stringa  $x$  viene indicato da  $|x|_b$ . Così risulta:

$$|aab|_a = 2, \quad |baa|_a = 2, \quad |baa|_c = 0$$

La *lunghezza*  $|x|$  d'una stringa  $x$  è il numero dei suoi caratteri, ad es.:  $|ab| = 2$ ;  $|abaa| = 4$

Due stringhe

$$x = a_1 a_2 \dots a_h, \quad y = b_1 b_2 \dots b_k$$

si dicono *eguali* se  $h = k$  e  $a_i = b_i$ , per  $i = 1, \dots, h$ ; cioè se i loro caratteri letti ordinatamente da sinistra a destra coincidono.

Ad es. si ha:

$$aba \neq baa, \quad baa \neq ba$$

### Operazioni sulle stringhe

Si definiscono ora varie operazioni e relazioni che permettono di manipolare le stringhe in modo espressivo. Date le stringhe

$$x = a_1 a_2 \dots a_h, \quad y = b_1 b_2 \dots b_k$$

il *concatenamento*<sup>3</sup> è definito da

$$x.y = a_1 a_2 \dots a_h b_1 b_2 \dots b_k$$

Spesso il punto viene omesso scrivendo  $xy$  invece di  $x.y$ . Questo è l'operatore fondamentale della teoria dei linguaggi, come la somma per la teoria dei numeri.

*Esempio 2.3.* Per le stringhe

$$x = \text{vice}, \quad y = \text{capo}, \quad z = \text{stazione}$$

si ha

$$xy = \text{vice} \text{capo}, \quad yx = \text{capovice} \neq xy$$

$$(xy)z = \text{vice} \text{capo} \text{stazione} = x(yz) = \text{vice} \text{capostazione} = \text{vice} \text{capostazione}$$

---

<sup>2</sup>Nei trattati matematici il termine parola è sinonimo di stringa.

<sup>3</sup>Anche chiamato *prodotto* nei trattati più matematici.

Evidentemente il concatenamento *non è commutativo*; ossia in generale è

$$xy \neq yx$$

Il concatenamento è *associativo*, cioè vale l'identità:

$$(xy)z = x(yz)$$

Questa proprietà permette di scrivere senza parentesi il concatenamento  $xyz$  di tre (o più) stringhe.

Inoltre la lunghezza del concatenamento è la somma della lunghezza delle stringhe componenti

$$|xy| = |x| + |y| \quad (2.1)$$

### Stringa vuota

Si introduce una particolare stringa, la stringa *vuota* (o nulla), indicata dalla lettera greca  $\epsilon$  epsilon, che per definizione soddisfa l'identità

$$x\epsilon = \epsilon x = x$$

per ogni stringa  $x$ . Dalla eguaglianza 2.1 segue che la stringa vuota ha lunghezza nulla,

$$|\epsilon| = 0$$

In termini algebrici, la stringa vuota è l'elemento neutro rispetto al concatenamento, poiché qualsiasi stringa, concatenata a destra o a sinistra con  $\epsilon$ , non muta.

Si badi a non confondere i concetti di stringa e di insieme vuoto: infatti  $\emptyset$  è un linguaggio che non contiene alcuna stringa, quindi è diverso dall'insieme  $\{\epsilon\}$ , che contiene una frase, la stringa vuota.

### Sottostringhe

Sia  $x = u y v$  il concatenamento di certe stringhe  $u, y, v$ , eventualmente vuote. Allora  $y$  è una *sottostringa* di  $x$ ; inoltre  $u$  è un *prefisso* di  $x$ , e  $v$  è un *suffisso* di  $x$ .

Una sottostringa (prefisso, suffisso) si dice *propria* se non coincide con la stringa  $x$ .

Sia  $x$  una stringa di lunghezza almeno  $k$ ,  $|x| \geq k \geq 1$ . Allora con  $Ini_k(x)$  si indica il prefisso  $u$  di  $x$  avente lunghezza  $k$ , detto anche l'*inizio* di lunghezza  $k$ .

*Esempio 2.4.* La stringa  $x = aabacba$  contiene le seguenti componenti:

prefissi:  $a, aa, aab, aaba, aabac, aabacb, aabacba$

suffissi:  $a, ba, cba, acba, bacba, abacba, aabacba$

sottostringhe: i prefissi, i suffissi e le stringhe interne  
come  $a, ab, ba, bacb, \dots$

Si noti che  $bc$  non è una sottostringa di  $x$ , pur comparendo sia  $b$  che  $c$  in  $x$ .

Risulta poi  $Ini_2(aabacba) = aa$

### Riflessione

I caratteri d'una stringa sono di solito letti da sinistra a destra, ma talvolta serve leggerli nel verso opposto. La *riflessione* d'una stringa  $x = a_1a_2\dots a_h$  è la stringa  $x^R = a_ha_{h-1}\dots a_1$  ottenuta leggendo  $x$  da destra a sinistra. Ad es. si ha:

$$x = \text{roma}, \quad x^R = \text{amor}$$

Si verificano facilmente le identità:

$$(x^R)^R = x \quad (xy)^R = y^Rx^R \quad \varepsilon^R = \varepsilon$$

### Ripetizioni

Allo scopo di descrivere concisamente stringhe contenenti parti ripetute, si introduce la seguente operazione.

La *potenza  $m$ -esima* ( $m \geq 1$ , intero) d'una stringa  $x$  è il concatenamento di  $x$  con se stessa  $m - 1$  volte:

$$x^m = \underbrace{xx\dots x}_{m \text{ volte}}$$

Si conviene poi che la potenza 0 di qualsiasi stringa è la stringa vuota.  
La definizione complessiva è la seguente:

$$\begin{cases} x^m = x^{m-1}x, & m > 0 \\ x^0 = \varepsilon & \end{cases}$$

Esempi:

$$\begin{array}{llll} x = ab & x^0 = \varepsilon & x^1 = x = ab & x^2 = (ab)^2 = abab \\ y = a^2 = aa & y^3 = a^2a^2a^2 = a^6 & & \\ \varepsilon^0 = \varepsilon & \varepsilon^2 = \varepsilon & & \end{array}$$

Nella scrittura delle formule è necessario racchiudere tra parentesi la stringa da elevare alla potenza, se essa è più lunga d'un solo carattere. Volendo la seconda potenza di  $ab$  cioè  $abab$ , si deve scrivere  $(ab)^2$  e non  $ab^2$ , il quale vale  $abb$ .

In altre parole, vi è una regola convenzionale di precedenza tra gli operatori che dà *precedenza* all'elevamento a potenza rispetto al concatenamento.

Anche la riflessione, ha precedenza sul concatenamento: ad es.  $ab^R$  vale  $ab$ , poiché  $b^R = b$ , mentre  $(ab)^R = ba$ .

### 2.2.2 Operazioni sui linguaggi

Vale il principio generale che, se una operazione è definita su una stringa, essa può applicarsi a tutte le stringhe del linguaggio, e in tale modo risultare definita anche sul linguaggio stesso. Con questa interpretazione, si rivisitano ora le operazioni precedenti, cominciando da quelle aventi un solo argomento. La riflessione d'un linguaggio  $L$  è l'insieme delle stringhe che sono riflessione di frasi del linguaggio:

$$L^R = \{x \mid \underbrace{x = y^R \wedge y \in L}_{\text{predicato caratteristico}}\}$$

In questa definizione le stringhe  $x$  sono caratterizzate da una proprietà, espressa dal predicato caratteristico.

Analogamente possiamo definire l'insieme dei prefissi propri d'un linguaggio  $L$ :

$$\text{Prefissi}(L) = \{y \mid x = yz \wedge x \in L \wedge y \neq \varepsilon \wedge z \neq \varepsilon\}$$

*Esempio 2.5. Linguaggio privo di prefissi*

In certe applicazioni si vuole essere certi che la caduta d'una o più lettere finali d'una frase del linguaggio faccia sì che la stringa non sia più valida: un troncamento accidentale sarà così scoperto dal compilatore.

Un linguaggio è *privò di prefissi* se nessuno dei prefissi propri delle sue frasi appartiene al linguaggio stesso, ossia se l'insieme  $\text{Prefissi}(L)$  è disgiunto da  $L$ . Ad es. è privo di prefissi il linguaggio  $L_1 = \{x \mid x = a^n b^n \wedge n \geq 1\}$  perché ogni prefisso ha la forma  $a^n b^m$ ,  $n > m \geq 0$  e non soddisfa il predicato caratteristico. Invece non è privo di prefissi il linguaggio  $L_2 = \{a^m b^n \mid m > n \geq 1\}$  perché contiene sia  $a^3 b^2$  sia il suo prefisso  $a^3 b$ .

Anche le operazioni definite su due stringhe possono essere estese a due linguaggi, quantificando il primo argomento sul primo linguaggio e il secondo argomento sul secondo.

Il *concatenamento* dei linguaggi  $L'$  e  $L''$  risulta così definito come

$$L'L'' = \{xy \mid x \in L' \wedge y \in L''\}$$

Si può ora riformulare la definizione della *potenza  $m$ -esima* per un intero linguaggio:

$$L^m = L^{m-1}L, \quad m > 0$$

$$L^0 = \{\varepsilon\}$$

Dalle definizioni precedenti scendono alcuni casi particolari:

$$\emptyset^0 = \{\varepsilon\}$$

$$L.\emptyset = \emptyset.L = \emptyset$$

$$L.\{\varepsilon\} = \{\varepsilon\}.L = L$$

*Esempio 2.6.* Si prendano i linguaggi:

$$L_1 = \{a^i \mid i \geq 0, \text{ pari}\} = \{\varepsilon, aa, aaaa, \dots\}$$

$$L_2 = \{b^j a \mid j \geq 1, \text{ dispari}\} = \{ba, bbba, \dots\}$$

Risulta allora:

$$\begin{aligned} L_1 L_2 &= \{a^i \cdot b^j a \mid (i \geq 0, \text{ pari}) \wedge (j \geq 1, \text{ dispari})\} \\ &= \{\varepsilon ba, a^2 ba, a^4 ba, \dots, \varepsilon b^3 a, a^2 b^3 a, a^4 b^3 a, \dots\} \end{aligned}$$

È da osservare che il linguaggio ottenuto, applicando la potenza alle *stesse* stringhe del linguaggio base  $L$ , è diverso e incluso nella potenza del linguaggio:

$$\{x \mid x = y^m \wedge y \in L\} \subseteq L^m, \quad m \geq 2$$

Ad es. per  $L = \{a, b\}$  con  $m = 2$  il primo membro vale  $\{aa, bb\}$  e il secondo vale  $\{aa, ab, ba, bb\}$

*Esempio 2.7.* Stringhe di lunghezza finita

L'operatore di potenza permette di definire in modo espressivo il linguaggio delle stringhe di lunghezza non superiore a un intero  $k$  finito. Con l'alfabeto  $\Sigma = \{a, b\}$  e per  $k = 3$ , il linguaggio

$$L = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb\} = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3$$

si formula più sinteticamente come

$$L = \{\varepsilon, a, b\}^3$$

notando che le frasi più corte di  $k$  stanno nel risultato grazie alla presenza della stringa vuota nel linguaggio base della potenza.

Modificando lievemente l'esempio, il linguaggio  $\{x \mid 1 \leq |x| \leq 3\}$  è espresso mediante il concatenamento e la potenza, dalla formula

$$L = \{a, b\} \{ \varepsilon, a, b \}^2$$

### 2.2.3 Operazioni insiemistiche

Naturalmente le consuete operazioni insiemistiche di unione ( $\cup$ ), intersezione ( $\cap$ ) e differenza ( $\setminus$ ), sono definite anche per i linguaggi in quanto insiemi di stringhe; come pure le relazioni di inclusione ( $\subseteq$ ), di inclusione stretta ( $\subset$ ) e di egualanza ( $=$ ).

Prima di parlare del complemento d'un linguaggio si deve introdurre il concetto di *linguaggio universale*, definito come l'insieme di tutte le stringhe di alfabeto  $\Sigma$ , di qualsiasi lunghezza, anche zero.

Chiaramente il linguaggio universale è infinito e può essere pensato come l'unione di tutte le potenze dell'alfabeto

$$L_{\text{universale}} = \Sigma^0 \cup \Sigma \cup \Sigma^2 \cup \dots$$

Il complemento d'un linguaggio  $L$  di alfabeto  $\Sigma$ , scritto  $\neg L$ , è definito come la differenza insiemistica, rispetto al linguaggio universale

$$\neg L = L_{\text{universale}} \setminus L$$

ossia come l'insieme delle stringhe di alfabeto  $\Sigma$  che non stanno in  $L$ . Se l'alfabeto è sottinteso, si può scrivere  $L_{\text{universale}} = \neg \emptyset$ .

*Esempio 2.8.* Il complemento d'un linguaggio finito è sempre infinito, ad es. l'insieme delle stringhe di ogni lunghezza tranne che due vale:

$$\neg(\{a, b\}^2) = \varepsilon \cup \{a, b\} \cup \{a, b\}^3 \cup \dots$$

Non è però detto che il complemento d'un linguaggio infinito sia finito, come si vede dal complemento delle stringhe di lunghezza pari di alfabeto  $\{a\}$ :

$$L = \{a^{2n} \mid n \geq 0\} \quad \neg L = \{a^{2n+1} \mid n \geq 0\}$$

Per la differenza, dato l'alfabeto  $\Sigma = \{a, b, c\}$  e i linguaggi

$$L_1 = \{x \mid |x|_a = |x|_b = |x|_c \geq 0\}$$

$$L_2 = \{x \mid |x|_a = |x|_b \wedge |x|_c = 1\}$$

risulta:

$$L_1 \setminus L_2 = \varepsilon \cup \{x \mid |x|_a = |x|_b = |x|_c \geq 2\}$$

l'insieme delle stringhe aventi lo stesso numero, purché non 1, di comparse delle tre lettere  $a, b, c$ .

$$L_2 \setminus L_1 = \{x \mid |x|_a = |x|_b \neq |x|_c = 1\}$$

l'insieme delle stringhe aventi una sola  $c$  e eguale numero di  $a, b$ , purché diverso da 1.

#### 2.2.4 Stella e croce

Nella maggior parte dei linguaggi artificiali e naturali le frasi possono essere allungate a piacere, quindi il loro numero è illimitato. Ma gli operatori finora studiati, con l'eccezione del complemento, non permettono di scrivere formule finite per definire dei linguaggi infiniti.

Allo scopo di permettere la costruzione di stringhe di qualsiasi lunghezza, sarà introdotto un nuovo importante operatore per denotare il passaggio al limite dell'elevamento a potenza.

L'operatore di *stella* di Kleene (detto anche chiusura rispetto al concatenamento) è definito come l'unione di tutte le potenze del linguaggio:

$$L^* = \bigcup_{h=0 \dots \infty} L^h = L^0 \cup L^1 \cup L^2 \cup \dots = \varepsilon \cup L \cup L^2 \cup \dots$$

*Esempio 2.9.* Dato  $L = \{ab, ba\}$  risulta

$$L^* = \{\varepsilon, ab, ba, abab, abba, baab, baba, \dots\}$$

Ogni stringa del linguaggio stella può essere segmentata in sottostringhe appartenenti al linguaggio base  $L$ .

Si noti che, pur essendo finito il linguaggio base  $L$ , il linguaggio «stellato»  $L^*$  è infinito.

Talvolta il linguaggio stellato coincide con quello di base

$$L = \{a^{2n} \mid n \geq 0\} \quad L^* = \{a^{2n} \mid n \geq 0\} \equiv L$$

Se come linguaggio di base si prende l'alfabeto  $\Sigma$ , la sua stella  $\Sigma^*$  contiene tutte le stringhe<sup>4</sup> che possono essere costruite concatenando i caratteri terminali. Tale linguaggio è dunque il *linguaggio universale* di alfabeto  $\Sigma$ , precedentemente definito.<sup>5</sup>

Evidentemente ogni linguaggio formale è un sottoinsieme del linguaggio universale di eguale alfabeto, e la scrittura

$$L \subseteq \Sigma^*$$

è spesso impiegata per dire che  $L$  è un linguaggio di alfabeto  $\Sigma$ .

Alcune proprietà della stella:

$$L \subseteq L^* \quad (\text{monotonicità})$$

$$\text{se } (x \in L^* \wedge y \in L^*) \text{ allora } xy \in L^* \quad (\text{chiusura resp. a concatenamento})$$

$$(L^*)^* = L^* \quad (\text{idempotenza})$$

$$(L^R)^* = (L^*)^R \quad (\text{commutatività di stella e riflessione})$$

*Esempio 2.10. Idempotenza*

Per la proprietà di monotonicità ogni linguaggio è incluso nella propria stella. Ma per il linguaggio  $L_1 = \{a^{2n} \mid n \geq 0\}$  si può dimostrare l'identità  $L_1^* = L_1$  grazie alla proprietà di idempotenza e all'osservazione che  $L_1$  è anche espresso dalla formula stellata  $\{aa\}^*$

Per il linguaggio vuoto e la stringa vuota si ha

$$\emptyset^* = \{\varepsilon\} \quad \{\varepsilon\}^* = \{\varepsilon\}$$

---

<sup>4</sup>Le frasi di  $\Sigma^*$  hanno lunghezza illimitata ma non infinita. Un ramo della teoria (si veda Perrin e Pin [39]) studia invece i linguaggi detti infinitari o omega-linguaggi, aventi anche frasi di lunghezza infinita. Essi modellano le situazioni in cui un sistema capace di funzionare eternamente può emettere o ricevere messaggi di lunghezza infinita.

<sup>5</sup>Esso è anche chiamato il *monoide libero*. In algebra un monoide è una struttura avente una legge associativa di composizione (concatenamento) e un elemento neutro (stringa vuota).

*Esempio 2.11.* Identificatori

Molti linguaggi artificiali assegnano dei nomi o identificatori alle entità (variabili, documenti, sottoprogrammi, classi, ecc.) di cui fanno uso. Una regola, comune a tanti linguaggi, dice che un identificatore è una stringa iniziente con una lettera  $\{A, B, \dots, Z\}$  e contenente un numero qualsiasi di lettere e di cifre  $\{0, 1, \dots, 9\}$ , ad es. CICLO3A2.

Definiti gli alfabeti

$$\Sigma_A = \{A, B, \dots, Z\}, \quad \Sigma_N = \{0, 1, \dots, 9\}$$

il linguaggio  $I \subseteq (\Sigma_A \cup \Sigma_N)^*$  degli identificatori risulta:

$$I = \Sigma_A(\Sigma_A \cup \Sigma_N)^*$$

Volendo limitare a 5 la lunghezza degli identificatori, posto  $\Sigma = \Sigma_A \cup \Sigma_N$ , si ha

$$I_5 = \Sigma_A(\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \Sigma^4) = \Sigma_A(\epsilon \cup \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup \Sigma^4)$$

che esprime il concatenamento del linguaggio  $\Sigma_A$ , le cui frasi sono singoli caratteri, con il linguaggio unione degli altri cinque. Più elegantemente si scrive:

$$I_5 = \Sigma_A(\epsilon \cup \Sigma)^4$$

*Croce*

Un operatore, utile ma non indispensabile, derivato dalla stella, è la *croce* (o chiusura non riflessiva rispetto al concatenamento)

$$L^+ = \bigcup_{h=1 \dots \infty} L^h = L \cup L^2 \cup \dots$$

che si distingue dalla stella perché l'unitoria non contiene la potenza zero. Valgono le relazioni:

$$\begin{aligned} L^+ &\subseteq L^* \\ \epsilon \in L^+ &\text{ se e solo se } \epsilon \in L \\ L^+ &= LL^* = L^*L \end{aligned}$$

*Esempio 2.12.*

$$\{ab, bb\}^+ = \{ab, b^2, ab^3, b^2ab, abab, b^4, \dots\}$$

$$\{\epsilon, aa\}^+ = \{\epsilon, a^2, a^4, \dots\} = \{a^{2n} \mid n \geq 0\}$$

Spesso lo stesso linguaggio può essere definito in più modi che differiscono nell'uso degli operatori.

*Esempio 2.13.* Le stringhe lunghe almeno quattro caratteri possono essere ottenute nei due modi seguenti:

- concatenando le stringhe di lunghezza quattro con altre stringhe arbitrarie:  $\Sigma^4 \Sigma^*$ ;
- costruendo la quarta potenza dell'insieme delle stringhe non vuote:  $(\Sigma^+)^4$ .

### 2.2.5 Quoziente

Quasi tutte le operazioni viste allungano le stringhe o aumentano la cardinalità del linguaggio su cui operano. Dati due linguaggi, l'operazione di *quoziente(destro)* accorcia una frase del primo linguaggio, decurtandola d'un suffisso, appartenente al secondo.

Il quoziente (destro) di  $L'$  rispetto a  $L''$  è definito come

$$L = L' /_D L'' = \{y \mid (x = yz \in L') \wedge z \in L''\}$$

*Esempio 2.14.* Siano

$$L' = \{a^{2n}b^{2n} \mid n > 0\}, \quad L'' = \{b^{2n+1} \mid n \geq 0\}$$

Allora si hanno i quozienti:

$$L' /_D L'' = \{a^r b^s \mid (r \geq 2, \text{ pari}) \wedge (1 \leq s < r, s \text{ dispari})\} = \{a^2 b, a^4 b, a^4 b^3, \dots\}$$

$$L'' /_D L' = \emptyset$$

L'operazione duale è il *quoziente sinistro*  $L'' /_S L'$  che accorcia una frase del primo linguaggio decurtandola di un prefisso appartenente al secondo linguaggio.

Altre operazioni saranno introdotte in seguito allo scopo di trasformare o tradurre i linguaggi formali sostituendo i caratteri terminali con altri.

## 2.3 Espressioni e linguaggi regolari

La ricerca teorica sui linguaggi formali, similmente a quanto anticamente avvenuto nel campo della teoria dei numeri, ha dato vita a una varietà di categorie di linguaggi, caratterizzate dalle proprietà matematiche e algoritmiche dei linguaggi corrispondenti. La prima semplice famiglia di linguaggi formali che si incontra è quella detta *regolare* (o razionale) che può essere definita in un numero sorprendente di modi molto diversi. I linguaggi regolari sono infatti sorti indipendentemente in campi di studio distinti: esaminando le sequenze temporali dei segnali di ingresso che portano una rete sequenziale<sup>6</sup> in un certo stato; descrivendo il lessico dei linguaggi di programmazione con certe semplici grammatiche; e analizzando il comportamento di modelli semplificati di

---

<sup>6</sup>Una rete sequenziale è un circuito digitale dotato di memoria.

reti di neuroni. A questi tre approcci si sono più tardi aggiunte le definizioni logiche basate sul calcolo dei predicati.

La prima definizione della famiglia è di stile algebrico e si basa sugli operatori di unione, concatenamento e stella; poi la stessa famiglia sarà definita per mezzo di certe semplici grammatiche; infine nel cap. 3 si vedranno gli automi finiti, ovvero gli algoritmi di riconoscimento dei linguaggi regolari.<sup>7</sup>

### 2.3.1 Definizione di espressione regolare

Un linguaggio di alfabeto  $\Sigma = \{a_1, a_2, \dots, a_n\}$  è detto *regolare* se può essere espresso mediante le operazioni di concatenamento, unione e stella applicate a un numero finito di volte ai linguaggi unitari<sup>8</sup>  $\{a_1\}, \{a_2\}, \dots, \{a_n\}$  e al linguaggio vuoto  $\emptyset$ .

Più rigorosamente, una *espressione regolare* (abbreviata in e.r.) è una stringa  $r$  costruita con i caratteri dell'alfabeto terminale  $\Sigma$  e con i metasimboli<sup>9</sup>

. concatenamento       $\cup$  unione      \* stella       $\emptyset$  ins. vuoto      ( )

in accordo con le regole sotto elencate:

- 1.  $r = \emptyset$
- 3.  $r = (s \cup t)$
- 5.  $r = (s)^*$
- 2.  $r = a, a \in \Sigma$
- 4.  $r = (s.t)$  o anche  $r = (st)$

dove  $s$  e  $t$  sono e.r..

Le parentesi, quando superflue, si possono eliminare, convenendo che nella precedenza tra gli operatori vi sia l'ordine: stella, concatenamento e unione.

Per una migliore espressività, anche il simbolo  $\varepsilon$  (stringa vuota) e la croce possono essere usati in una e.r., essendo essi derivabili dagli operatori di base, grazie alle note identità  $\varepsilon = \emptyset^*$  e  $s^+ = s(s)^*$ .

Spesso il simbolo di unione viene scritto, invece che come coppa ' $\cup$ ', come barra verticale '|', chiamata *alternativa*.

Le regole precedenti definiscono la sintassi delle e.r., che sarà anche formalizzata per mezzo d'una grammatica più avanti (es. 2.31, p. 31).

Il significato d'una e.r.  $r$  è un linguaggio  $L_r$  di alfabeto  $\Sigma$ , secondo la seguente tabella di corrispondenza:

<sup>7</sup>Una famiglia di linguaggi può essere anche definita dal tipo di predicati logici che caratterizzano le frasi del linguaggio. Al riguardo si veda [48].

<sup>8</sup>Ossia contenenti una sola stringa.

<sup>9</sup>Al fine di evitare confusione tra caratteri terminali e metasimboli, si suppone che i metasimboli siano assentiti dall'alfabeto terminale. Se così non fosse, i metasimboli dovrebbero essere opportunamente ricodificati.

| <i>espressione r</i>       | <i>linguaggio denotato L<sub>r</sub></i> |
|----------------------------|--|
| 1. $\varepsilon$           | $\{\varepsilon\}$                        |
| 2. $a \in \Sigma$          | $\{a\}$                                  |
| 3. $s \cup t$ o $s \mid t$ | $L_s \cup L_t$                           |
| 4. $s.t$ o $st$            | $L_s, L_t$                               |
| 5. $s^*$                   | $L_s^*$                                  |

*Esempio 2.15.* Sia  $\Sigma = \{1\}$ , dove 1 è pensabile come un segnale. Il linguaggio denotato dall'espressione

$$e = (111)^*$$

contiene le sequenze di segnali multiple di tre

$$L_e = \{1^n \mid n \bmod 3 = 0\}$$

Si badi che, omettendo le parentesi tonde, il linguaggio cambia, a causa della precedenza del concatenamento rispetto all'unione

$$e_1 = 111^* = 11(1)^* \quad L_{e_1} = \{1^n \mid n \geq 2\}$$

*Esempio 2.16.* Numeri interi.

Sia  $\Sigma = \{+, -, d\}$  dove  $d$  denota una cifra decimale  $0, 1, \dots, 9$ . L'espressione

$$e = (+ \cup - \cup \varepsilon)dd^* \equiv (+ \mid - \mid \varepsilon)dd^*$$

produce il linguaggio

$$L_e = \{+, -, \varepsilon\}\{d\}\{d\}^*$$

dei numeri interi con o senza segno, quali  $+353, -5, 969, +001$ .

Poiché la corrispondenza tra la e.r. e il linguaggio denotato è piuttosto immediata, si suole denotare il linguaggio  $L_e$  mediante l'espressione  $e$  stessa.

Un linguaggio è detto *regolare* se è denotato da una espressione regolare. La collezione di tutti i linguaggi regolari costituisce un insieme di linguaggi, la *famiglia REG* dei linguaggi *regolari*.

Un'altra semplice famiglia di linguaggi è quella dei *linguaggi finiti*, *FIN*. Un linguaggio appartiene a *FIN* se la sua cardinalità è finita, ad es. il linguaggio dei numeri binari di 32 bit. Confrontando le famiglie *REG* e *FIN*, è immediato vedere che ogni linguaggio finito è regolare, ossia  $FIN \subseteq REG$ . Basta infatti osservare che un linguaggio finito è l'unione d'un numero finito di stringhe  $x_1, x_2, \dots, x_k$  ognuna delle quali è il concatenamento d'un numero finito di caratteri,  $x_i = a_1 a_2 \dots a_{n_i}$ . Il linguaggio è dunque denotato dalla e.r. costituita dall'unione di  $k$  termini, ognuno dei quali è il concatenamento di  $n_i$  caratteri terminali. Poiché inoltre *REG* contiene anche linguaggi di cardinalità non finita, la inclusione tra le due famiglie è stretta,  $FIN \subset REG$ .

Altre famiglie di linguaggi saranno introdotte e confrontate con *REG* nel corso del libro.

### 2.3.2 Derivazione e linguaggio

Per definire più precisamente il linguaggio generato da una e.r., si inizia dal concetto di *sottoespressione* (s.e.) supponendo dapprima che la e.r. data  $e$  sia completamente parentesizzata (tranne per semplicità i termini atomici):

$$e_0 = \left( \overbrace{(a \cup (bb))^*}^{e_1} \right) \left( \overbrace{(c^+) \cup \underbrace{(a \cup (bb))}_s}^{e_2} \right)$$

La struttura di questa e.r. è il concatenamento di due parti  $e_1$  e  $e_2$  che sono dette *sottoespressioni*.

In generale una s.e.  $f$  d'una e.r.  $e$  è una sottostringa ben parentesizzata di  $e$  contenuta direttamente entro le parentesi più esterne. In altro modo, non esiste in  $e$  un'altra sottostringa ben parentesizzata che contenga  $f$  al proprio interno.

Nell'esempio, la sottostringa marcata  $s$  non è s.e. di  $e_0$  ma è s.e. di  $e_2$ .

Se la e.r. non è completamente parentesizzata, per identificare le sue s.e. è necessario prima aggiungere (magari solo mentalmente) le parentesi, tenendo conto delle precedenze convenzionali tra gli operatori.

Si noti però che tre o più termini posti in unione possono essere raggruppati in più modi diversi ma equivalenti, essendo associativa l'operazione. Si vedano le due parentesizzazioni:

$$(c^+ \cup a \cup bb) \text{ parentesizzata come: } \overbrace{(c^+ \cup a \cup bb)}^{\text{un solo termine}} \quad (c^+ \cup \overbrace{a \cup bb}^{\text{due termini}})$$

Lo stesso vale se nell'e.r. ci sono tre o più termini concatenati.

Le s.e. d'una e.r.  $e_0$  sono allora le s.e. delle e.r. completamente parentesizzate, ottenute parentesizzando in tutti i modi possibili l'espressione data  $e_0$ .

Gli operatori di unione e di ripetizione presenti in una e.r. designano possibili scelte di stringhe. Fissando una scelta, si ottiene una e.r. che definisce un linguaggio incluso nel precedente. Più precisamente, si dice che una e.r. è una *scelta* di un'altra nei seguenti casi:

1.  $e_k, 1 \leq k \leq n$ , è una scelta dell'unione  $e_1 \cup \dots \cup e_k \cup \dots \cup e_n$
2.  $e^n = \underbrace{e \dots e}_n, n \geq 1$  è una scelta delle espressioni  $e^*$ ,  $e^+$   
 $n$  volte
3. la stringa vuota è una scelta di  $e^*$

Da una e.r.  $e'$  data si può derivare una seconda e.r.  $e''$ , sostituendo al posto d'una s.e.  $\beta$  di  $e'$  una e.r. scelta da essa.

**Definizione 2.17. Derivazione<sup>10</sup>**

*La relazione di derivazione tra due espressioni regolari  $e', e''$  è così definita: da  $e'$  deriva  $e''$ , scritto  $e' \Rightarrow e''$ , se le due e.r. si possono fattorizzare come*

$$e' = \alpha\beta\gamma, \quad e'' = \alpha\delta\gamma$$

*dove  $\beta$  è una s.e. di  $e'$ , e  $\delta$  è una scelta di  $\beta$ .*

La relazione di derivazione è applicabile più volte di seguito. Si dice che  $e_n$  deriva da  $e_0$  in  $n$  passi, scritto

$$e_0 \xrightarrow{n} e_n$$

se

$$e_0 \Rightarrow e_1, \quad e_1 \Rightarrow e_2, \quad \dots, \quad e_{n-1} \Rightarrow e_n$$

La scrittura

$$e_0 \stackrel{+}{\Rightarrow} e_n$$

indica che  $e_0$  deriva  $e_n$  in  $n \geq 1$  passi. Se si vuole includere il caso  $n = 0$ , ossia l'identità  $e_0 = e_n$ , si scrive la stella al posto della croce.

**Esempio 2.18.** Derivazioni immediate:

$$a^* \cup b^+ \Rightarrow a^*, \quad a^* \cup b^+ \Rightarrow b^+, \quad (a^* \cup bb)^* \Rightarrow (a^* \cup bb)(a^* \cup bb)$$

Si noti che le scelte vanno fatte dall'esterno all'interno. Ad es. da  $e' = (a^* \cup bb)^*$  non deriva  $(a^2 \cup bb)^*$ , perché  $a^*$  non è una s.e. di  $e'$ , anche se l'esponente 2 è una scelta valida per la stella.

Derivazioni a più passi:

$$a^* \cup b^+ \Rightarrow a^* \Rightarrow \epsilon \text{ ossia } a^* \cup b^+ \xrightarrow{2} \epsilon \text{ o anche } a^* \cup b^+ \stackrel{+}{\Rightarrow} \epsilon$$

$$a^* \cup b^+ \Rightarrow b^+ \Rightarrow bbb \text{ o anche } (a^* \cup b^+) \stackrel{+}{\Rightarrow} bbb$$

Tra le e.r. ottenute per derivazione da una espressione  $r$ , alcune contengono anche i metasimboli (operatori e parentesi), altre soltanto i simboli terminali e la stringa vuota. Queste ultime costituiscono il linguaggio definito dalla e.r. Il *linguaggio definito da una espressione regolare  $r$*  è

$$L(r) = \{x \in \Sigma^* \mid r \xrightarrow{*} x\}$$

Due e.r. sono dette *equivalenti* se definiscono lo stesso linguaggio.

Nel prossimo esempio, ordini diversi di applicazione delle scelte producono la stessa frase del linguaggio.

**Esempio 2.19.** Si hanno le seguenti derivazioni:

---

<sup>10</sup>Anche chiamata *implicazione*.

1.  $a^*(b \cup c \cup d)f^+ \Rightarrow aaa(b \cup c \cup d)f^+ \Rightarrow aaacf^+ \Rightarrow aaacf$
2.  $a^*(b \cup c \cup d)f^+ \Rightarrow a^*cf^+ \Rightarrow aaacf^+ \Rightarrow aaacf$

Si confrontino le derivazioni 1. e 2. La 1. sceglie la s.e. posta più a sinistra ( $a^*$ ), mentre la 2. sviluppa la s.e. ( $b \cup c \cup d$ ) che non è quella più a sinistra. I due passi sono indipendenti uno dall'altro e possono essere eseguiti in qualsiasi ordine. Applicando un altro passo si ottiene la e.r.  $aaacf^+$ , e l'ultimo passo produce la frase  $aaacf$ . Anche l'ultimo passo, essendo indipendente dagli altri, avrebbe potuto essere eseguito prima, o in mezzo, tra essi.

In conclusione vi sono molti ordini distinti, ma sostanzialmente equivalenti, per produrre una frase del linguaggio.

### Ambiguità delle espressioni regolari

Concettualmente diverso è il seguente esempio, in cui una frase è ottenuta con due derivazioni che differiscono in modo più sostanziale del solo ordine dei passi.

*Esempio 2.20.* Espressione regolare ambigua

Il linguaggio di alfabeto  $\{a, b\}$ , caratterizzato dalla presenza di almeno una lettera  $a$ , è definito dalla e.r.

$$(a \cup b)^*a(a \cup b)^*$$

in cui si evidenzia una comparsa obbligatoria della  $a$ . Le frasi contenenti due o più lettere  $a$  sono ottenibili per mezzo di diverse derivazioni, che si differenziano a seconda di quale delle lettere  $a$  corrisponda alla lettera centrale della e.r. Ad es. la stringa  $aa$  offre due diverse possibilità:

$$(a \cup b)^*a(a \cup b)^* \Rightarrow (a \cup b)a(a \cup b)^* \Rightarrow aa(a \cup b)^* \Rightarrow aa\varepsilon = aa$$

$$(a \cup b)^*a(a \cup b)^* \Rightarrow \varepsilon a(a \cup b)^* \Rightarrow \varepsilon a(a \cup b) \Rightarrow \varepsilon aa = aa$$

Tale frase (e la e.r. che la definisce) è detta ambigua, in quanto esistono due modi strutturalmente diversi per generarla. Invece la frase  $ba$  non è ambigua, avendo la sola derivazione

$$(a \cup b)^*a(a \cup b)^* \Rightarrow (a \cup b)a(a \cup b)^* \Rightarrow ba(a \cup b)^* \Rightarrow ba\varepsilon = ba$$

Per precisare il concetto di ambiguità, basta numerare, come già visto, le lettere <sup>11</sup> della e.r.  $f$  ottenendo la e.r. numerata :

$$f' = (a_1 \cup b_2)^*a_3(a_4 \cup b_5)^*$$

Essa definisce un linguaggio regolare di alfabeto  $\{a_1, b_2, a_3, a_4, b_5\}$ .

Una e.r.  $f$  è *ambigua* se, e solo se, nel linguaggio definito dalla corrispondente

<sup>11</sup> Il simbolo  $\varepsilon$  eventualmente presente non va numerato.

e.r. marcata  $f'$  vi sono due diverse stringhe  $x, y$  tali che, cancellando i numeri, esse vengono a coincidere.

Nell'es. le stringhe  $a_1a_3$  e  $a_3a_4$  del linguaggio di  $f'$  dimostrano l'ambiguità. Le definizioni ambigue sono problematiche in molte applicazioni, pur se possono avere il pregio della concisione. Il concetto di ambiguità sarà studiato in modo più completo nell'ambito delle grammatiche.

### 2.3.3 Altri operatori

Nelle applicazioni si suole per comodità ammettere nelle e.r., non solo gli *operatori di base* (unione, concatenamento, stella), ma anche gli operatori di elevamento a potenza e croce, derivabili da essi.

Per un ulteriore guadagno in concisione, si usano anche altri operatori, che sono derivabili dai precedenti:

Ripetizione da  $k$  a  $n > k$  volte:  $[a]^n_k = a^k \cup a^{k+1} \cup \dots a^n$

Opzionalità:  $[a] = (\epsilon \cup a)$

Intervallo d'un insieme ordinato: per indicare ogni cifra appartenente all'insieme ordinato  $0, 1, \dots, 9$  vale la stenografia  $(0 \dots 9)$ .

Similmente si può scrivere  $(a \dots z)$  al posto dell'insieme delle lettere minuscole e  $(A \dots Z)$  per quelle maiuscole.

Talvolta si considerano anche altri operatori insiemistici, l'intersezione, la differenza e il complemento. Le e.r. prendono allora il nome di espressioni regolari estese con detti operatori.

#### Esempio 2.21. Operazione di intersezione

Questo operatore permette di esprimere più direttamente il fatto che le frasi del linguaggio devono soddisfare contemporaneamente due condizioni. Per illustrare, sia  $\{a, b\}$  l'alfabeto e supponiamo che una stringa per essere valida debba (1) contenere la sottostringa  $bb$  e (2) avere lunghezza pari. La prima condizione è formalizzata dalla e.r.

$$(a \mid b)^* bb(a \mid b)^*$$

la seconda dalla e.r.

$$((a \mid b)^2)^*$$

e il linguaggio dalla e.r. estesa con l'intersezione

$$((a \mid b)^* bb(a \mid b)^*) \cap ((a \mid b)^2)^*$$

Lo stesso linguaggio può essere definito da una e.r. senza l'intersezione, più complicata, esprimente il fatto che la sottostringa  $bb$  può essere attorniata da due stringhe di lunghezza pari o da due stringhe di lunghezza dispari:

$$((a \mid b)^2)^* bb((a \mid b)^2)^* \mid (a \mid b)((a \mid b)^2)^* bb(a \mid b)((a \mid b)^2)^*$$

Qualche volta è più semplice definire le frasi d'un linguaggio ex negativo, enunciando una proprietà che esse non devono avere.

*Esempio 2.22.* Operazione di complemento

Il linguaggio è l'insieme  $L$  delle stringhe di alfabeto  $\{a, b\}$  che non contengono  $aa$  come sottostringa. Il complemento del linguaggio è

$$\neg L = \{x \in (a \mid b)^* \mid x \text{ contiene la sottostringa } aa\}$$

facilmente denotato dalla e.r.  $(a \mid b)^*aa(a \mid b)^*$ , da cui la e.r. estesa

$$L = \neg((a \mid b)^*aa(a \mid b)^*)$$

La definizione mediante una e.r. non estesa

$$(ab \mid b)^*(a \mid \varepsilon)$$

è forse meno comprensibile.

Non è un caso che sia stato possibile eliminare l'intersezione e il complemento dalle e.r. precedenti. Infatti un risultato teorico, esposto nel capitolo 3, afferma che, anche se una e.r. fa uso degli operatori di complemento e intersezione, il linguaggio da essa definito è regolare e di conseguenza può essere definito da una e.r. non estesa.

### 2.3.4 Chiusura della famiglia $REG$ rispetto alle operazioni

Sia  $\vartheta$  un operatore che applicato a un linguaggio, o a una coppia di linguaggi, ne produce un altro. Una famiglia di linguaggi si dice *chiusa rispetto a un operatore*  $\vartheta$  se il linguaggio risultante dall'applicazione di  $\vartheta$  ai linguaggi della famiglia appartiene ancora alla stessa famiglia.

*Proprietà 2.23.* La famiglia  $REG$  dei linguaggi regolari è chiusa rispetto agli operatori di concatenamento, unione, stella (quindi anche rispetto agli operatori derivati come la croce).

Ciò è evidente dalla stessa definizione di e.r..

Il significato pratico è che due linguaggi regolari possono essere combinati tra di loro con detti operatori senza pericolo di fuoriuscire dai linguaggi definibili con e.r..

Inoltre la famiglia  $REG$  risulta chiusa rispetto a intersezione, complemento e riflessione, ma per dimostrarlo si devono attendere i concetti della teoria degli automi nel prossimo capitolo.

Un'affermazione più forte della proprietà 2.23 è la seguente.

*Proprietà 2.24.* La famiglia  $REG$  dei linguaggi regolari è la più piccola famiglia di linguaggi che (i) contiene tutti i linguaggi finiti e (ii) è chiusa rispetto a concatenamento, unione e stella.

La dimostrazione è semplice. Si supponga per assurdo che esista una famiglia  $F \subset REG$ , chiusa rispetto agli stessi operatori e contenente ogni linguaggio finito. Preso un qualsiasi linguaggio  $L(e)$  definito dalla e.r.  $e$ , esso è ottenuto applicando ripetutamente gli operatori presenti nella  $e$  ai linguaggi finiti costituiti dai singoli caratteri terminali. Per l'ipotesi, il linguaggio  $L(e)$  appartiene anche alla famiglia  $F$ , la quale dunque conterebbe ogni linguaggio regolare, contraddicendo la inclusione  $F \subset REG$ .

Esistono altre famiglie con la stessa proprietà di chiusura 2.23, tra le quali spicca quella  $LIB$  dei linguaggi liberi dal contesto, di prossima introduzione. Dall'enunciato segue che tra le due famiglie vi è inclusione,  $REG \subset LIB$ .

## 2.4 Astrazione linguistica

Se si andassero a vedere i linguaggi tecnici esistenti, si constaterebbe che essi, al di là dell'apparente diversità delle loro forme, sono molto simili. Per sfondare i linguaggi delle diversità superficiali e ridurli alle loro strutture essenziali, si sposterà l'attenzione dalla sintassi concreta verso quella astratta. Si ricorda, dal dizionario Zingarelli, che *astrarre* significa 'separare mentalmente nell'oggetto dato (per noi il linguaggio) qualche proprietà per considerarla separatamente'. Nell'arte l'*astrattismo* tende ad astrarre da ogni rappresentazione delle forme della realtà sensibile. Nel caso attuale, la realtà sensibile è il linguaggio reale che si vuole progettare, analizzare o trasformare. Le forme della realtà sensibile sono la rappresentazione concreta del linguaggio con i caratteri dell'alfabeto.

L'astrazione linguistica trasforma le frasi d'un linguaggio reale, detto concreto, in una forma più semplice, detta rappresentazione astratta. Essa trascura in qualche misura i simboli dell'alfabeto concreto, e impiega al loro posto i caratteri d'un altro alfabeto, quello astratto.

Al livello astratto, le strutture sintattiche dei linguaggi artificiali si possono ottenere come composizione di pochi paradigmi elementari, attraverso le operazioni di concatenamento di unione, di iterazione e di sostituzione (mostrata dopo).

Per ottenere il linguaggio effettivo, il linguaggio astratto, ossia la sintassi astratta, va rimpolpato o, come si usa dire, guarnito di 'glossa sintattica' (syntactic sugar), con un'opportuna scelta degli elementi lessicali (parole chiave, delimitatori, identificatori, ecc.).

Questo modo di evidenziare le strutture astratte è uno strumento concettuale di analisi e sintesi molto efficace, non solo per analizzare e progettare i linguaggi, ma anche per realizzare i compilatori. Infatti tale approccio porta ad applicare le funzioni del compilatore non al linguaggio reale (ad es. FORTRAN) ma a una rappresentazione astratta del medesimo; consentendo così di riutilizzare le stesse funzioni anche per un altro linguaggio (ad es. C), dopo che anch'esso è stato tradotto nella rappresentazione astratta.

Le strutture astratte che si incontrano nei linguaggi artificiali sono pochissime:

sime, e saranno mostrate, cominciando ora da quelle descrivibili mediante espressioni regolari, le liste.

### 2.4.1 Liste astratte e concrete

Una *lista* contiene un numero imprecisato di elementi *e* dello stesso tipo. Essa è generata dalla e.r.  $e^+$  o da  $e^*$ , se gli elementi possono mancare.

Un elemento può essere per ora visto come un simbolo terminale; ma in una progettazione modulare e per astrazioni successive, esso potrà divenire una stringa, appartenente ad un altro linguaggio formale: si pensi ad es. a una lista di numeri.

#### *Liste con separatori e marche di apertura e chiusura*

Spesso il linguaggio concreto richiede che gli elementi siano tra loro separati da certe stringhe, dette *separatori* *s*, nella sintassi astratta. Per esempio, in una lista di numeri, è necessario interporre un separatore per segnare la fine d'un numero e l'inizio del successivo.

Una *lista con separatori* è definita dalla e.r.  $e(se)^*$  che dice che una lista contiene un elemento, seguito da zero o più coppie *se*. Una definizione equivalente è  $(es)^*e$ , che mette in evidenza l'ultimo elemento invece del primo.

Per facilitare il riconoscimento dell'inizio o della fine della lista, si suole aggiungere un carattere di inizio *i* o di fine *f* lista, detto anche *marca di apertura* o di *chiusura*.

La e.r. delle liste con separatori e marche di apertura e chiusura, diviene:

$$ie(se)^*f$$

#### *Esempio 2.25. Esempi di liste concrete*

Le strutture a lista sono onnipresenti nei linguaggi, tanto naturali quanto artificiali: alcuni esempi tipici sono mostrati.

Blocco di istruzioni: *begin istr<sub>1</sub>; istr<sub>2</sub>; ... istr<sub>n</sub> end*

dove *istr* può essere un assegnamento, un salto, una frase *if*, una frase *write*, ecc. La corrispondenza tra i termini astratti e concreti è la seguente:

| alfabeto astratto | alfabeto concreto |
|-------------------|-------------------|
| <i>i</i>          | <i>begin</i>      |
| <i>e</i>          | <i>istr</i>       |
| <i>s</i>          | <i>;</i>          |
| <i>f</i>          | <i>end</i>        |

Parametri d'una procedura: come

procedure STAMPA(par<sub>1</sub>,par<sub>2</sub>,... , par<sub>n</sub>)

Se la lista dei parametri può essere vuota, come in *procedure STAMPA()*, la e.r. diviene  $i[e(s e)^*]f$ .

Definizione di *array*:  $\underbrace{\text{array } MATRICE'}_i \underbrace{['}_e \underbrace{\text{int}_1}_{s}, \underbrace{\text{int}_2, \dots, \text{int}_n}_{f} \underbrace{]}_f$

dove ogni *int* è un intervallo come 10...50.

## Sostituzione

In tutti gli esempi di liste concrete si è applicato il principio di astrazione, per sostituire a un carattere astratto, come la marca di inizio, un insieme d'una o più stringhe concrete, ossia un linguaggio. In fase di progetto d'un linguaggio artificiale complesso è sempre utile procedere per sviluppi successivi, un po' come si fa in ogni campo dell'ingegneria, quando un sistema viene decomposto in sottosistemi, ognuno dei quali è poi esploso nelle sue componenti. A tale fine, si formalizza l'operazione di sostituzione, che rimpiazza un carattere terminale d'un primo linguaggio (detto sorgente) con le frasi d'un secondo linguaggio (detto pozzo).

Sia al solito  $\Sigma$  l'alfabeto sorgente e  $L \subseteq \Sigma^*$  il linguaggio, detto *sorgente*. Si consideri un carattere sorgente  $b$  e una frase di  $L$  contenente una o più comparse di tale carattere:

$$x = a_1 a_2 \dots a_n \quad \text{dove per qualche } i, a_i = b$$

Sia  $\Delta$  un alfabeto, detto *pozzo*, e  $L_b \subseteq \Delta^*$  il *linguaggio immagine* di  $b$ . La *sostituzione* del linguaggio  $L_b$  al posto di  $b$  nella stringa  $x$  produce un insieme di stringhe, più precisamente un linguaggio di alfabeto  $(\Sigma \setminus \{b\}) \cup \Delta$ , così definito:

$$\{y \mid y = y_1 y_2 \dots y_n \wedge (\text{se } a_i \neq b \text{ allora } y_i = a_i \text{ altrimenti } y_i \in L_b)\}$$

Si noti che i caratteri diversi da  $b$  restano immutati. Al solito modo, si può estendere la definizione di sostituzione all'intero linguaggio sorgente, applicandola a ogni sua frase.

*Esempio 2.26.* Esempio 2.25 continuato.

Riprendendo il caso delle liste dei parametri d'una procedura, alla sintassi astratta

$$ie(se)^*f$$

sono applicate le seguenti sostituzioni:

| car. astratto | immagine   | nota                   |
|---------------|--|------------------------|
| $i$           | $L_i = \text{procedure } \langle \text{ident. di procedura} \rangle()$ | secondo le convenzioni |
| $e$           | $L_e = \langle \text{ident. di parametro} \rangle$                     | secondo le convenzioni |
| $s$           | $L_s = ,$  | virgola                |
| $f$           | $L_f = )$  | parentesi chiusa       |

Ad es. la marca d'inizio  $i$  può essere sostituita da una frase del linguaggio  $L_i$ , dove l'identificatore di procedura è da specificare in conformità con le convenzioni del linguaggio tecnico considerato.

I linguaggi immagine possono cambiare da un linguaggio tecnico a un altro. Si noti che le quattro sostituzioni sono indipendenti e possono essere applicate in qualsiasi ordine.

*Esempio 2.27.* Identificatori con trattino.

Per migliorare la leggibilità, in certi linguaggi tecnici i nomi degli identificatori possono essere suddivisi in uno o più campi, separati da un trattino: un nome lecito è *CICLO3\_DI\_35*. Si precisa che il primo campo deve iniziare con una lettera, mentre gli altri possono contenere qualsiasi stringa alfanumerica non vuota. Non sono permessi due trattini consecutivi, né un trattino alla fine del nome.

In prima approssimazione il linguaggio è una lista di campi  $c$ , separati dal trattino

$$c(\_c)^*$$

A meglio vedere, il primo campo va differenziato dagli altri perché deve iniziare con una lettera, e può essere visto come la marca d'inizio d'una lista anche vuota

$$i(\_c)^*$$

Sostituendo infine a  $i$  il linguaggio  $(A \dots Z)(A \dots Z \mid 0 \dots 9)^*$ , e a  $c$  il linguaggio  $(A \dots Z \mid 0 \dots 9)^+$ , si ottiene la e.r. desiderata.

Il progetto dei linguaggi per raffinamenti successivi sarà sviluppato in questo capitolo dopo l'introduzione delle grammatiche. Altre trasformazioni dei linguaggi saranno studiate nel capitolo 5.

*Liste con precedenze o livelli.*

Costrutti frequenti sono le liste in cui un elemento è a sua volta una lista. La prima lista è detta *primaria* o di livello 1, la seconda di livello 2, ecc. In questa categoria stanno però soltanto le liste che hanno un numero limitato di livelli, mentre il caso illimitato sarà trattato con le grammatiche sotto la rubrica delle strutture annidate. Le liste con un numero limitato di livelli sono anche dette *liste con precedenze*, perché una lista di livello  $k$  ha maggiore forza associativa della lista di livello  $k - 1$ , nel senso che gli elementi del livello maggiore devono essere assemblati a formare un elemento del livello minore. Naturalmente ogni livello può avere marche di inizio, marche di fine e separatori, che per chiarezza sono di solito distinti da livello a livello.

La struttura delle liste a precedenze con  $k \geq 2$  livelli è:

$$\text{lista}_1 = i_1 \text{lista}_2 (s_1 \text{lista}_2)^* f_1$$

$$\text{lista}_2 = i_2 \text{lista}_3 (s_2 \text{lista}_3)^* f_2$$

$$\dots \\ lista_k = i_k e_k (s_k e_k)^* f_k$$

Si incontra anche una variante in cui a ogni livello  $j$ , e non solo all'ultimo, possono comparire degli elementi atomici  $e_j$ , oltre che delle liste del livello immediatamente maggiore. Alcuni esempi concreti di liste a più livelli sono ora evocati.

*Esempio 2.28.* Liste a più livelli.

Blocco di istruzioni di stampa: *begin*  $istr_1; istr_2; \dots istr_n$  *end*

dove  $istr$  è una istruzione di stampa,  $STAMPA(var_1, var_2, \dots, var_n)$  os-  
sia una lista (dall'es. 2.25). I livelli sono:

Livello 1: lista di istruzioni  $istr$ , iniziate da *begin*, separate dal punto e virgola e terminate da *end*

Livello 2: lista di variabili  $var$  separate dalla virgola, con  $i_2 = STAMPA($  e  $f_2 = )$ .

Espressioni aritmetiche senza parentesi: i livelli di precedenza degli operatori determinano il numero di livelli della lista. Ad es. gli operatori  $\times, \div, +, -$  stanno su due livelli e la stringa

$$3 + \underbrace{5 \times 7 \times 4 - 8 \times 2}_{\text{monomio}_1} \div 5 + 8 + 3 \underbrace{-}_{\text{monomio}_2}$$

è una lista a due livelli, senza marche di inizio e fine. Al primo livello sta una lista di monomi ( $e_1 = \text{monomio}_1$ ) separati dai segni  $+$  e  $-$  cioè dagli operatori di minore precedenza. Come caso particolare, un monomio è un numero. Al secondo livello sta una lista di numeri, separati dai segni di maggiore precedenza  $\times, \div$ .

Si potrebbe aggiungere l'operazione di elevamento a potenza  $**$ , introducendo un terzo livello.

Anche nelle lingue naturali le liste a più livelli sono una struttura fondamentale; si pensi a una lista di sostantivi

padre, madre, figlio e figlia

Si noterà una variante rispetto alla lista astratta: il separatore tra il penultimo e l'ultimo elemento è la congiunzione non la virgola, forse allo scopo di avvertire l'ascoltatore che l'elenco sta per finire. Naturalmente ogni sostantivo può arricchirsi di una lista di secondo livello, con gli attributi:

un padre forte, severo e giusto, una madre amorevole e fedele ...

Nei documenti e nei media le liste con livelli sono comunissime. Ad es. un libro è una lista di capitoli, separati da pagine bianche, chiusa tra due copertine; un capitolo è una lista di sezioni; una sezione è una lista di paragrafi, e così via.

## 2.5 Grammatiche generative libere dal contesto

Inizia ora lo studio della famiglia che occupa il posto centrale nella compilazione: i linguaggi liberi dal contesto o in breve liberi. Le grammatiche libere sono un modello nato dalla linguistica negli anni 1950, ma è soprattutto nell'informatica che esse hanno avuto uno straordinario successo applicativo. Infatti tutti i linguaggi tecnici sono stati definiti mediante tali grammatiche. Inoltre sono state sviluppati fin dagli anni 1960 degli algoritmi molto efficienti di riconoscimento e analisi delle frasi, esposti nel capitolo 4. Il capitolo si conclude inquadrandole le grammatiche libere all'interno della classica gerarchia dei modelli linguistici e computazionali proposti da Noam Chomsky.

### 2.5.1 Limiti dei linguaggi regolari

Le espressioni regolari, pur se adeguate in molti casi utili come le liste, non possono definire altri importanti costrutti. Un esempio sono le strutture a blocchi (o annidate) presenti in tanti linguaggi tecnici, schematizzate da

$$\begin{array}{c} \text{begin } \underbrace{\text{begin } \underbrace{\dots \text{end}}_{\text{begin } \dots \text{end}}}_{\text{begin } \dots \text{end}} \text{ end } \end{array}$$

*Esempio 2.29.* Semplici strutture a blocchi.

Con l'abbreviazione  $\{b, e\}$ , si consideri ora un caso limitato di strutture annidate, in cui le due lettere compaiono, nell'ordine  $b, e$ , lo stesso numero di volte:

$$L_1 = \{x \mid x = b^n e^n, n \geq 1\}$$

Si dimostrerà poi che il linguaggio non è regolare, ma fin d'ora non è difficile convincersi dell'impossibilità di scriverne una e.r.. Infatti, volendo produrre stringhe in cui le  $b$  precedono le  $e$ , o si scrive una definizione troppo permissiva, come la  $b^+ e^+$ , che ammette stringhe non valide come  $b^3 e^5$ ; o ci si accontenta di generare un campione finito del linguaggio, elencando le frasi fino a una certa lunghezza. D'altra parte, una e.r. che impone alle due lettere di comparire lo stesso numero di volte, come  $(be)^+$ , deriva anche  $bebe$ , violando la prescrizione sull'ordine dei caratteri.

Per definire questo e altri linguaggi, regolari o non, si espone il modello formale delle grammatiche dette *generative*.

### 2.5.2 Introduzione alle grammatiche libere

Allo scopo di definire un insieme di stringhe, si possono usare delle regole, che, attraverso ripetute applicazioni, permettono di generare tutte e soltanto le frasi del linguaggio. L'insieme delle regole costituisce la grammatica<sup>12</sup> o *sintassi* generativa.

<sup>12</sup>Il termine *grammatica* è altre volte inteso in un senso più generale di *sintassi*, come quando si aggiungono alle regole per la formazione delle frasi quelle per il calcolo dei significati. Si preciserà, quando necessario, il senso del termine.

**Esempio 2.30.** Palindromi

Una grammatica  $G$ , generante il linguaggio

$$L = \{uu^R \mid u \in \{a, b\}^*\} = \{\varepsilon, aa, bb, abba, baab, \dots, abbbb, \dots\}$$

delle stringhe di lunghezza pari, aventi simmetria speculare ( dette palindromi), è costituita da tre regole:

$$\begin{aligned} pal &\rightarrow \varepsilon \\ pal &\rightarrow a \text{ pal } a \\ pal &\rightarrow b \text{ pal } b \end{aligned}$$

La freccia ' $\rightarrow$ ' è un metasimbolo, riservato, per separare la parte sinistra dalla parte destra d'una regola.

Attraverso sostituzioni successive del simbolo (detto *nonterminale*) 'pal' con la parte destra di una regola, derivano altre stringhe, ad es.:

$$pal \Rightarrow a \text{ pal } a \Rightarrow ab \text{ pal } ba \Rightarrow abb \text{ pal } bba \Rightarrow \dots$$

Una catena di derivazione termina quando in essa tutti i simboli nonterminali sono eliminati; si è allora conclusa la generazione d'una frase. Continuando la derivazione si ha:

$$abb \text{ pal } bba \Rightarrow abbebb = abbbba$$

(Si osservi per inciso che il linguaggio dei palindromi non è regolare).

Si consideri ora un linguaggio un po' più complesso, una lista di palindromi separati dalla virgola, esemplificata dalla frase *abba, bbaabb, aa*. Il linguaggio è definito dalla seguente grammatica, che alle precedenti aggiunge due regole per generare la lista:

$$\begin{array}{ll} \text{lista} \rightarrow pal, \text{lista} & pal \rightarrow \varepsilon \\ \text{lista} \rightarrow pal & pal \rightarrow a \text{ pal } a \\ & pal \rightarrow b \text{ pal } b \end{array}$$

La prima regola dice: concatenando un palindromo, la virgola e una lista si ottiene ancora una lista. La seconda afferma che una lista può essere fatta d'un solo palindromo.

I simboli nonterminali sono ora due: *lista* e *pal*; il primo è detto l'*assioma* perché definisce il linguaggio desiderato, mentre il secondo definisce dei componenti (o sottostringhe *constituenti*) di esso, i palindromi.

**Esempio 2.31.** Metalinguaggio delle espressioni regolari.

Le espressioni regolari che definiscono i linguaggi di alfabeto terminale fissato  $\Sigma = \{a, b\}$  sono delle formule ossia delle stringhe di alfabeto  $\Sigma_{e.r.} = \{a, b, \cup, *, \emptyset, (,), \}$ , quindi esse stesse costituiscono un linguaggio.

In conformità con la definizione delle e.r. di p. 18, tale linguaggio è generato dalla sintassi  $G_{e.r.}$ :

1.  $\text{espr} \rightarrow \emptyset$
2.  $\text{espr} \rightarrow a$
3.  $\text{espr} \rightarrow b$
4.  $\text{espr} \rightarrow (\text{espr} \cup \text{espr})$
5.  $\text{espr} \rightarrow (\text{espr} \text{ espr})$
6.  $\text{espr} \rightarrow (\text{espr})^*$

dove la numerazione è per riferimento. Una derivazione, ottenuta applicando le regole sopra indicate, è:

$$\begin{aligned} \text{espr} &\Rightarrow_4 (\text{espr} \cup \text{espr}) \Rightarrow_5 ((\text{espr} \text{ espr}) \cup \text{espr}) \Rightarrow_2 ((a \text{ espr}) \cup \text{espr}) \Rightarrow_6 \\ &\Rightarrow ((a(\text{espr})^*) \cup \text{espr}) \Rightarrow_4 ((a((\text{espr} \cup \text{espr}))^*) \cup \text{espr}) \Rightarrow_2 \\ &\Rightarrow ((a((a \cup \text{espr}))^*) \cup \text{espr}) \Rightarrow_3 ((a((a \cup b))^*) \cup \text{espr}) \Rightarrow_3 ((a((a \cup b))^*) \cup b) \end{aligned}$$

Si osservi che la stringa ottenuta, interpretata come una e.r., definisce a sua volta il linguaggio di alfabeto  $\Sigma$ :

$$L(a(a \mid b)^* \mid b) = \{a, b, aa, ab, aaa, aba, \dots\}$$

L'insieme delle stringhe inizianti con la lettera  $a$  e la stringa  $b$ .

Attenzione: in questo esempio vi sono due livelli di linguaggio: la sintassi definisce delle stringhe che possono essere interpretate come definizioni di altri linguaggi, quelli della famiglia *REG*.

Si dice che la sintassi sta al livello metalinguistico, cioè sta sopra al livello linguistico; o anche che essa è una *metagrammatica*.

Per non confondere i due livelli si osservino gli alfabeti: al metalivello l'alfabeto è  $\Sigma_{e.r.} = \{a, b, \cup, ^*, \emptyset, (, )\}$ , mentre il linguaggio finale descritto ha l'alfabeto  $\Sigma = \{a, b\}$ , che è privo dei metasimboli.

Per chiarire la situazione, vale una analogia con le lingue naturali: in una grammatica del russo scritta in inglese, il testo contiene caratteri degli alfabeti latino e cirillico. L'inglese è il metalinguaggio mentre il russo è il linguaggio finale.

Si formalizza il modello della grammatica.

**Definizione 2.32.** Una grammatica (o sintassi) libera dal contesto (detta anche *non contestuale* o del tipo 2 o BNF<sup>13</sup>)  $G$  è definita da quattro entità:

1.  $V$ , alfabeto non terminale, è un insieme di simboli detti (metasimboli) nonterminali.
2.  $\Sigma$ , alfabeto terminale, è l'insieme dei caratteri con cui sono fatte le frasi.
3.  $P$ , insieme delle regole (o produzioni) sintattiche.
4.  $S \in V$ , un particolare nonterminale detto assioma.

Ogni regola di  $P$  è una coppia ordinata  $\langle X, \alpha \rangle$ , con  $X \in V$  e  $\alpha \in (V \cup \Sigma)^*$ . La regola  $\langle X, \alpha \rangle \in P$  sarà per chiarezza scritta nella forma:  $X \rightarrow \alpha$ .

Più regole

$$X \rightarrow \alpha_1, X \rightarrow \alpha_2, \dots, X \rightarrow \alpha_n$$

<sup>13</sup>Backus Normal Form, o anche Backus Naur Form, dal nome di John Backus e Peter Naur, tra i primi a usare queste grammatiche per i linguaggi programmativi.

a venti la stessa parte sinistra  $X$  possono essere raggruppate con l'abbreviazione

$$X \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n \quad \text{oppure} \quad X \rightarrow \alpha_1 \cup \alpha_2 \cup \dots \cup \alpha_n$$

Si dice che  $\alpha_1, \alpha_2, \dots, \alpha_n$  sono le (parti destre) alternative di  $X$ .

### 2.5.3 Rappresentazioni convenzionali delle grammatiche

Per evitare confusioni, i metasimboli ' $\rightarrow$ ', '|', ' $\cup$ ', ' $\epsilon$ ' non devono apparire tra i simboli terminali o non; inoltre gli alfabeti terminale e nonterminale devono essere disgiunti. In pratica, per distinguere i terminali dai nonterminali, sono in uso varie convenzioni, riassunte nella tabella:

| Nonterminali   | Terminali   | Esempio  |
|--|---|--|
| parole racchiuse tra parentesi angolari, ad es.: $< frase >$ , $< lista\_di\_frasi >$  | scritti senza particolari accorgimenti  | $< frase if > \rightarrow$<br>$if < cond > then < frase >$<br>$else < frase >$   |
| parole scritte senza particolari accorgimenti; non possono contenere spazi al loro interno, ad es.: <i>frase</i> , <i>lista_di_frasi</i> | scritti in neretto, in corsivo o racchiusi tra apici, ad es.:<br><br>a then<br>'a' 'then' | $frase\_if \rightarrow$<br>$if cond \ then \ frase \ else \ frase$<br>opp.<br>$frase\_if \rightarrow$<br>$'if' cond 'then' frase 'else' frase$ |
| lettere latine maiuscole; alfabeto terminale e nonterminale disgiunti  | scritti senza particolari accorgimenti  | $F \rightarrow if C \ then \ D \ else \ D$   |

La grammatica dell'esempio 2.30 con la prima convenzione diviene:

$$< frase > \rightarrow \epsilon, \quad < frase > \rightarrow a < frase > a, \quad < frase > \rightarrow b < frase > b$$

Le regole alternative possono essere raccolte:

$$< frase > \rightarrow \epsilon | a < frase > a | b < frase > b$$

Quando la grammatica raggiunge grandi dimensioni, dell'ordine del centinaio di regole, il documento che la descrive deve essere trattato in modo da facilitare il reperimento delle informazioni, le modifiche e i riferimenti incrociati. I nomi dei nonterminali devono essere autoesplicativi e le regole numerate

per riferimento. Per grammatiche di dimensioni ancora maggiori è opportuno suddividere in moduli l'insieme delle regole.

Al contrario, negli esempi più semplici e ridotti conviene usare la convenzione tre, prendendo due alfabeti disgiunti per i terminali e i nonterminali. Nel seguito, per immediatezza di lettura, si preferiranno le seguenti convenzioni:

- lettere latine iniziali minuscole  $\{a, b, \dots\}$  per i caratteri terminali
- lettere latine maiuscole  $\{A, B, \dots, Z\}$  per i simboli nonterminali
- lettere latine finali minuscole  $\{r, s, \dots, z\}$  per le stringhe di  $\Sigma^*$  (fatte di soli terminali)
- lettere minuscole dell'alfabeto greco  $\{\alpha, \beta, \dots\}$  per le stringhe di  $(V \cup \Sigma)^*$  (fatte di simboli terminali e non).

### Tipi speciali di regole

Nello studio delle grammatiche le regole sono classificate a seconda della loro forma. La classificazione serve a studiare le proprietà della grammatica e del linguaggio. Si elencano ora a scopo di riferimento alcuni tipi di regole con il loro nome tecnico. Per ogni tipo la tabella riporta lo schema delle regole con le convenzioni dette sopra:  $a, b$  denotano dei caratteri terminali,  $u, v, w$  denotano delle stringhe terminali (anche vuote),  $A, B, C$  denotano dei nonterminali,  $\alpha, \beta$  denotano delle stringhe (anche vuote) che possono contenere simboli sia terminali che nonterminali; inoltre la lettera  $\sigma$  indica delle stringhe di soli nonterminali.

La classificazione si basa soprattutto sulla forma della parte destra PD, tranne nelle classi ricorsive che considerano anche la parte sinistra PS. È omessa quella parte della regola che non è rilevante per la classificazione.

| Classe e Descrizione  | Esempi                             |
|---|------------------------------------|
| terminale: la PD contiene terminali o la stringa vuota  | $\rightarrow u \mid \epsilon$      |
| vuota (o nulla): la PD è vuota  | $\rightarrow \epsilon$             |
| iniziale: la PS è l'assioma   | $S \rightarrow$                    |
| ricorsiva: la PS compare nella PD   | $A \rightarrow \alpha A \beta$     |
| ricorsiva a sinistra: la PS è prefisso della PD   | $A \rightarrow A \beta$            |
| ricorsiva a destra: la PS è suffisso della PD   | $A \rightarrow \beta A$            |
| ricorsiva a destra e sinistra: congiunzione dei due casi precedenti   | $A \rightarrow A\beta A$           |
| copiatura o categorizzazione: la PD è un nontermi-nale singolo  | $A \rightarrow B$                  |
| lineare: al più un nonterminale nella PD  | $\rightarrow u B v \mid w$         |
| lineare a destra (tipo 3): come lineare, con nonterminale suffisso  | $\rightarrow u B \mid w$           |
| lineare a sinistra (tipo 3): come lineare, con nonterminale prefisso  | $\rightarrow B v \mid w$           |
| omogenea: $n$ nonterminali o un solo terminale  | $\rightarrow A_1 \dots A_n \mid a$ |
| normale di Chomsky (o omogenea di grado 2): due nonterminali o un solo terminale  | $\rightarrow B C \mid a$           |
| normale di Greibach: un terminale seguito da nonterminali   | $\rightarrow a \sigma \mid b$      |
| a operatori: due nonterminali separati da un terminale (operatore); più in generale, stringhe prive di nonterminali adiacenti | $\rightarrow A a B$                |

Le forme lineari a sinistra o a destra sono anche note come grammatiche del *tipo 3* della classificazione di Chomsky.

Si incontreranno nel libro le regole di molti dei tipi elencati; per i rimanenti, la conoscenza del nome potrà comunque essere utile.

Si vedrà che è possibile imporre qualcuna di queste forme alla grammatica del linguaggio, senza perdita di generalità. Le forme che godono di questa proprietà sono dette *normali*.

#### 2.5.4 Derivazioni e linguaggio generato

Si rivede e approfondisce il meccanismo di derivazione delle stringhe. Si consideri una stringa contenente un nonterminale,  $\beta = \eta A \delta$ , dove  $\eta$  e  $\delta$  sono stringhe qualsiasi anche vuote. Sia  $A \rightarrow \alpha$  una regola di  $G$ , e sia  $\gamma = \eta \alpha \delta$  la stringa ottenuta sostituendo in  $\beta$  la parte destra  $\alpha$  al posto del nonterminale  $A$ .

La relazione intercorrente tra le due stringhe è detta di *derivazione*. Si dice che  $\beta$  deriva  $\gamma$  per la grammatica  $G$ , e si scrive

$$\beta \xrightarrow{G} \gamma$$

o più semplicemente  $\beta \Rightarrow \gamma$  con  $G$  sottintesa. Si dice che la regola  $A \rightarrow \alpha$  è applicata nella derivazione, e che la stringa  $\alpha$  si riduce al nonterminale  $A$ . Si può considerare una catena di derivazioni di lunghezza  $n \geq 0$ :

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \dots \Rightarrow \beta_n$$

che si scrive

$$\beta_0 \xrightarrow{n} \beta_n$$

Per convenzione, con  $n = 0$ , ogni stringa  $\beta$  deriva se stessa:  $\beta \xrightarrow{0} \beta$ .

Si usano inoltre le scritture:

$$\beta_0 \xrightarrow{*} \beta_n \text{ (e risp. } \beta_0 \xrightarrow{+} \beta_n\text{)}$$

se la lunghezza della catena è  $n \geq 0$  (risp.  $n \geq 1$ ).

Il *linguaggio generato* o definito da  $G$  partendo dal nonterminale  $A$  è:

$$L_A(G) = \{x \in \Sigma^* \mid A \xrightarrow{*} x\}$$

esso è l'insieme delle stringhe terminali che in uno o più passi derivano da  $A$ . Se il nonterminale è l'assioma  $S$ , allora si ha il *linguaggio generato da G*:

$$L(G) = L_S(G) = \{x \in \Sigma^* \mid S \xrightarrow{*} x\}$$

Talvolta interessano anche le derivazioni che producono delle stringhe ancora contenenti simboli nonterminali. Una *forma di stringa* generata da  $G$  partendo dal nonterminale  $A \in V$ , è una stringa  $\alpha \in (V \cup \Sigma)^*$  tale che  $A \xrightarrow{*} \alpha$ . In particolare, se  $A$  è l'assioma, si ha una *forma di frase*.

Si può dire che una frase è una forma di frase priva di nonterminali.

*Esempio 2.33.* Struttura d'un libro.

La grammatica genera la struttura d'un libro: esso contiene un frontespizio ( $f$ ) e una serie (derivata dal nonterminale  $A$ ) di uno o più capitoli, ognuno dei quali inizia con il titolo ( $t$ ) del capitolo e contiene una serie (derivata da  $B$ ) d'una o più linee ( $l$ ). Ecco la grammatica  $G_l$

$$\begin{aligned} S &\rightarrow fA \\ A &\rightarrow AtB \mid tB \\ B &\rightarrow lB \quad | \quad l \end{aligned}$$

e alcune derivazioni. Partendo da  $A$  si genera la forma  $tBtB$  e la stringa  $tltl \in L_A(G_l)$ ; partendo dall'assioma  $S$  si generano le forme di frase  $fAtlB, ftBtB$  e la frase  $ftlfltl$ . Il linguaggio generato partendo da  $B$ , vale  $L_B(G_l) = l^+$ ; il linguaggio  $L(G_l)$  generato da  $G_l$  è definito dalla espressione regolare  $f(tl^+)^+$ . Questo linguaggio è regolare, essendo il noto paradigma delle liste a più livelli.

Un linguaggio è *libero (dal contesto)* se esiste una grammatica libera che lo genera. La famiglia dei linguaggi liberi è chiamata *LIB*.

Due grammatiche  $G$  e  $G'$  sono *equivalenti* se generano lo stesso linguaggio, ossia se  $L(G) = L(G')$ .

*Esempio 2.34.* È banalmente equivalente alla  $G_l$  dell'es. 2.33 la seguente  $G_{l2}$ :

$$\begin{array}{l} S \rightarrow fX \\ X \rightarrow XtY \mid tY \\ Y \rightarrow lY \quad | \quad l \end{array}$$

che si differenzia soltanto per i nomi dei nonterminali. Anche la seguente  $G_{l3}$

$$\begin{array}{l} S \rightarrow fA \\ A \rightarrow AtB \mid tB \\ B \rightarrow Bl \quad | \quad l \end{array}$$

è equivalente. Essa differisce solo nella terza riga che definisce  $B$  con la regola ricorsiva a sinistra, non a destra come in  $G_l$ . Chiaramente le derivazioni di lunghezza  $n \geq 1$

$$B \xrightarrow[G_l]{n} l^n \quad \text{e} \quad B \xrightarrow[G_{l3}]{n} l^n$$

generano lo stesso linguaggio  $L_B = l^+$ .

### 2.5.5 Grammatiche erronee e regole inutili

Quando si scrive una grammatica occorre assicurarsi che tutti i nonterminali siano definiti e che ognuno di essi effettivamente contribuisca alla produzione del linguaggio. Infatti certe regole d'una grammatica potrebbero non essere produttive.

Una grammatica  $G$  è *pulita* (o *ridotta*) se valgono le condizioni:

1. ogni nonterminale  $A$  è *raggiungibile* dall'assioma, ossia esiste una derivazione  $S \xrightarrow{*} \alpha A \beta$ ;
2. ogni nonterminale  $A$  è *ben definito*, ossia genera un linguaggio non vuoto,  $L_A(G) \neq \emptyset$ .

Si descrive un algoritmo per ripulire una grammatica.

#### Pulizia della grammatica

L'algoritmo opera in due fasi, la prima trova i nonterminali non definiti, la seconda quelli non raggiungibili. Infine si eliminano le regole contenenti nonterminali dei due tipi.

Fase 1. Calcola l'insieme  $DEF \subseteq V$  dei nonterminali ben definiti.

- L'insieme  $DEF$  è inizializzato con i nonterminali delle regole terminali, quelle che hanno come parte destra una stringa terminale:

$$DEF := \{A \mid (A \rightarrow u) \in P, \text{ con } u \in \Sigma^*\}$$

Si applica poi la seguente trasformazione fino alla convergenza:

$$DEF := DEF \cup \{B \mid (B \rightarrow D_1 D_2 \dots D_n) \in P\}$$

dove ogni  $D_i$  è un terminale o un simbolo nonterminale appartenente a  $DEF$ .

A ogni iterazione sono date due possibilità:

- si scoprono nuovi nonterminali aventi come parte destra una stringa di elementi tutti definiti o terminali, o
- si termina.

I nonterminali appartenenti a  $V \setminus DEF$  sono indefiniti e vanno eliminati.

Fase 2. Il calcolo dei nonterminali raggiungibili da  $S$  è facilmente ricondotto all'esistenza d'un cammino nel grafo della seguente relazione binaria tra nonterminali, detta *produce*.

$$A \xrightarrow{\text{produce}} B$$

letta  $A$  produce  $B$ , se, e solo se, esiste la regola  $A \rightarrow \alpha B \beta$ , dove  $A, B$  sono nonterminali e  $\alpha, \beta$  sono stringhe qualsiasi.

È ovvio che  $C$  è raggiungibile da  $S$  se, e solo se, esiste nel grafo della relazione un cammino da  $S$  a  $C$ . I nonterminali irraggiungibili sono allora il complemento rispetto a  $V$ .

I nonterminali irraggiungibili vanno eliminati, perché non generano alcuna frase.

Spesso si aggiunge il seguente requisito alle due condizioni di pulizia precedenti:

- $G$  non consente *derivazioni circolari* del tipo  $A \stackrel{+}{\Rightarrow} A$ .

Infatti tali derivazioni sono inessenziali, perché, se la stringa  $x$  fosse ottenuta con la derivazione  $A \Rightarrow A \Rightarrow x$ , essa sarebbe generata anche dalla derivazione  $A \Rightarrow x$ .

Inoltre tali derivazioni causano l'ambiguità (come si vedrà).

Nel libro si suppone sempre che le grammatiche siano pulite e prive di circolarità.

*Esempio 2.35.* Esempi non puliti.

- La grammatica con le regole  $\{S \rightarrow aASb, A \rightarrow b\}$  non genera alcuna frase.
- La grammatica  $G$  con le regole  $\{S \rightarrow a, A \rightarrow b\}$  ha il nonterminale  $A$ , irraggiungibile dall'assioma, quindi inutile; lo stesso linguaggio  $L(G)$  è generato dalla grammatica pulita  $\{S \rightarrow a\}$ .
- Derivazione circolare:  
La grammatica con le regole  $\{S \rightarrow aASb \mid A, A \rightarrow S \mid c\}$  offre la derivazione circolare  $S \Rightarrow A \Rightarrow S$ , inutile. La grammatica  $\{S \Rightarrow aSSb \mid c\}$  è equivalente.
- Si noti che la circolarità può nascere per effetto d'una regola vuota, come ad es. nel frammento di grammatica:

$$X \rightarrow XY \mid \dots \quad Y \rightarrow \varepsilon \mid \dots$$

Anche se ripulita una grammatica può ancora presentare delle regole ridondanti, come mostra il seguente caso.

*Esempio 2.36.* Regole doppie.

- |                             |                      |
|-----------------------------|----------------------|
| 1. $S \rightarrow aASb$     | 4. $A \rightarrow c$ |
| 2. $S \rightarrow aBSb$     | 5. $B \rightarrow c$ |
| 3. $S \rightarrow \epsilon$ |                      |

Una delle coppie (1,4) e (2,5), che generano esattamente le stesse frasi, può essere cancellata.

### 2.5.6 Ricorsione delle regole e infinitezza del linguaggio

Una caratteristica di quasi tutti i linguaggi artificiali e naturali è quella di essere infiniti. Conviene ora esaminare da che cosa dipende la capacità d'una grammatica di generare dei linguaggi infiniti. Per produrre un numero illimitato di frasi è ovviamente necessario che le regole permettano la derivazione di frasi di lunghezza illimitata. Affinché ciò possa avvenire, la grammatica deve avere la proprietà di ricorsione, che ora si definisce.

Una derivazione a  $n \geq 1$  passi  $A \xrightarrow{n} xAy$  è detta *ricorsiva* (*immediatamente se  $n = 1$* ), e il nonterminale  $A$  è pure detto *ricorsivo*.

Se poi  $x$  (risp.  $y$ ) è vuota, la ricorsione è detta *sinistra* (risp. *destra*).

*Proprietà 2.37.* Condizione necessaria e sufficiente perché sia infinito il linguaggio  $L(G)$ , dove  $G$  è una grammatica pulita e priva di derivazioni circolari, è che  $G$  permetta delle derivazioni ricorsive.

La condizione è necessaria: infatti è facile vedere che, se non vi fossero derivazioni ricorsive, ogni derivazione avrebbe una lunghezza limitata, quindi ogni frase sarebbe limitata in lunghezza, e  $L(G)$  sarebbe finito.

La condizione è sufficiente: la presenza della ricorsione  $A \xrightarrow{n} xAy$  permette la derivazione  $A \xrightarrow{\pm} x^m A y^m$ , per  $m \geq 1$  arbitrario, in cui  $x$  e  $y$  non sono entrambe vuote, essendo  $G$  priva di circolarità per ipotesi. Ma essendo  $G$  pulita,  $A$  è raggiungibile dall'assioma con una derivazione  $S \xrightarrow{*} uAv$ , e da  $A$  deriva almeno una stringa terminale  $A \xrightarrow{\pm} w$ . In conclusione esistono le derivazioni

$$S \xrightarrow{*} uAv \xrightarrow{\pm} ux^m A y^m v \xrightarrow{\pm} ux^m w y^m v, \quad (m \geq 1)$$

che generano un linguaggio infinito.

Per decidere se una grammatica ha delle ricorsioni basta esaminare la relazione binaria *produce* di p. 38: la grammatica è priva di ricorsioni se, e solo se, il grafo della relazione è privo di cicli.

Si mostrano due grammatiche, che generano un linguaggio finito e infinito.

*Esempio 2.38.* Linguaggio finito.

$$S \rightarrow aBc \quad B \rightarrow ab \mid Ca \quad C \rightarrow c$$

Questa grammatica è priva di ricorsioni e presenta soltanto due derivazioni terminali che generano il linguaggio finito  $\{aab, acac\}$ .

Il prossimo esempio è uno dei paradigmi più diffusi e replicati nei linguaggi artificiali. Sarà più volte ripreso per illustrare vari aspetti della materia.

### Esempio 2.39. Espressioni aritmetiche.

La grammatica

$$G = (\{E, T, F\}, \{i, +, *\}, \{\}, P, E)$$

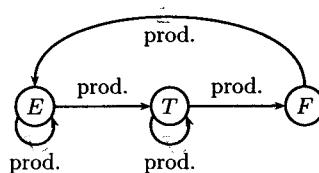
ha le regole

$$E \rightarrow E + T \mid T \quad T \rightarrow T * F \mid F \quad F \rightarrow (E) \mid i$$

e genera il linguaggio

$$L(G) = \{i, i + i + i, i * i, (i + i) * i, \dots\}$$

le cui frasi sono le espressioni aritmetiche nella lettera  $i$ , con gli operatori di somma e prodotto, e eventualmente le parentesi tonde. Il nonterminale  $F$  (fattore) ha una ricorsione indiretta, mentre  $T$  (termine) e  $E$  (espressione) hanno anche ricorsioni immediate del tipo sinistro: ciò appare chiaramente dal grafo della relazione *produce*:



La grammatica è pulita e non circolare, quindi il linguaggio generato è infinito.

### 2.5.7 Alberi sintattici e derivazioni canoniche

È utile rappresentare graficamente il processo di derivazione per mezzo d'un albero sintattico. Si ricordi che un *albero* è un grafo orientato e ordinato privo di cicli, tale che per ogni coppia di nodi esiste un solo cammino (non necessariamente orientato) che li congiunge. Un arco (orientato)  $\langle N_1 \rightarrow N_2 \rangle$  definisce la relazione *(padre, figlio)*, che di solito è rappresentata dall'alto verso il basso come in un albero genealogico. I figli sono ordinati da sinistra a destra. Il *grado* d'un nodo è il numero dei suoi figli. In un albero esiste un solo nodo privo di padre, la *radice*.

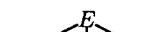
Preso un nodo interno  $N$ , il *sottoalbero* di radice  $N$  è l'albero avente  $N$  come radice e comprendente tutti figli di  $N$ , i figli dei figli, ecc., cioè tutti i *discendenti* di  $N$ . I nodi privi di figli sono detti *foglie* o *nodi terminali*. La sequenza delle foglie lette da sinistra a destra (con riferimento all'ordinamento) è la *frontiera*.

Un *albero sintattico* ha come radice l'assioma e come frontiera una frase generata. Esso viene costruito disegnando per ogni regola  $A_0 \rightarrow A_1A_2\dots A_r, r \geq 1$  usata nella derivazione, l'albero avente  $A_0$  come radice e i figli  $A_1A_2\dots A_r$ , dove i figli sono terminali o nonterminali. Se la regola è  $A_0 \rightarrow \epsilon$ , si disegna un solo figlio, etichettato con la epsilon. Tali alberi sono poi incollati insieme, facendo combaciare ogni figlio nonterminale, mettiamo  $A_i$ , con il nodo padre della regola, avente lo stesso  $A_i$  come parte sinistra, usata per espanderlo nella derivazione.

*Esempio 2.40.* Albero sintattico.

La grammatica delle espressioni aritmetiche è qui ripetuta numerando le regole per riferimento e disegnando l'alberi associati a ogni regola.

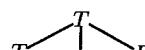
$$1. E \rightarrow E + T$$



$$2. E \rightarrow T$$



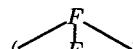
$$3. T \rightarrow T * F$$



$$4. T \rightarrow F$$



$$5. F \rightarrow (E)$$



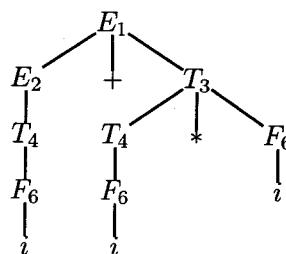
$$6. F \rightarrow i$$



La derivazione:

$$E \xrightarrow{1} E+T \xrightarrow{2} T+T \xrightarrow{4} F+T \xrightarrow{6} i+T \xrightarrow{3} i+T*F \xrightarrow{4} i+F*F \xrightarrow{6} i+i*F \xrightarrow{6} i+i*i \quad (2.2)$$

ha l'albero sintattico seguente:



Per riferimento si indicano i numeri delle regole applicate. Si noti che lo stesso albero rappresenta anche la derivazione:

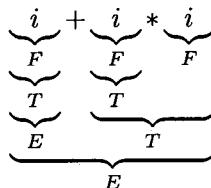
$$E \xrightarrow{1} E+T \xrightarrow{3} E+T*F \xrightarrow{6} E+T*i \xrightarrow{4} E+F*i \xrightarrow{6} E+i*i \xrightarrow{2} T+i*i \xrightarrow{4} F+i*i \xrightarrow{6} i+i*i \quad (2.3)$$

e tante altre che solo differiscono nell'ordine con cui si applicano le regole. La derivazione 2.2 è detta *sinistra*, la 2.3 è detta *destra*.

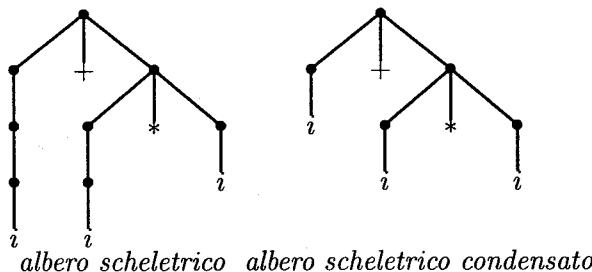
L'albero sintattico d'una frase  $x$  può anche essere codificato in forma di testo, racchiudendo ogni sottoalbero tra una coppia di parentesi<sup>14</sup>, etichettata con il nome del simbolo nonterminale. All'albero precedente corrisponde la *espressione parentetica*

$$[[[i]_F]_T]_E + [[[i]_F]_T * [i]_F]_T]_E$$

anche nella variante grafica



Se dell'albero interessa solo la frontiera e la struttura, si cancellano i simboli nonterminali ottenendo un *albero scheletrico* (a sinistra) o un *albero scheletrico condensato* (a destra):



o la corrispondente stringa a parentesi:

$$[[[i]]] + [[[i]] * [i]]]$$

Nell'*albero scheletrico condensato* (a destra) si fondono i nodi interni che stanno su un cammino privo di biforazioni (quelli cioè ottenuti con regole di ricopertura). La corrispondente frase parantetizzata è allora

$$[[i] + [[i] * [i]]]$$

Talvolta, per enfatizzare il fatto che una grammatica assegna alle frasi del linguaggio una precisa struttura, la grammatica è vista come la definizione formale d'un insieme di alberi, ossia d'un linguaggio di alberi e non di semplici stringhe.<sup>15</sup>

<sup>14</sup>Si suppone che le parentesi quadre non appartengano all'alfabeto terminale.

<sup>15</sup>Un riferimento per la teoria dei linguaggi di alberi è [20].

### Derivazioni sinistre e destre

Una derivazione

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \dots \Rightarrow \beta_p$$

dove

$$\beta_i = \eta_i A_i \delta_i \text{ e } \beta_{i+1} = \eta_i \alpha_i \delta_i$$

è detta (canonica) *destra* (risp. *sinistra*) se, per ogni  $0 \leq i \leq p-1$ , è  $\delta_i \in \Sigma^*$  (risp.  $\eta_i \in \Sigma^*$ ).

In altre parole, in una derivazione destra (sinistra) a ogni passo si espande il nonterminale posto più a destra (sinistra). Si usa porre sotto la freccia una ‘d’ o una ‘s’ per indicare una derivazione destra o sinistra.

Naturalmente si può avere una derivazione che non è né destra né sinistra, o perché espande un nonterminale che non è a destra o a sinistra, o perché a un passo procede in modo destro, e a un altro in modo sinistro.

Riprendendo l'esempio precedente la derivazione 2.2 è sinistra e può essere indicata da  $E \xrightarrow[s]{+} i + i * i$ . La derivazione 2.3 è destra, mentre la derivazione

$$\begin{aligned} E &\xrightarrow[s,d]{} E + T \xrightarrow[d]{} E + T * F \xrightarrow[s]{} T + T * F \xrightarrow{} T + F * F \xrightarrow[d]{} \\ &T + F * i \xrightarrow[s]{} F + F * i \xrightarrow[d]{} F + i * i \xrightarrow[d]{} i + i * i \end{aligned} \tag{2.4}$$

non è né destra né sinistra. Tutte e tre le derivazioni sono rappresentate dallo stesso albero.

L'esempio illustra una proprietà essenziale delle grammatiche libere.

*Proprietà 2.41.* Ogni frase d'una grammatica libera può essere generata mediante una derivazione sinistra (oppure destra).

Di conseguenza possiamo sostituire le derivazioni sinistre (oppure destre) nella definizione (p. 36) del linguaggio generato da una grammatica.

Altri modelli più complessi, come le grammatiche dipendenti dal contesto, non hanno questa proprietà, di cui vedremo l'importanza per gli algoritmi di analisi sintattica, dove permette di organizzare nel modo più opportuno l'ordine di costruzione della derivazione d'una frase.

### 2.5.8 Linguaggi a parentesi

Nei linguaggi artificiali si incontrano frequentemente delle strutture a parentesi (o annidate), caratterizzate dalla presenza di coppie di elementi che aprono e chiudono un inciso (o stringa subordinata); all'interno di una coppia di parentesi possono poi aprirsi altri incisi.

La rappresentazione delle parentesi cambia da linguaggio a linguaggio: così in Pascal le strutture a blocchi stanno entro le parentesi ‘begin’ ... ‘end’, mentre nel linguaggio C si usano per lo stesso scopo le parentesi graffe. Le parole ‘begin’ e ‘end’ e le parentesi graffe sono dette *marche di apertura e di chiusura*.

Un uso molto massiccio delle strutture a parentesi viene fatto nei documenti, destinati alla Rete, scritti nel formato di marcatura XML, che prevede un numero illimitato di diverse marche di apertura e chiusura, ad esempio la coppia `<title>...</title>` è usata per delimitare il titolo di un documento. Similmente nel linguaggio LaTeX, in cui questo libro è stato composto, una formula matematica è racchiusa tra le marche `\begin{equation}... \end{equation}`. Si osservi che un inciso può essere contenuto (si dice *autoincluso*) in un altro inciso dello stesso genere. L'autoinclusione è potenzialmente illimitata nei linguaggi artificiali, mentre nelle lingue naturali se ne fa uso moderato, per non rendere difficile la comprensione a causa dell'interruzione nel filo del discorso. Ecco un esempio di una complessa frase tedesca<sup>16</sup> con tre clausole relative incassate:

*der Mann der die Frau die das Kind das die Katze füttert sieht liebt schläft*

Astraendo dalla particolare codifica delle marche, il paradigma a parentesi è noto come *linguaggio di Dyck*. L'alfabeto terminale ha una o più coppie di parentesi aperte e chiuse. Ad es. con l'alfabeto terminale  $\Sigma = \{', ', '[, ']\}$  una frase è `[](([]))`

Le frasi di Dyck sono caratterizzate dalla cosiddetta *regola di cancellazione*. Si applichi ripetutamente alla stringa data la trasformazione che sostituisce la stringa vuota a una coppia di parentesi concordi e adiacenti:

$$[ ] \Rightarrow \epsilon \quad ( ) \Rightarrow \epsilon$$

Al termine, cioè quando la trasformazione non è più applicabile, se la stringa è vuota, la stringa originaria è valida, altrimenti non lo è.

*Esempio 2.42.* Linguaggio di Dyck.

Per facilitare la lettura, conviene sostituire alle parentesi aperte le lettere  $a, b, \dots$  e a quelle chiuse le corrispondenti lettere apostrofate  $a', b', \dots$

Preso ad es. l'alfabeto  $\Sigma = \{a, a', b, b'\}$ , il linguaggio di Dyck è generato dalla grammatica:

$$S \rightarrow aSa'S \mid bSb'S \mid \epsilon$$

Si osservi che le prime due alternative non sono regole di tipo lineare; infatti per generare questo linguaggio sono essenziali le regole non lineari.

Si confronti il linguaggio precedente con  $L_1$  dell'es. 2.29 di p. 30. Quest'ultimo, pur di sostituire all'alfabeto  $\{b, e\}$  l'alfabeto  $\{a, a'\}$ , è incluso strettamente nel linguaggio di Dyck, perché  $L_1$  non ammette le stringhe contenenti più d'un nido di parentesi, come

$$\overbrace{aa} \quad \overbrace{aa'} \quad \overbrace{a' a} \quad \overbrace{aa'} \quad \overbrace{a' a' a'}$$

le quali necessariamente hanno un albero sintattico ramificato.

---

<sup>16</sup>L'uomo che ama la donna (la quale vede il bimbo (che nutre il gatto)) dorme.

Un altro modo per generare strutture parentetiche è di imporre che ogni regola della grammatica sia parentetizzata.

#### Definizione 2.43. Grammatica parentesizzata

Sia  $G = (V, \Sigma, P, S)$  una grammatica il cui alfabeto  $\Sigma$  non contiene le parentesi. La grammatica parentesizzata  $G_p$  ha l'alfabeto terminale  $\Sigma \cup \{(')\}$  e le regole

$$A \rightarrow (\alpha) \text{ dove } A \rightarrow \alpha \text{ è una regola di } G$$

*La grammatica è parentesizzata con segni distinti se ogni regola ha la forma*

$$A \rightarrow ({}_A \alpha) {}_A \quad B \rightarrow ({}_B \alpha) {}_B$$

*dove  $({}_A \alpha) {}_A$  sono parentesi contrassegnate con il nome del nonterminale.*

Ogni frase del linguaggio definito da tali grammatiche ha una struttura a parentesi.

#### Esempio 2.44. Grammatica a parentesi.

La versione parentesizzata della grammatica delle liste di palindromi (p. 31) è

$$\begin{array}{ll} \text{lista} \rightarrow (\text{pal}, \text{lista}) & \text{pal} \rightarrow () \\ \text{lista} \rightarrow (\text{pal}) & \text{pal} \rightarrow (a \text{ pal } a) \\ & \text{pal} \rightarrow (b \text{ pal } b) \end{array}$$

Alla frase originale *aa* corrisponde la frase parentesizzata

$$(((a \ ( \ ) a)))$$

Un effetto della parentesizzazione è di facilitare il controllo che una stringa appartenga al linguaggio parentesizzato (come si vedrà nel cap. 4).



#### 2.5.9 Composizione regolare di linguaggi liberi

Le operazioni basilari dei linguaggi regolari, unione, concatenamento e stella, possono essere applicate a linguaggi liberi, e producono ancora linguaggi liberi, come facilmente si vedrà.

Siano  $G_1 = (\Sigma_1, V_1, P_1, S_1)$  e  $G_2 = (\Sigma_2, V_2, P_2, S_2)$  le grammatiche che definiscono i linguaggi  $L_1$  e  $L_2$ . È necessaria l'ipotesi che i due alfabeti nonterminali siano disgiunti,  $V_1 \cap V_2 = \emptyset$ . Inoltre si suppone che il simbolo  $S$ , che sarà il nuovo assioma, non compaia in nessuna delle due grammatiche,  $S \notin (V_1 \cup V_2)$ .

**Unione:** L'unione  $L_1 \cup L_2$  dei linguaggi è generata dalla grammatica ottenuta unendo le regole delle due grammatiche, e aggiungendo le regole iniziali  $S \rightarrow S_1 \mid S_2$ .

In formule, la grammatica è

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_1 \cup V_2, \{S \rightarrow S_1 \mid S_2\} \cup P_1 \cup P_2, S)$$

**Concatenamento:** Il concatenamento  $L_1L_2$  dei linguaggi è generato dalla grammatica ottenuta unendo le regole delle due grammatiche, e aggiungendo la regola iniziale  $S \rightarrow S_1S_2$ .

In formule, la grammatica è

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_1 \cup V_2, \{S \rightarrow S_1S_2\} \cup P_1 \cup P_2, S)$$

**Stella:** La grammatica  $G$  del linguaggio  $(L_1)^*$  è ottenuta aggiungendo a  $G_1$  le regole  $S \rightarrow SS_1 \mid \epsilon$

**Croce:** Per l'identità  $X^+ = XX^*$  si può costruire la grammatica della croce, applicando le costruzioni del concatenamento e della stella, ma è più semplice scriverla direttamente.

La grammatica  $G$  del linguaggio  $(L_1)^+$  è ottenuta aggiungendo a  $G_1$  le regole  $S \rightarrow SS_1 \mid S_1$

In sintesi vale la seguente proprietà.

**Proprietà 2.45.** La famiglia  $LIB$  dei linguaggi liberi è chiusa rispetto alle operazioni di unione, concatenamento, stella e croce.

**Esempio 2.46.** Unione di linguaggi.

Il linguaggio

$$L = \{a^i b^i c^* \mid i \geq 0\} \cup \{a^* b^i c^i \mid i \geq 0\} = L_1 \cup L_2$$

contiene frasi della forma  $a^i b^j c^k$  con  $i = j \vee j = k$ , quali

$$a^5 b^5 c^2, a^5 b^5 c^5, b^5 c^5$$

È facile scrivere le regole dei linguaggi componenti:

| $G_1$                             | $G_2$                             |
|-----------------------------------|-----------------------------------|
| $S_1 \rightarrow XC$              | $S_2 \rightarrow AY$              |
| $X \rightarrow aXb \mid \epsilon$ | $Y \rightarrow bYc \mid \epsilon$ |
| $C \rightarrow cC \mid \epsilon$  | $A \rightarrow aA \mid \epsilon$  |

alle quale si aggiungono le alternative  $S \rightarrow S_1 \mid S_2$ , che lanciano le derivazioni verso l'uno o l'altro linguaggio.

Si sottolinea che, se cade l'ipotesi che i due alfabeti nonterminali siano disgiunti, la costruzione produce una grammatica che non genera l'unione ma un linguaggio più ampio. Così, se la precedente  $G_2$  fosse sostituita dalla grammatica banalmente equivalente  $G''$ :

$$S'' \rightarrow AX \quad X \rightarrow bXc \mid \epsilon \quad A \rightarrow aA \mid \epsilon$$

la grammatica aventi le regole  $\{S \rightarrow S_1 \mid S''\} \cup P_1 \cup P''$  offrirebbe delle derivazioni ibride, che usano regole di entrambe le grammatiche componenti e producono frasi estranee all'unione, come ad es.  $abcbc$ .

La proprietà 2.45 è comune alle famiglie  $REG$  e  $LIB$ , ma soltanto la prima è chiusa rispetto alle operazioni di intersezione e di complemento, come si vedrà.

### *Grammatica del linguaggio speculare*

Continuando lo studio dell'effetto delle operazioni sui linguaggi della famiglia *LIB*, è facile vedere che la famiglia *LIB* (come *REG*) è chiusa rispetto alla trasformazione speculare del linguaggio.

Data la grammatica d'un linguaggio, è semplice ottenere quella del linguaggio speculare: basta invertire specularmente la parte destra di ogni regola.

#### 2.5.10 Ambiguità

L'ambiguità, un fenomeno linguistico comune nel linguaggio naturale, si manifesta quando una frase presenta due o più significati. L'ambiguità può essere di natura semantica o sintattica. È semantica nella frase *la pesca fu bella*, in cui *pesca* è sempre un sostantivo, ma con due sensi distinti (polisemia): il frutto oppure il lavoro del pescatore.

Diversamente un esempio di ambiguità sintattica (o strutturale) si vede nella frase inglese *half baked chicken*, traducibile in *pollo mezzo cotto* oppure in *mezzo pollo cotto*, a seconda che la sua struttura sintattica sia *[[half baked] chicken]* oppure *[half [baked chicken]]*. Si vede che la presenza di due strutture sintattiche alternative induce due interpretazioni della frase. L'ambiguità può causare malintesi nella comunicazione.

Anche nei linguaggi artificiali si manifestano, seppure in misura assai minore, i fenomeni di ambiguità, che sono generalmente considerati negativi e vanno controllati.

Una *frase x* del linguaggio definito dalla grammatica *G* è (sintatticamente) *ambigua*, se essa è generata con due alberi sintattici distinti. Anche la grammatica *G* è allora detta *ambigua*.

*Esempio 2.47.* Per generare il linguaggio delle espressioni aritmetiche dell'esempio 2.39 di p. 40, si scrive un'altra grammatica *G'* equivalente alla precedente:

$$E \rightarrow E + E \mid E * E \mid (E) \mid i$$

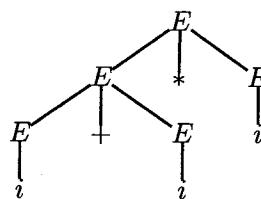
Le derivazioni sinistre

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow i + E * E \Rightarrow i + i * E \Rightarrow i + i * i \quad (2.5)$$

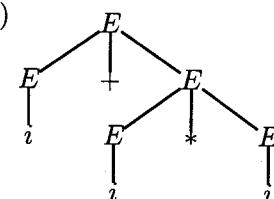
$$E \Rightarrow E + E \Rightarrow i + E \Rightarrow i + E * E \Rightarrow i + i * E \Rightarrow i + i * i \quad (2.6)$$

generano la stessa frase, con due alberi diversi:

(2.5)



(2.6)



La frase  $i + i * i$  è dunque ambigua.

Se si considera anche il significato delle espressioni aritmetiche, il primo albero assegna alla frase l'interpretazione  $(i+i)*i$ , il secondo l'interpretazione  $i+(i*i)$ ; la seconda è presumibilmente preferibile, perché si adegua alla precedenza tradizionale del prodotto rispetto alla somma.

Anche la frase  $i + i + i$  è ambigua, con due alberi che differiscono per l'ordine in cui vengono associate le sottoespressioni, da sinistra a destra o da destra a sinistra.

La grammatica  $G'$  è dunque ambigua.

L'altro inconveniente di questa grammatica è che non impone la tradizionale precedenza del prodotto sulla somma.

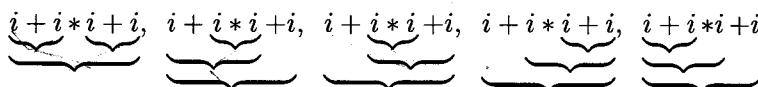
Invece la grammatica  $G$  dell'esempio 2.39 genera le stesse frasi con una, e una sola, derivazione sinistra, dunque tali frasi non sono ambigue. Si potrebbe vedere che nessuna frase è ambigua per la grammatica  $G$ , che quindi risulta inambigua.

Si noti d'altra parte che la nuova grammatica  $G'$  è più piccola della vecchia  $G$ : manifestazione d'una proprietà frequente delle grammatiche ambigue, quella di essere più piccole delle grammatiche non ambigue equivalenti. Talvolta, in circostanze particolari, il ridotto ingombro delle grammatiche ambigue può essere sfruttato per ridurre le dimensioni del compilatore; ma in generale la maggiore semplicità non compensa l'equivocità di tali grammatiche.

Il *grado di ambiguità* d'una frase  $x$  del linguaggio  $L(G)$  è il numero di alberi distinti con cui essa è generata da  $G$ . Per una grammatica, il grado di ambiguità è il massimo tra i gradi delle frasi. Il grado di ambiguità d'una grammatica può risultare illimitato.

*Esempio 2.48.* (Es. 2.47 continuato.)

Il grado di ambiguità vale 2 per la frase  $i + i + i$ ; vale 5 per  $i + i * i + i$  che ha gli alberi scheletro sotto schematizzati:



Generalizzando, si vede facilmente che, all'allungarsi delle frasi considerate, il grado di ambiguità cresce senza limite.

Un problema pratico importante è quello di accertare l'inambiguità d'una grammatica data. Come altri problemi apparentemente semplici della teoria dei linguaggi, esso non è decidibile<sup>17</sup>. Ciò significa che non esiste un algoritmo generale che, data una generica grammatica libera, si fermi dopo un numero finito di passi con la risposta: è (non è) ambigua. L'impossibilità nasce dal fatto che la procedura che ricerca le eventuali ambiguità sarebbe trascinata a esaminare derivazioni sempre più lunghe. Questo in generale: ma l'ambiguità di una particolare grammatica può essere esaminata, come un problema a sé,

<sup>17</sup> La dimostrazione si può trovare in [27].

caso per caso, facendo uso di ragionamenti induttivi.

In pratica si procede in due modi. Si esamina un piccolo numero di frasi sulle quali si fa un test di ambiguità, consistente nel costruire i loro alberi sintattici in tutti i modi possibili.

Superato il test, si verifica se la grammatica appartiene a una delle sottoclassi deterministiche delle grammatiche libere. Tali sottoclassi saranno ampiamente studiate nel capitolo 4, per costruire gli analizzatori sintattici veloci. Tale verifica è effettivamente calcolabile e garantisce l'assenza di ambiguità.

Ma meglio ancora è evitare l'ambiguità nel momento stesso in cui la grammatica è progettata, schivando certi errori che ora si mostreranno.

### 2.5.11 Catalogo di forme ambigue e rimedi

L'ambiguità sintattica si manifesta quando la sintassi assegna due o più strutture alla stessa frase. Se tali strutture sono sensate (semanticamente corrette), l'ambiguità dà luogo a interpretazioni diverse della frase.

Nei linguaggi naturali abbondano le ambiguità, ma la comunicazione non ne soffre troppo, grazie alla presenza d'un contesto non linguistico (gesti, intonazione della frase, presupposti e attese, ecc.) che chiarisce quale sia l'interpretazione intesa dal parlante o dallo scrittore.

Nei linguaggi artificiali l'ambiguità non può essere tollerata, perché la macchina, a differenza dell'uomo, non riesce a sfruttare bene il contesto. Essa è quasi sempre il sintomo d'un errore di progetto, in quanto una frase ambigua può rendere non univoco il comportamento del traduttore o dell'interprete del linguaggio.

Si propone di mostrare come si possano evitare molte ambiguità grammaticali con opportuni accorgimenti nella stesura delle regole o con piccole modifiche al linguaggio.

#### Ambiguità nella ricorsione bilaterale

Un simbolo nonterminale  $A$  è ricorsivo bilaterale, se esso è ricorsivo a sinistra (derivazione  $A \xrightarrow{*} A\gamma$ ) e a destra (derivazione  $A \xrightarrow{*} \beta A$ ). Si distinguono le situazioni in cui la ricorsione bilaterale è prodotta dalla stessa regola (o derivazione) oppure da regole diverse.

*Esempio 2.49.* Ricorsioni sinistra e destra nella stessa regola.

La grammatica  $G_1$ :

$$E \rightarrow E + E \mid i$$

genera la stringa  $i+i+i$  in due modi diversi, denotati dalle derivazioni sinistre:

$$E \Rightarrow E + E \Rightarrow E + E + E \Rightarrow i + E + E \Rightarrow i + i + E \Rightarrow i + i + i$$

$$E \Rightarrow E + E \Rightarrow i + E \Rightarrow i + E + E \Rightarrow i + i + E \Rightarrow i + i + i$$

L'ambiguità è causata dalla mancata imposizione di un unico ordine di generazione, da sinistra a destra, o viceversa.

Guardando alle frasi come espressioni aritmetiche, la grammatica non precisa in quale ordine il calcolo del valore vada fatto.

L'ambiguità si elimina osservando che il linguaggio generato non è altro che una lista con separatore,  $L(G_1) = i(+i)^*$ , che può essere definito da una grammatica inambigua, ricorsiva soltanto a destra,  $E \rightarrow i + E \mid i$ ; o, dualmente, da una grammatica ricorsiva a sinistra  $E \rightarrow E + i \mid i$ .

*Esempio 2.50.* Ricorsioni sinistra e destra in regole diverse.

Un secondo caso di ambiguità, causata da un nonterminale bilateralmente ricorsivo, è la grammatica  $G_2$ :

$$A \rightarrow aA \mid Ab \mid c$$

Anche questo linguaggio  $L(G_2) = a^*cb^*$  è regolare, esso è il concatenamento di due liste,  $a^*$  e  $b^*$ , con interposto  $c$  come elemento cuscinetto. Per eliminare l'ambiguità, basta generare ognuna delle due liste con regole separate: La decomposizione suggerisce la grammatica:

$$S \rightarrow AcB \quad A \rightarrow aA \mid \varepsilon \quad B \rightarrow bB \mid \varepsilon$$

Un altro modo di togliere l'ambiguità genera la prima lista prima della seconda (o viceversa):

$$S \rightarrow aS \mid X \quad X \rightarrow Xb \mid c$$

Osservazione: una doppia ricorsione non causa ambiguità se essa non è sinistra e destra. Si osservi la grammatica

$$S \rightarrow +SS \mid \times SS \mid i$$

che genera le espressioni polacche prefisse (ad es.  $++ii \times ii$ ) con i segni di somma e moltiplicazione. Pur essendovi due regole doppiamente ricorsive rispetto a  $S$ , vi sono ricorsioni destre ma non sinistre, e la grammatica non è ambigua.

### Ambiguità dell'unione

Se due linguaggi  $L_1 = L(G_1)$  e  $L_2 = L(G_2)$  condividono alcune frasi, ossia la loro intersezione non è vuota, la grammatica  $G$  del linguaggio unione dei due, ottenuta con la costruzione classica (p. 45), risulta ambigua. Si deve naturalmente supporre che ognuna delle due grammatiche abbia nonterminali distinti, altrimenti l'unione delle regole potrebbe generare frasi non appartenenti all'unione dei linguaggi.

Si consideri una frase  $x \in L_1 \cap L_2$ . Essa ammette due derivazioni distinte, una con le regole di  $G_1$  l'altra con quelle di  $G_2$ , quindi risulta ambigua per la grammatica  $G$  che contiene tutte le regole. Ogni frase  $x$  appartenente al primo linguaggio ma non al secondo,  $x \in L_1 \setminus L_2$ , è invece generata con le sole regole di  $G_1$  (e dualmente ogni frase di  $L_2 \setminus L_1$ ).

*Esempio 2.51.* Unione di linguaggi sovrapposti.

Nel progetto dei linguaggi tecnici, questa situazione può capitare in più situazioni.

1. La prima si presenta quando si vuole trattare con regole separate, all'interno d'una classe più ampia, un particolare costrutto, magari per tradurlo in modo particolare. Come esempio, sono date le espressioni aritmetiche additive aventi operandi costanti  $C$  o variabili, denotate da  $i$ . Una possibile grammatica è

$$E \rightarrow E + C \mid E + i \mid C \mid i \quad C \rightarrow 0 \mid 1D \mid \dots \mid 9D \quad D \rightarrow 0D \mid \dots \mid 9D \mid \varepsilon$$

Si supponga che le espressioni come  $i + 1$  o  $1 + i$ , che sommano la costante uno, vadano trattate separatamente dal compilatore, perché possono essere tradotte in modo più efficiente delle altre nel codice della macchina. A tale scopo si aggiungono alla grammatica le regole

$$E \rightarrow i + 1 \mid 1 + i$$

Purtroppo la nuova grammatica risulta ambigua, perché una frase come  $1 + i$  è anche generata dalle regole originali.

2. Un secondo caso è dato dall'uso di uno stesso operatore con significato diverso, in costrutti diversi del linguaggio. Nel linguaggio Pascal il segno '+' può indicare la somma aritmetica:

$$E \rightarrow E + T \mid T \quad T \rightarrow V \quad V \rightarrow \dots$$

o l'unione tra insiemi

$$E_{set} \rightarrow E_{set} + T_{set} \mid T_{set} \quad T_{set} \rightarrow V$$

Per eliminare tali ambiguità il rimedio è drastico. Occorre rendere disgiunti i due costrutti che danno luogo all'ambiguità, oppure fonderli insieme. Purtroppo in nessuno dei due esempi precedenti è agevole disgiungere i costrutti, a meno di modificare il linguaggio.

Infatti nel primo esempio non è possibile sottrarre la stringa '1' dall'insieme delle costanti intere che derivano dal notterminale  $C$ . Il trattamento particolare del valore '1' può essere invece ottenuto, modificando la sintassi del linguaggio con l'aggiunta di un operatore  $inc$  di incremento, sostituendo la regola  $E \rightarrow i + 1 \mid 1 + i$  con la regola  $E \rightarrow inc i$ .

Nel secondo esempio l'ambiguità è di natura semantica, a causa della polisemia dell'operatore '+'. Occorre quindi fondere le produzioni delle espressioni aritmetiche (generate da  $E$ ) e insiemistiche (generate da  $E_{set}$ ), rinunciando a distinguere sintatticamente le une dalle altre. Altrimenti si può modificare il linguaggio, sostituendo al '+' il carattere 'U' per denotare l'unione tra insiemi.

I prossimi esempi si prestano invece all'eliminazione della sovrapposizione.

*Esempio 2.52.* (McNaughton)

1. La grammatica  $G$ :

$$S \rightarrow bS \mid cS \mid D \quad D \rightarrow bD \mid cD \mid \varepsilon$$

è ambigua perché  $L(G) = \{b, c\}^* = L_D(G)$ . Le derivazioni

$$S \stackrel{+}{\Rightarrow} bbcD \Rightarrow bbc \quad S \Rightarrow D \stackrel{+}{\Rightarrow} bbcD \Rightarrow bbc$$

hanno lo stesso effetto. Eliminando le regole di  $D$ , che sono ridondanti, si ottiene  $S \rightarrow bS \mid cS \mid \varepsilon$ .

2. La grammatica

$$S \rightarrow B \mid D \quad B \rightarrow bBc \mid \varepsilon \quad D \rightarrow dDe \mid \varepsilon$$

dove  $B$  genera  $b^n c^n, n \geq 0$  e  $D$  genera  $d^n e^n, n \geq 0$  ha una sola frase ambigua:  $\varepsilon$ . La si può generare direttamente dall'assioma:

$$S \rightarrow B \mid D \mid \varepsilon \quad B \rightarrow bBc \mid bc \quad D \rightarrow dDe \mid de$$

**Ambiguità del concatenamento**

Il concatenamento di due linguaggi può causare ambiguità, quando una frase del primo linguaggio ha un suffisso che sia anche un prefisso d'una frase del secondo linguaggio.

La costruzione della grammatica  $G$  del concatenamento  $L_1 L_2$  (p. 45) aggiunge la regola  $S \rightarrow S_1 S_2$  alle regole di  $G_1$  e di  $G_2$  (entrambe per ipotesi inambigue). La grammatica è ambigua se esistono nei due linguaggi le frasi:

$$u' \in L_1 \quad u'v \in L_1 \quad vz'' \in L_2 \quad z'' \in L_2$$

Allora la stringa  $u'vz''$  appartiene al linguaggio  $L_1 \cdot L_2$  ed è ambigua poiché può essere derivata in due modi:

$$S \Rightarrow S_1 S_2 \stackrel{+}{\Rightarrow} u' S_2 \stackrel{+}{\Rightarrow} u' v z'' \quad S \Rightarrow S_1 S_2 \stackrel{+}{\Rightarrow} u' v S_2 \stackrel{+}{\Rightarrow} u' v z''$$

*Esempio 2.53.* Concatenamento di linguaggi di Dyck.

Si guardi il concatenamento di due linguaggi di Dyck (p. 44)  $L_1$  e  $L_2$  di alfabeti rispettivi  $\{a, a', b, b'\}$  e  $\{b, b', c, c'\}$ . Una frase di  $L = L_1 L_2$  è  $aa'bb'cc'$ . La grammatica ovvia di  $L$  è

$$S \rightarrow S_1 S_2 \quad S_1 \rightarrow aS_1 a' S_1 \mid bS_1 b' S_1 \mid \varepsilon \quad S_2 \rightarrow bS_2 b' S_2 \mid cS_2 c' S_2 \mid \varepsilon$$

La frase è derivabile nei due modi schematizzati;

$$\begin{array}{ccc} \overbrace{aa'}^{S_1} & \overbrace{bb'}^{S_2} & \overbrace{cc'} \\ aa'bb' & cc' & \\ \overbrace{aa'}^{S_1} & \overbrace{bb'}^{S_2} & \overbrace{cc'} \\ aa' & bb' & cc' \end{array}$$

Per rimediare all'ambiguità del concatenamento, occorre impedire lo spostamento d'una stringa dal suffisso del primo linguaggio al prefisso del secondo (o viceversa).

Se è lecito modificare il linguaggio, un rimedio semplice è di interporre tra i due linguaggi un carattere terminale, che faccia da separatore. Per garantire che il separatore non crei confusione, basta che esso non appartenga agli alfabeti terminali.

Nell'esempio precedente, scelto il separatore  $\sharp$ , il linguaggio  $L_1 \sharp L_2$  è facilmente generato in modo non ambiguo dalla grammatica avente la regola iniziale  $S \rightarrow S_1 \sharp S_2$ .

Altrimenti, complicando un po' la soluzione, si potrebbe scrivere una grammatica inambigua, avente la proprietà di attribuire al linguaggio  $L_1$  le sottostringhe come  $bb'$  che non contengono delle  $c$ . Si noti che la stringa  $bcc'b'$  va invece attribuita al secondo linguaggio.

### Codici univoci e non

Una bell'illustrazione della ambiguità di concatenamento viene dallo studio dei codici nella teoria dell'informazione. Una sorgente può essere vista come un processo che produce un messaggio, ovvero una sequenza di simboli d'un insieme finito  $\Gamma = \{A, B, \dots, Z\}$ . I simboli generati dalla sorgente sono poi codificati come stringhe d'un alfabeto terminale  $\Sigma$ , tipicamente binario, usando una funzione di codifica, che fa corrispondere a ogni simbolo una stringa terminale, il suo codice.

Si considerino ad es. i simboli della sorgente e i loro codici di alfabeto  $\Sigma = \{0, 1\}$ :

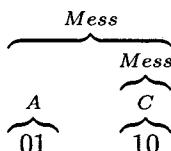
$$\Gamma = \{ \overbrace{A}^{01}, \overbrace{C}^{10}, \overbrace{E}^{11}, \overbrace{R}^{001} \}$$

Allora il messaggio *ARRECA* viene codificato nella stringa 01 001 001 11 10 01, dalla quale si decodifica, ossia si ricostruisce univocamente, il testo originale. La codifica d'un messaggio è espressa dalla grammatica seguente  $G_1$ :

$$Mess \rightarrow A \text{ } Mess \mid C \text{ } Mess \mid E \text{ } Mess \mid R \text{ } Mess \mid A \mid C \mid E \mid R$$

$$A \rightarrow 01 \quad C \rightarrow 10 \quad E \rightarrow 11 \quad R \rightarrow 001$$

La grammatica genera un messaggio, come *AC*, concatenando i codici dei simboli, come appare dall'albero sintattico:



Poiché la grammatica non è ambigua, ogni messaggio codificato, ossia ogni frase del linguaggio  $L(G_1)$ , ha un solo albero sintattico e quindi ammette una

sola decodifica.

Al contrario, la seguente cattiva scelta della funzione di codifica

$$\Gamma = \{ \overbrace{A}^{00}, \overbrace{C}^{01}, \overbrace{E}^{10}, \overbrace{R}^{010} \}$$

rende ambigua la grammatica

$$Mess \rightarrow A\,Mess \mid C\,Mess \mid E\,Mess \mid R\,Mess \mid A \mid C \mid E \mid R$$

$$A \rightarrow 00 \quad C \rightarrow 01 \quad E \rightarrow 10 \quad R \rightarrow 010$$

e non univoca la decifrabilità del messaggio 00010010100100, che può essere decodificato come *ARRECA* o come *ACAECA*.

Il difetto nasce dalla combinazione di due fatti: vale l'eguaglianza

$$\underbrace{01}_{\text{primo}} \cdot 00 \cdot 10 = \underbrace{010}_{\text{primo}} \cdot 010$$

tra le stringhe ottenute concatenando codici diversi; e i primi codici, 01 e 010, sono l'uno prefisso dell'altro.

La teoria dei codici studia queste e altre condizioni atte a garantire che un insieme di codici sia univocamente decifrabile.

### Altre situazioni ambigue

La prossima ambiguità si ricollega a quella d'una espressione regolare (p. 22).

*Esempio 2.54.* La grammatica è:

$$S \rightarrow DcD \quad D \rightarrow bD \mid cD \mid \epsilon$$

Per la prima regola, ogni frase contiene almeno una *c*; le alternative di *D* generano  $\{b, c\}^*$ . La struttura è dunque la stessa della espressione regolare  $\{b, c\}^* c \{b, c\}^*$  che è ambigua: ogni frase contenente due o più *c* è ambigua, perché ogni comparsa di *c* potrebbe essere quella individuata dalla prima regola. Questa ambiguità si può sanare imponendo che il *c* da individuare sia quello più a sinistra nella frase:

$$S \rightarrow BcD \quad D \rightarrow bD \mid cD \mid \epsilon \quad B \rightarrow bB \mid \epsilon$$

dove *B* non deriva stringhe contenenti *c*.

*Esempio 2.55.* Imposizione di ordine alle regole

La grammatica

$$S \rightarrow bSc \mid bbSc \mid \epsilon$$

ha il difetto che la regola che genera due *b* può essere applicata prima o dopo la regola che genera un solo *b*. Ne risulta l'ambiguità:

$$S \Rightarrow bbSc \Rightarrow bbbScc \Rightarrow bbbcc \quad S \Rightarrow bSc \Rightarrow bbbScc \Rightarrow bbbcc$$

Un rimedio consiste nel precisare che la regola che genera un solo *b* debba sempre precedere quella che ne genera due:

$$S \rightarrow bSc \mid D \quad D \rightarrow bbDc \mid \epsilon$$

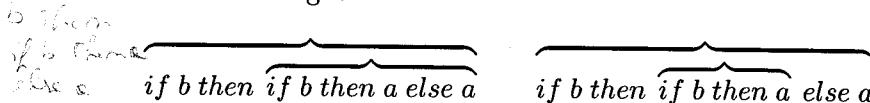
### Ambiguità delle frasi condizionali

L'esempio più spesso citato di ambiguità nei linguaggi di programmazione riguarda le frasi condizionali, nella prima versione del linguaggio Algol 60,<sup>18</sup> uno dei primi impieghi di grammatiche libere nella storia dell'informatica. Si consideri la grammatica

$$S \rightarrow if\ b\ then\ S\ else\ S \mid if\ b\ then\ S\mid a$$

dove *b* sta per una condizione booleana e *a* per un'istruzione non condizionale, qui non specificate. La prima alternativa produce un *condizionale doppio*, la seconda *semplice*.

L'ambiguità nasce quando si annidano una nell'altra due frasi, la più esterna delle quali sia un condizionale doppio. Ad es. la frase *if b then if b then a else a* offre due letture ambigue:



*if b then  
if b then a  
else a*

In modo espressivo si usa dire che lo *else* sballonzola.

L'ambiguità è eliminabile, pur di complicare la grammatica. Si decida di scegliere l'albero scheletrico di sinistra, che attribuisce lo *else* allo *if* immediatamente precedente.

La nuova grammatica è:

$$S \rightarrow S_E \mid S_T \quad S_E \rightarrow if\ b\ then\ S_E\ else\ S_E \mid a$$

$$S_T \rightarrow if\ b\ then\ S_E\ else\ S_T \mid if\ b\ then\ S$$

La categoria sintattica *S* è stata divisa in due: *S<sub>E</sub>* definisce un condizionale doppio e tale che i due eventuali condizionali in esso presenti siano ancora del tipo *S<sub>E</sub>*. L'altra categoria, *S<sub>T</sub>*, definisce un condizionale semplice, oppure un condizionale doppio, tale che il primo condizionale contenuto sia del tipo *S<sub>E</sub>* e il secondo sia del tipo *S<sub>T</sub>*; sono quindi esclusi i casi

$$if\ b\ then\ S_T\ else\ S_T \quad \text{e} \quad if\ b\ then\ S_T\ else\ S_E$$

Il fatto che solo *S<sub>E</sub>* può precedere *else*, mentre solo *S<sub>T</sub>* definisce un condizionale semplice, esclude l'albero scheletro indesiderabile di destra.

<sup>18</sup>Nella successiva versione ufficiale [37] l'inconveniente fu eliminato.

Peraltro, se al progettista fosse consentito modificare il linguaggio, sarebbe più semplice eliminare l'ambiguità introducendo una marca di chiusura per delimitare il costrutto, come fa la grammatica seguente, che usa la marca *end-if*:

$$S \rightarrow if\ b\ then\ S\ else\ S\ end\_if\ | if\ b\ then\ S\ end\_if\ | a$$

Questa modifica è una forma di parentesizzazione della grammatica (p. 45).

### Ambiguità inherente del linguaggio

Si è finora riscontrato che un linguaggio libero può essere definito da grammatiche equivalenti, alcune ambigue altre non, ma tale circostanza non è sempre verificata. Un linguaggio si dice *inherentemente ambiguo* se tutte le grammatiche (tra loro equivalenti) che lo generano sono ambigue.

Per quanto ciò possa sembrare sorprendente, esistono dei linguaggi liberi inherentemente ambigi.

*Esempio 2.56.* Ambiguità d'unione non eliminabile.

Si riveda l'es. 2.46 di p. 46, il linguaggio

$$L = \{a^i b^j c^k \mid (i, j, k \geq 0) \wedge ((i = j) \vee (j = k))\}$$

anche definito come unione dei linguaggi

$$L = \{a^i b^i c^* \mid i \geq 0\} \cup \{a^* b^i c^i \mid i \geq 0\} = L_1 \cup L_2$$

non disgiunti.

Segue un argomento intuitivo per giustificare l'affermazione che ogni grammatica di questo linguaggio è ambigua. La grammatica di p. 46, ottenuta unendo le regole delle due grammatiche componenti, è ambigua per le frasi  $\varepsilon, abc, \dots a^i b^i c^i$  comuni ai due linguaggi. Esse infatti sono generate da  $G_1$  con regole sintattiche che verificano se  $|x|_a = |x|_b$ , operazione che richiede una struttura sintattica del tipo

$$\overbrace{a \dots a}^{\text{under } a} \underbrace{ab}_{\text{under } b} \underbrace{b \dots b}_{\text{under } b} \underbrace{cc \dots c}_{\text{under } c}$$

Ma la stessa  $x$ , in quanto appartenente a  $L_2$ , richiede di essere generata anche con la struttura

$$\overbrace{a \dots aa}^{\text{under } a} \underbrace{b \dots b}_{\text{under } b} \underbrace{bc}_{\text{under } c} \underbrace{c \dots c}_{\text{under } c}$$

per verificare che sia  $|x|_b = |x|_c$ . Comunque si modifichi la grammatica, i due controlli sulle egualianze degli esponenti di dette frasi sono inevitabili e la grammatica rimane ambigua.

Per fortuna le ambiguità inherenti al linguaggio sono rare e non si riscontrano nei casi di interesse pratico.

### 2.5.12 Equivalenza debole e strutturale

Una grammatica non serve soltanto a definire le frasi di un linguaggio, ma ha lo scopo di assegnare ad ogni frase una struttura, in modo coerente con il suo significato. Questa esigenza di *adeguatezza strutturale* è stata più volte affermata, ad es. quando si è considerato la precedenza tra gli operatori nelle espressioni a più livelli.

Prima di approfondire il concetto di equivalenza, si ricorda la precedente definizione di p. 36: due grammatiche sono equivalenti se generano lo stesso linguaggio,  $L(G) = L(G')$ .

Tale definizione di equivalenza è detta *debole*, ed è poco rispondente all'esigenza pratica di caratterizzare la reale sostituibilità di due grammatiche, al fine del loro impiego nella definizione d'un linguaggio artificiale e nella costruzione del compilatore. Infatti le grammatiche potrebbero assegnare strutture diversissime alla stessa frase, una delle quali potrebbe essere inadatta all'interpretazione semantica desiderata.

Conviene dunque introdurre una definizione più stringente, limitandosi per semplicità al caso delle grammatiche inambigue.

Due grammatiche  $G$  e  $G'$  sono equivalenti in senso forte o strutturale, se  $L(G) = L(G')$  e inoltre  $G$  e  $G'$  assegnano a ogni frase due alberi sintattici, che possono essere considerati strutturalmente simili.

La condizione di somiglianza tra alberi può essere così precisata: due alberi sintattici sono strutturalmente simili se i corrispondenti alberi scheletrici condensati (p. 42) sono eguali. Ma altre definizioni di somiglianza sarebbero possibili.

Due grammatiche equivalenti in senso debole sono allora strutturalmente equivalenti se, per ogni frase, gli alberi scheletrici condensati, per  $G$  e  $G'$ , sono eguali.

• L'equivalenza in senso forte implica quella debole, ma a differenza di questa, è una proprietà decidibile.<sup>19</sup>

*Esempio 2.57.* Adeguatezza strutturale di espressioni aritmetiche.

Per illustrare la differenza tra equivalenza debole e strutturale, si definisce una espressione aritmetica, quale  $3 + 5 \times 8 + 2$ , come lista di cifre separate dai segni di somma e prodotto.

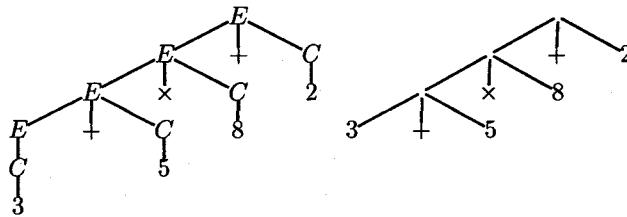
- Prima grammatica  $G_1$ :

$$\begin{array}{l} E \rightarrow E + C \quad E \rightarrow E \times C \quad E \rightarrow C \\ C \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

L'albero sintattico della frase precedente è:



<sup>19</sup>L'algoritmo, esposto in [43], è simile a quello per decidere l'equivalenza di due automi finiti, che si vedrà nel prossimo capitolo.

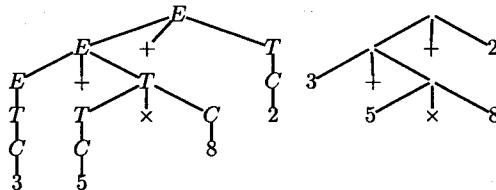


Nella struttura scheletrica di destra sono omessi i nonterminali; anche le regole di categorizzazione, come  $E \rightarrow C$ , sono state tolte poiché non producono ramificazioni nell'albero.

- Una seconda grammatica  $G_2$  per lo stesso linguaggio è:

$$\begin{array}{ll} E \rightarrow E + T & E \rightarrow T \\ C \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 & T \rightarrow T \times C \\ & T \rightarrow C \end{array}$$

Essa è equivalente (in senso debole) alla precedente. La stessa frase riceve dalla grammatica  $G_2$  una diversa struttura:



Si noti nell'albero scheletrico il sottoalbero con frontiera  $5 \times 8$ , moltiplicazione, che mancava nel precedente. Le due grammatiche non sono quindi equivalenti in senso strutturale.

Ci si chiede se una delle due sia da preferire in qualche senso. Se una grammatica fosse inambigua ma non l'altra, certamente la prima sarebbe da preferire; ma questo non è il caso.

Si osservi che soltanto la grammatica  $G_2$  è *adeguata strutturalmente*, se si prende in considerazione il significato del linguaggio. Infatti la frase  $3 + 5 \times 8 + 2$  esprime un calcolo aritmetico, da eseguirsi nell'ordine tradizionale:  $3 + (5 \times 8) + 2 = (3 + 40) + 2 = (43 + 2) = 45$ .

Questa è l'*interpretazione semantica* che si intende assegnare al linguaggio. Le sottoespressioni via via calcolate sono state racchiuse tra parentesi: l'ordine corretto di calcolo è quello espresso dalla stringa completamente parentesizzata:

$$((3 + (5 \times 8)) + 2)$$

i cui gruppi coincidono con i sottoalberi dell'albero scheletrico di  $G_2$ . Invece la  $G_1$  produce la parentesizzazione

$$(((3 + 5) \times 8) + 2)$$

strutturalmente inadeguata, perché, dando la precedenza alla prima somma sul prodotto, assegna alla frase l'interpretazione semantica scorretta 66 invece

di 45.

Per inciso si osservi che la seconda grammatica è più complicata della prima: l'adeguatezza strutturale ha non di rado un costo.

Si deve sottolineare che nella definizione formale di un linguaggio non si può prescindere dall'adeguatezza strutturale, se la grammatica deve servire, come di regola avviene, da supporto per l'interpretazione semantica o per la traduzione guidata dalla sintassi, come meglio si dirà nei due ultimi capitoli.

- Il nuovo concetto di equivalenza è esemplificato dalla grammatica  $G_3$ , che equivale strutturalmente alla precedente<sup>20</sup>:

$$\begin{aligned} E &\rightarrow E + T \mid T + T \mid C + T \mid E + C \mid T + C \mid C + C \mid T \times C \mid C \times C \mid C \\ T &\rightarrow T \times C \mid C \times C \mid C \\ C &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

L'albero scheletrico di ogni espressione aritmetica è lo stesso della grammatica  $G_2$ . Dunque le grammatiche  $G_2$  e  $G_3$  forniscono un supporto equivalente ai fini dell'interpretazione semantica delle frasi.

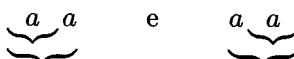
### **Equivalenza strutturale in senso lato**

Per rendere più flessibile il concetto di equivalenza strutturale di due grammatiche  $G$  e  $G'$ , la condizione di egualanza tra gli alberi sintattici scheletrici può essere sostituita dalla richiesta che i due alberi della stessa frase siano facilmente trasformabili uno nell'altro. A seconda degli ambiti, si dovrà precisare che cosa s'intende: ad es. si potranno dire facilmente trasformabili due alberi quando si possa stabilire una corrispondenza biunivoca fra i loro sottoalberi.

Le grammatiche

$$\{S \rightarrow Sa \mid a\} \qquad \{X \rightarrow aX \mid a\}$$

sono debolmente equivalenti nel generare  $L = a^+$ . Però gli alberi scheletrici condensati d'una frase come  $aa$  sono diversi nei due casi:



Le due grammatiche possono tuttavia essere considerate strutturalmente equivalenti in senso lato, perché a ogni albero lineare a sinistra della prima corrisponde un albero lineare a destra della seconda.

<sup>20</sup>Questa grammatica ha più regole della precedente a causa del fatto che non sfrutta tutte le regole di categorizzazione della precedente. In ogni campo del sapere le categorizzazioni e le tassonomie hanno l'effetto di ridurre la complessità della descrizione.

L'intuizione che le due grammatiche siano sostituibili una per l'altra è soddisfatta, perché esse, non solo generano lo stesso linguaggio, ma due alberi corrispondenti sono specularmente identici, ossia sistematicamente ottenibili uno dall'altro girando le ricorsioni da sinistra a destra.

### 2.5.13 Trasformazioni delle grammatiche e forme normali

Si presentano numerose trasformazioni delle regole, che preservano il linguaggio generato e consentono di ottenere grammatiche che godono di certe proprietà. Le forme normali delle grammatiche sono caratterizzate da certe restrizioni, che però non riducono la famiglia dei linguaggi generati. L'impiego delle forme normali è di interesse prevalentemente teorico, per semplificare la dimostrazione di molti teoremi; ma talvolta ingigantisce e oscura la grammatica. In sostanza l'uniformità di alcune forme normali le rende attraenti per le dimostrazioni matematiche, ma meno aderenti alle esigenze del progettista di linguaggi artificiali.

Tuttavia hanno importanza nel progetto degli analizzatori sintattici certe trasformazioni, come la conversione delle ricorsioni da sinistra a destra. Segue una rassegna delle forme normali e delle trasformazioni per ottenerle.

Sia  $G = (V, \Sigma, P, S)$  la grammatica di partenza.

#### Espansione d'un nonterminale

Prima di considerare la costruzione delle forme normali, conviene introdurre una diffusa trasformazione delle grammatiche, che conserva il linguaggio: la *espansione* d'un simbolo nonterminale nelle sue alternative.

Se alla generica regola  $A \rightarrow \alpha B \gamma$  si sostituiscono le regole

$$A \rightarrow \alpha \beta_1 \gamma \mid \alpha \beta_2 \gamma \mid \dots \mid \alpha \beta_n \gamma$$

dove  $B \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$  sono tutte le alternative di  $B$ , il linguaggio non cambia: infatti si è così trasformata una derivazione  $A \Rightarrow \alpha B \gamma \Rightarrow \alpha \beta_i \gamma$  in una derivazione immediata  $A \Rightarrow \alpha \beta_i \gamma$ .

#### Eliminazione dell'assioma dalle parti destre

E' sempre possibile restringere le parti destre delle regole a essere stringhe in  $(\Sigma \cup (V \setminus \{S\}))$ , cioè *prive dell'assioma*. Basta introdurre un nuovo assioma  $S_0$  e la regola  $S_0 \rightarrow S$ .

#### Nonterminali annullabili e forma normale senza regole vuote

Un nonterminale  $A$  è *annullabile* se esiste una derivazione  $A \xrightarrow{*} \epsilon$ , che deriva la stringa vuota.

Detto  $Null \subseteq V$  l'insieme dei nonterminali annullabili, le seguenti clausole logiche ne definiscono il calcolo, che termina quando si raggiunge un punto fisso, ossia quando l'insieme calcolato non muta più:

$$A \in Null \text{ if } A \rightarrow \varepsilon \in P$$

$$A \in Null \text{ if } (A \rightarrow A_1 A_2 \dots A_n \in P \text{ con } A_i \in V \setminus \{A\}) \wedge \forall A_i (A_i \in Null)$$

La prima riga trova i nonterminali immediatamente annullabili; la seconda trova i nonterminali che derivano una stringa di altri nonterminali annullabili.

*Esempio 2.58. Calcolo dei nonterminali annullabili.*

$$S \rightarrow SAB \mid AC \quad A \rightarrow aA \mid \varepsilon \quad B \rightarrow bB \mid \varepsilon \quad C \rightarrow cC \mid c$$

Eseguendo l'algoritmo risulta:  $Null = \{A, B\}$ . Se vi fosse la regola  $S \rightarrow AB$  risulterebbe anche  $S \in Null$ .

La *forma normale senza regole vuote* (o *non annullabile*) è caratterizzata dalla condizione: nessun nonterminale diverso dall'assioma è annullabile.

È palese che l'assioma è annullabile soltanto se la stringa vuota è nel linguaggio.

Per costruire la forma normale, calcolato l'insieme  $Null$  per la grammatica data  $G$ , si fanno le seguenti modifiche.

Per ogni regola  $A \rightarrow A_1 A_2 \dots A_n \in P$ , con  $A_i \in V \cup \Sigma$ , si aggiungono come regole alternative quelle ottenute cancellando dalla parte destra, in tutti i modi possibili, i simboli nonterminali  $A_i$  annullabili.

Si tolgono poi le regole  $A \rightarrow \varepsilon$ , per ogni  $A \neq S$ .

La grammatica ottenuta può risultare non pulita o circolare, e va mondata con gli algoritmi noti (p. 37).

*Esempio 2.59. (Es. 2.58 continuato)*

Nella prossima tabella, la prima colonna è il predicato di annullabilità sopra calcolato. Si riportano fianco a fianco le regole originali e nella forma non annullabile:

| Annul. | $G$ originale                       | $G'$ da pulire   | $G'$ senza regole vuote                   |
|--------|-------------------------------------|--|---|
| F      | $S \rightarrow SAB$                 | $S \rightarrow SAB \mid SA\varepsilon \mid SB\varepsilon \mid S$ | $S \rightarrow SA \mid SB \mid AC \mid C$ |
|        | $AC$                                | $  AC \mid C$  |   |
| V      | $A \rightarrow aA \mid \varepsilon$ | $A \rightarrow aA \mid a\varepsilon \mid \varepsilon$            | $A \rightarrow aA \mid a$                 |
| V      | $B \rightarrow bB \mid \varepsilon$ | $B \rightarrow bB \mid b\varepsilon \mid \varepsilon$            | $B \rightarrow bB \mid b$                 |
| F      | $C \rightarrow cC \mid c$           | $C \rightarrow cC \mid c\varepsilon \mid \varepsilon$            | $C \rightarrow cC \mid c$                 |

### Copiature o sottocategorizzazioni e loro eliminazione

Si chiama *copiatura* (o *sottocategorizzazione*) una regola  $A \rightarrow B$ , dove  $B \in V$  è un simbolo nonterminale. Tale regola esprime la proprietà di inclusione  $L_B(G) \subseteq L_A(G)$ . Si dice che la classe sintattica  $B$  è inclusa nella classe  $A$ . Un esempio concreto: le regole

$$\text{frase\_iterativa} \rightarrow \text{frase\_while} \mid \text{frase\_for} \mid \text{frase\_repeat}$$

dichiarano che vi sono tre sottocategorie di frasi iterative: le frasi **for**, le frasi **while** e le frasi **repeat**.

Si può fare a meno delle copiature senza perdita di generalità, ma spesso con grave deterioramento della leggibilità perché le regole proliferano. La grammatica equivalente senza copiature genera alberi sintattici di minore profondità. Per la grammatica  $G$  e il nonterminale  $A$ , si definisce l'insieme  $\text{Copia}(A) \subseteq V$  dei nonterminali in cui esso si può ricopiare, anche transitivamente:

$$\text{Copia}(A) = \{B \in V \mid \text{esiste la derivazione } A \xrightarrow{*} B\}$$

Nota: Se in  $G$  vi fosse un nonterminale  $C$  annullabile, la derivazione potrebbe avere la forma

$$A \xrightarrow{\pm} BC \Rightarrow B$$

Per semplicità si fa l'ipotesi che la grammatica sia nella forma normale senza regole vuote.

Il calcolo di *Copia* è espresso dalle seguenti clausole logiche:

$$\begin{aligned} A &\in \text{Copia}(A) && \text{-- inizializzazione} \\ C &\in \text{Copia}(A) \text{ if } (B \in \text{Copia}(A)) \wedge (B \rightarrow C \in P) \end{aligned}$$

che vanno applicate finché si raggiunge il punto fisso.

Poi si costruiscono le regole  $P'$  della grammatica  $G'$ , equivalente a  $G$  e priva di copiatura, nel modo seguente:

$$\begin{aligned} P' &:= P \setminus \{A \rightarrow B \mid A, B \in V\} && \text{-- cancellazione delle copiature} \\ P' &:= \{A \rightarrow \alpha \mid \alpha \in ((\Sigma \cup V)^* \setminus V)\}, \text{ dove } (B \rightarrow \alpha) \in P \wedge B \in \text{Copia}(A) \end{aligned}$$

L'effetto è che la derivazione  $A \xrightarrow{\pm} B \Rightarrow \alpha$  si contrae nella derivazione immediata  $A \Rightarrow \alpha$ .

Si noti che questa trasformazione conserva tutte le regole originali che non siano di copiatura. Nel capitolo 3 la stessa trasformazione sarà applicata per eliminare le mosse spontanee d'un automa finito.

*Esempio 2.60.* Espressioni aritmetiche senza copiature.

Per la grammatica  $G_2$

$$\begin{array}{ll} E \rightarrow E + T \mid T & T \rightarrow T \times C \mid C \\ C \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

risulta:

$$\text{Copia}(E) = \{E, T, C\}, \quad \text{Copia}(T) = \{T, C\}, \quad \text{Copia}(C) = \{C\}$$

La grammatica equivalente senza copiature è:

$$\begin{array}{l} E \rightarrow E + T \mid T \times C \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ T \rightarrow T \times C \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ C \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

In una grammatica senza copiature gli alberi sintattici si accorciano perché scompaiono i cammini privi di biforazioni.

Giova ripetere che la presenza delle copiature permette di mettere in comune certe regole, riducendo così le dimensioni della grammatica. Per questa ragione, nelle grammatiche dei linguaggi tecnici le copiature sono sempre sfruttate.

### **Forma normale di Chomsky**

Le regole sono di due tipi:

1. regole *omogenee binarie*  $A \rightarrow BC$ , dove  $B, C \in V$
2. regole *terminali con parte destra unitaria*  $A \rightarrow a$ , dove  $a \in \Sigma$

Inoltre, se la stringa vuota è nel linguaggio, si ha la regola  $S \rightarrow \epsilon$ ; in tal caso però l'assioma non può stare nella parte destra d'una regola.

Ora negli alberi sintattici il grado dei nodi vale due per i nodi interni e uno per i nodi padri di una foglia.

Per semplicità si suppone che la grammatica data sia priva di nonterminali annullabili. Per ottenere la forma di Chomsky si procede nel modo seguente. Ogni regola  $A_0 \rightarrow A_1 A_2 \dots A_n$  di lunghezza  $n > 2$  è trasformata in una regola di lunghezza due, mettendo in evidenza il simbolo iniziale  $A_1$  e il rimanente suffisso  $A_2 \dots A_n$ . Si introduce un nuovo nonterminale di servizio, denominato  $\langle A_2 \dots A_n \rangle$  con la regola

$$\langle A_2 \dots A_n \rangle \rightarrow A_2 \dots A_n$$

Si sostituisce la regola originale con

$$A_0 \rightarrow A_1 \langle A_2 \dots A_n \rangle$$

Se il simbolo  $A_1$  è terminale, la regola è ulteriormente trasformata nelle due regole in forma di Chomsky

$$A_0 \rightarrow \langle A_1 \rangle \langle A_2 \dots A_n \rangle \quad \langle A_1 \rangle \rightarrow A_1$$

dove  $\langle A_1 \rangle$  è un nuovo nonterminale. Si riapplicano le stesse trasformazioni alla grammatica così ottenuta, finché ogni regola è nella forma voluta.

*Esempio 2.61.* Conversione in forma di Chomsky.

La grammatica:

$$S \rightarrow dA \mid cB \quad A \rightarrow dAA \mid cS \mid c \quad B \rightarrow cBB \mid dS \mid d$$

si trasforma nella forma normale di Chomsky:

$$S \rightarrow \langle d \rangle A \mid \langle c \rangle B \quad A \rightarrow \langle d \rangle \langle AA \rangle \mid \langle c \rangle S \mid c \quad B \rightarrow \langle c \rangle \langle BB \rangle \mid \langle d \rangle S \mid d$$

$$\langle d \rangle \rightarrow d \quad \langle c \rangle \rightarrow c \quad \langle AA \rangle \rightarrow AA \quad \langle BB \rangle \rightarrow BB$$

Questa forma è molto usata nei trattati matematici.

### Trasformazione delle ricorsioni sinistre in destre

La forma *non ricorsiva a sinistra* esclude la presenza di regole o derivazioni ricorsive a sinistra (s-ricorsioni); essa è indispensabile per la costruzione degli analizzatori sintattici discendenti, studiati nel capitolo 4. Si presenta il metodo di sostituzione delle s-ricorsioni con ricorsioni destre.

#### Trasformazione delle s-ricorsioni immediate

Il caso più comune e semplice è quello delle s-ricorsioni immediate.

Siano

$$A \rightarrow A\beta_1 \mid A\beta_2 \mid \dots \mid A\beta_h, h \geq 1 \quad A \Rightarrow A\beta_1 - A\beta_h$$

dove nessun  $\beta_i$  è vuoto, le alternative s-ricorsive di  $A$  e siano

$$A \rightarrow \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_k, k \geq 1 \quad A \Rightarrow \gamma_1 - \gamma_k$$

le rimanenti alternative.

Si crea un nuovo nonterminale  $A'$  e si scrivono, al posto delle precedenti, le regole:

$$A \rightarrow \gamma_1 A' \mid \gamma_2 A' \mid \dots \mid \gamma_k A' \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_k$$

$$A' \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_h A' \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_h$$

Ora ogni derivazione originale s-ricorsiva, quale ad es.

$$A \Rightarrow A\beta_2 \Rightarrow A\beta_3\beta_2 \Rightarrow \gamma_1\beta_3\beta_2$$

è sostituita dall'equivalente derivazione ricorsiva a destra

$$A \Rightarrow \gamma_1 A' \Rightarrow \gamma_1\beta_3 A' \Rightarrow \gamma_1\beta_3\beta_2$$

*Esempio 2.62.* Spostamento a destra delle s-ricorsioni immediate.

Nella solita grammatica delle espressioni aritmetiche

$$\begin{array}{l} E \rightarrow E(+T) \mid T \\ T \rightarrow T(*F) \mid F \\ F \rightarrow (E) \mid i \end{array}$$

i nonterminali  $E$  e  $T$  sono immediatamente ricorsivi a sinistra. Con la precedente trasformazione si ottiene la grammatica ricorsiva a destra:

$$E \rightarrow TE' \mid T \quad E' \rightarrow +TE' \mid +T$$

$$T \rightarrow FT' \mid F \quad T' \rightarrow *FT' \mid *F \quad F \rightarrow (E) \mid i$$

Nota: in questo caso basterebbe rovesciare specularmente le regole s-ricorsive, ottenendo

$$E \rightarrow T + E \mid T \quad T \rightarrow F * T \mid F \quad F \rightarrow (E) \mid i$$

ma questa semplice trasformazione non è sempre adeguata.

### *Trasformazione delle s-ricorsioni non immediate*

Per eliminare anche le s-ricorsioni non immediate, si usa il seguente algoritmo. Si supponga per semplicità che la grammatica  $G$  sia in forma omogenea e non annullabile, con regole terminali di lunghezza unitaria; ossia che la forma sia analoga a quella di Chomsky ma senza il vincolo di avere due nonterminali. L'algoritmo esegue due cicli annidati. Il ciclo esterno sfrutta l'espansione per ottenere delle s-ricorsioni immediate. Il ciclo interno trasforma le s-ricorsioni immediate in ricorsioni destre, e introduce nuovi nonterminali.

Sia  $\underline{V} = \{A_1, A_2, \dots, A_m\}$  l'alfabeto nonterminale e  $A_1$  l'assioma. Conviene pensare l'alfabeto nonterminale come ordinato da 1 a  $m$ .

#### *Algoritmo di eliminazione delle ricorsioni sinistre (s-ricorsioni) anche non immediate*

```

for  $i := 1$  to  $m$  do
    for  $j := 1$  to  $i - 1$  do
        sostituisci a ogni regola del tipo  $A_i \rightarrow A_j \alpha$  (con  $i > j$ ) le regole:
         $A_i \rightarrow \gamma_1 \alpha \mid \gamma_2 \alpha \mid \dots \mid \gamma_k \alpha$ 
        (creando possibili s-ricorsioni immediate)
        dove  $A_j \rightarrow \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_k$  sono le alternative di  $A_j$ 
    end do
    elimina, con l'algoritmo precedente, le eventuali s-ricorsioni
    immediate apparse nelle alternative di  $A_i$ ,
    creando il nuovo nonterminale  $A'_i$ 
end do

```

L'idea dell'algoritmo<sup>21</sup> è di modificare le regole in modo che, se una regola inizia con un nonterminale  $A_i \rightarrow A_j$ , allora risulta  $j > i$ .

*Esempio 2.63.* Applicando l'algoritmo alla grammatica  $G_3$ :

$$A_1 \rightarrow A_2 a \mid b \quad A_2 \rightarrow A_2 c \mid A_1 d \mid e$$

che presenta la s-ricorsione  $A_1 \Rightarrow A_2 a \Rightarrow A_1 da$ , si hanno i passi seguenti:

---

<sup>21</sup>La dimostrazione della validità dell'algoritmo si trova in [27] o in [14].

| <i>i j</i> |  | <i>Grammatica</i>   |
|------------|--|---|
| 1          | Elimina le s-ricorsioni immediate di $A_1$ (non ve ne sono)  | idem  |
| 2 1        | <p>Sostituisci a <math>A_2 \rightarrow A_1d</math> le regole ottenute con l'espansione di <math>A_1</math> ottenendo:</p> <p>Elimina la s-ricorsione immediata, ottenendo <math>G'_3</math>:</p> | $\begin{array}{l} A_1 \rightarrow A_2a \mid b \\ A_2 \rightarrow A_2c \mid A_2ad \mid bd \mid e \end{array}$<br>$\begin{array}{l} A_1 \rightarrow A_2a \mid b \\ A_2 \rightarrow bdA'_2 \mid eA'_2 \mid bd \mid e \\ A'_2 \rightarrow cA'_2 \mid adA'_2 \mid c \mid ad \end{array}$ |

La grammatica  $G'_3$  è priva di s-ricorsioni.

Si noti che, con una ovvia modifica, gli stessi algoritmi permettono di trasformare le ricorsioni destre in sinistre, operazione talvolta opportuna per migliorare la grammatica ai fini della compilazione.

### Forma normale di Greibach o in tempo reale

Nella forma normale in *tempo reale* ogni regola inizia con un simbolo terminale:

$$A \rightarrow a\alpha \text{ dove } a \in \Sigma, \alpha \in \{\Sigma \cup V\}^*$$

Un caso particolare del precedente è la forma normale di *Greibach*:

$$A \rightarrow a\alpha \text{ dove } a \in \Sigma, \alpha \in V^*$$

Ogni regola inizia con un carattere terminale, seguito da zero o più nonterminali.

La qualifica ‘in tempo reale’ si spiegherà con una proprietà dell’algoritmo di analisi sintattica: a ogni passo esso legge e consuma un carattere terminale, cosicché il numero di passi per completare l’analisi è esattamente eguale alla lunghezza della stringa da analizzare.

Per la precisione, le forme considerate escludono dal linguaggio la stringa vuota.

Si suppone per semplicità che la grammatica sia nella forma normale non annullabile. Per trasformarla nella forma in tempo reale e in quella di Greibach, per primo si eliminano le ricorsioni sinistre; poi, con trasformazioni elementari, si espandono i nonterminali che fossero presenti in prima posizione e si introducono dei nuovi nonterminali al posto dei terminali che cadessero in posizioni diverse dalla prima.

*Esempio 2.64.* La grammatica

$$A_1 \rightarrow A_2a \quad A_2 \rightarrow A_1c \mid bA_1 \mid d$$

è trasformata nella forma di Greibach attraverso i passi seguenti.

1. Eliminazione delle s-ricorsioni, mediante il passaggio:

$$A_1 \rightarrow A_2a \quad A_2 \rightarrow A_2ac \mid bA_1 \mid d$$

e poi

$$A_1 \rightarrow A_2a \quad A_2 \rightarrow bA_1A'_2 \mid dA'_2 \mid d \mid bA_1 \quad A'_2 \rightarrow acA'_2 \mid ac$$

2. Sostituzione dei nonterminali in prima posizione, fino a far comparire un terminale in prima posizione:

$$A_1 \rightarrow bA_1A'_2a \mid dA'_2a \mid da \mid bA_1a \quad A_2 \rightarrow bA_1A'_2 \mid dA'_2 \mid d \mid bA_1$$

$$A'_2 \rightarrow acA'_2 \mid ac$$

3. Introduzione di nuovi nonterminali al posto dei terminali in posizioni diverse dalla prima:

$$A_1 \rightarrow bA_1A'_2\langle a \rangle \mid dA'_2\langle a \rangle \mid d\langle a \rangle \mid bA_1\langle a \rangle \quad A_2 \rightarrow bA_1A'_2 \mid dA'_2 \mid d \mid bA_1$$

$$A'_2 \rightarrow a\langle c \rangle A'_2 \mid a\langle c \rangle$$

$$\langle a \rangle \rightarrow a \quad \langle c \rangle \rightarrow c$$

Se non si esegue l'ultimo passo dell'algoritmo, nelle regole possono rimanere simboli terminali in posizioni successive alla prima; la grammatica è allora nella forma in tempo reale, pur se non in quella di Greibach.

Mentre nei libri di orientamento teorico le forme normali sono adottate per semplificare le dimostrazioni di molte proprietà, in questo testo il loro uso sarà molto limitato. Tuttavia le manipolazioni presentate sono un valido armamentario di trasformazioni grammaticali per il progettista delle grammatiche dei linguaggi tecnici.

## 2.6 Le grammatiche dei linguaggi regolari

I linguaggi regolari sono un caso molto particolare dei linguaggi liberi, e possono essere generati da grammatiche soggette a una forte restrizione nella forma delle regole. Approfondendo lo studio dei linguaggi regolari, si mostrerà poi che le frasi, al crescere della lunghezza, presentano necessariamente certe ripetitività. Questa proprietà permetterà di dimostrare che certi linguaggi liberi non appartengono alla famiglia *REG*. Altri aspetti del confronto tra i linguaggi regolari e liberi emergeranno nei capitoli 3 e 4 dallo studio della memoria impiegata per riconoscere se una stringa appartiene al linguaggio: memoria finita per i linguaggi regolari e illimitata per quelli liberi.

### 2.6.1 Dalla espressione regolare alla grammatica libera

Data una e.r. non è difficile costruire una grammatica libera che generi lo stesso linguaggio. Il procedimento analizza la struttura sintattica dell'espressione per scrivere le regole della grammatica. Il cuore del procedimento sta nell'uso di regole ricorsive al posto degli operatori iterativi (stella e croce).

*Algoritmo 2.65.* Dalla e.r. alla grammatica

Si decompone ripetutamente la e.r. data  $r$  nelle sue sottoespressioni, numerandole progressivamente. Per la definizione stessa di e.r., i casi possibili sono i seguenti (mettendo la stringa vuota al posto dell'insieme vuoto), ognuno dei quali dà luogo alle regole grammaticali scritte a destra, dove le maiuscole indicano dei simboli nonterminali.

| sottoespressione                         | regola grammaticale  |
|--|--|
| 1 $r = r_1 \cdot r_2 \dots \cdot r_k$    | $E \rightarrow E_1 E_2 \dots E_k$  |
| 2 $r = r_1 \cup r_2 \cup \dots \cup r_k$ | $E \rightarrow E_1 \cup E_2 \cup \dots \cup E_k$                               |
| 3 $r = (r_1)^*$                          | $E \rightarrow E E_1 \mid \epsilon$ oppure $E \rightarrow E_1 E \mid \epsilon$ |
| 4 $r = (r_1)^+$                          | $E \rightarrow E E_1 \mid E_1$ oppure $E \rightarrow E_1 E \mid E_1$           |
| 5 $r = b \in \Sigma$                     | $E \rightarrow b$  |
| 6 $r = \epsilon$                         | $E \rightarrow \epsilon$   |

Per abbreviare la grammatica, in tutte le regole di corrispondenza, se un termine  $r_i$  è un simbolo terminale  $a_i \in \Sigma$ , si scrive direttamente  $a_i$  invece di  $E_i$ . Si noti che nelle 3 e 4 si può scegliere la ricorsione sinistra o destra.

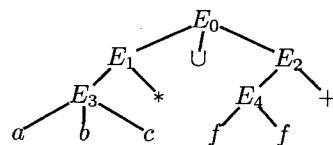
Lo schema di corrispondenza andrà applicato assegnando come nome al non-terminale  $E$  un numero che contraddistingue la sottoespressione considerata. L'assioma è associato al primo passo della decomposizione. È sufficiente un esempio per chiarire il procedimento.

*Esempio 2.66.* Dalla e.r. alla grammatica.

L'espressione

$$E = (abc)^* \cup (ff)^+$$

è decomposta in sottoespressioni, numerate (arbitrariamente) come mostrato nell'albero:



Nell'albero si legge che  $E_0$  è l'unione delle sottoespressioni  $E_1$  e  $E_2$ ,  $E_1$  è la stella della sottoespressione  $E_3$ , ecc.

La figura è una sorta di albero sintattico della e.r. con l'aggiunta della numerazione.

Applicando le corrispondenze, si scrivono le regole della grammatica:

| Corrispondenza | Sottoespressione | Regole sintattiche                      |
|----------------|------------------|---|
| 2              | $E_1 \cup E_2$   | $E_0 \rightarrow E_1 \mid E_2$          |
| 3              | $E_3^*$          | $E_1 \rightarrow E_1 E_3 \mid \epsilon$ |
| 4              | $E_4^+$          | $E_2 \rightarrow E_2 E_4 \mid E_4$      |
| 1              | $a b c$          | $E_3 \rightarrow a b c$                 |
| 1              | $f f$            | $E_4 \rightarrow f f$                   |

Dall'assioma derivano le forme  $E_1$  e  $E_2$ ; il metasimbolo  $E_1$  genera le forme  $E_3^*$ , dalle quali si deriva  $(abc)^*$ . Similmente  $E_2$  genera le stringhe  $E_4^+$ , dalle quali si ottiene  $(ff)^+$ .

Si osservi che, se la e.r. è ambigua (p. 22), anche la grammatica così ottenuta è ambigua (vedasi es. 2.69 a p. 70).

In conclusione, per ogni operatore d'una formula regolare, si è mostrato come scrivere le regole che generano lo stesso linguaggio. Ne segue che ogni linguaggio regolare è libero, e, ricordando che altri linguaggi liberi (come i palindromi o il linguaggio di Dyck) non sono regolari, vale la seguente proprietà.

*Proprietà 2.67.* La famiglia REG dei linguaggi regolari è strettamente contenuta nella famiglia LIB dei linguaggi liberi, ossia è  $REG \subset LIB$ .

### 2.6.2 Grammatiche lineari

L'algoritmo 2.65 costruisce una grammatica del linguaggio d'una e.r.. Ma per un linguaggio regolare si può scrivere una grammatica particolarmente semplice, detta unilineare o del tipo 3. Tale forma mette in evidenza le proprietà caratteristiche d'un linguaggio regolare, e facilita la costruzione dell'algoritmo che ne riconosce le frasi.

Si ricorda che una grammatica è detta *lineare* se ogni regola ha la forma

$$A \rightarrow uBv \text{ dove } u, v \in \Sigma^*, B \in (V \cup \epsilon)$$

ossia se vi è al più un nonterminale nella parte destra.

Pittoricamente un albero sintattico generato da tale grammatica non è ramificato, ma ha un solo stelo centrale cui aderiscono le foglie (terminali). Le grammatiche lineari non sono capaci di generare ogni linguaggio libero (un esempio è il linguaggio di Dyck), ma sono già troppo potenti per i linguaggi regolari. Ad es. il seguente ben noto linguaggio è generato da una grammatica lineare, ma non è regolare (ciò sarà dimostrato in seguito a p. 77).

*Esempio 2.68.* Linguaggio lineare non regolare

$$L_1 = \{b^n e^n \mid n \geq 1\} = \{be, bbbe, \dots\}$$

Grammatica lineare:  $S \rightarrow bSe \mid be$

Una regola è *lineare a destra* se è nella forma:

$$A \rightarrow uB \text{ dove } u \in \Sigma^*, B \in (V \cup \epsilon)$$

Dualmente una regola è *lineare a sinistra* se ha la forma:

$$A \rightarrow Bu, \text{ con le stesse condizioni.}$$

Questi sono casi speciali di regole lineari, caratterizzati dalla presenza d'una sola delle due stringhe terminali che potevano abbracciare il nonterminale  $B$ . Una grammatica, in cui tutte le regole appartengono a uno solo dei due tipi precedenti, è detta *unilineare* o anche del *tipo 3*.<sup>22</sup>

Un albero d'una grammatica lineare a destra (risp. a sinistra) crescerà evidentemente in direzione obliqua verso destra (risp. sinistra).

Analizzando la forma delle grammatiche unilineari appare evidente che, se in una grammatica lineare a destra (sinistra) vi sono derivazioni ricorsive, esse sono ricorsive a destra (sinistra).

*Esempio 2.69.* Le frasi contenenti la sottostringa  $aa$  e terminanti per  $b$  sono definite dalla e.r. (ambigua):

$$(a \mid b)^* aa(a \mid b)^* b$$

Il linguaggio è generato dalle seguenti grammatiche unilineari.

1. Grammatica lineare a destra  $G_d$ :

$$S \rightarrow aS \mid bS \mid aaA \quad A \rightarrow aA \mid bA \mid b$$

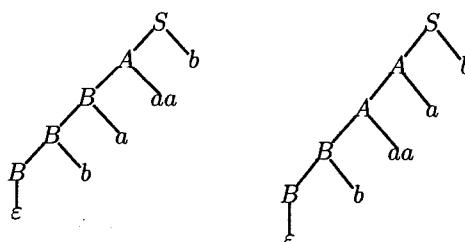
2. Grammatica lineare a sinistra  $G_s$ :

$$S \rightarrow Ab \quad A \rightarrow Aa \mid Ab \mid Baa \quad B \rightarrow Ba \mid Bb \mid \epsilon$$

Una grammatica equivalente, ma non unilineare, è quella prodotta dall'algoritmo di p. 68:

$$E_1 \rightarrow E_2 aa E_2 b \quad E_2 \rightarrow E_2 a \mid E_2 b \mid \epsilon$$

Per la grammatica  $G_s$  si mostrano gli alberi sintattici lineari a sinistra della frase ambigua  $baaab$ :



<sup>22</sup>Con riferimento alla gerarchia di Chomsky (p. 87).

*Esempio 2.70.* Espressioni aritmetiche senza parentesi  
Il linguaggio

$$L = \{a, a + a, a * a, a + a * a, \dots\}$$

è generato dalla grammatica lineare a destra  $G_d$ :

$$S \rightarrow a \mid a + S \mid a * S$$

e anche dalla grammatica lineare a sinistra  $G_s$ :

$$S \rightarrow a \mid S + a \mid S * a$$

Per inciso, queste grammatiche sono inadatte all'impiego nella compilazione perché la struttura sintattica, ponendo allo stesso livello le somme e i prodotti, non rispetta la precedenza convenzionale tra gli operatori.

### *Grammatiche strettamente unilineari*

Conviene convertire una grammatica unilineare in una forma ancora più semplice, detta *strettamente unilineare*, in cui ogni regola contiene al più un carattere terminale, ossia è del tipo

$$A \rightarrow aB \quad (\text{oppure } A \rightarrow Ba), \text{ dove } a \in (\Sigma \cup \varepsilon), B \in (V \cup \varepsilon)$$

Sempre senza perdita di generalità, si può imporre che le regole terminali siano nulle, ossia che la grammatica contenga soltanto regole dei tipi seguenti:

$$A \rightarrow aB \mid \varepsilon \quad \text{dove } a \in \Sigma, B \in V$$

Pertanto si può usare indifferentemente la forma unilineare o quella strettamente unilineare, con regole terminali nulle o non.

*Esempio 2.71.* Es. 2.70 continuato.

Introducendo opportuni nonterminali, la grammatica  $G_d$  si trasforma nella  $G'_d$ , equivalente e strettamente lineare a destra:

$$S \rightarrow a \mid aA \quad A \rightarrow +S \mid *S$$

oppure anche nella forma in cui tutte le regole terminali sono nulle:

$$S \rightarrow aA \quad A \rightarrow +S \mid *S \mid \varepsilon$$

### 2.6.3 Equazioni lineari del linguaggio

Si mostra che la famiglia dei linguaggi generati dalle grammatiche unilineari contiene quella dei linguaggi regolari, attraverso un procedimento matematico per ottenere la e.r. equivalente alla grammatica data. Nel capitolo 3 si vedrà che, non soltanto vi è contenimento, ma le due famiglie coincidono.

Le regole d'una grammatica lineare a destra possono essere trascritte come

equazioni aventi per incognite i linguaggi generati da ogni nonterminale. Sia  $G = (V, \Sigma, P, S)$  la grammatica, che per semplicità si suppone in forma strettamente lineare a destra e con regole terminali nulle.

Al solito, il linguaggio generato partendo dal nonterminale  $A$  è

$$L_A = \{x \in \Sigma^* \mid A \xrightarrow{*} x\}$$

e in particolare è  $L(G) \equiv L_S$ .

Una stringa  $x \in \Sigma^*$  appartiene a  $L_A$  nei seguenti casi:

- $x$  è la stringa vuota e vi è in  $P$  la regola  $A \rightarrow \varepsilon$ ;
- $x$  è la stringa vuota, vi è in  $P$  la regola  $A \rightarrow B$  e  $\varepsilon \in L_B$ ;
- $x = ay$  inizia con il carattere  $a$ , vi è in  $P$  la regola  $A \rightarrow aB$  e la stringa  $y \in \Sigma^*$  appartiene al linguaggio  $L_B$ .

Sia  $n = |V|$  il numero dei nonterminali. Ogni nonterminale  $A_i$  è definito dalle alternative

$$A_i \rightarrow a_1 A_1 \mid \dots \mid a_n A_n \mid A_1 \mid \dots \mid A_n \mid \varepsilon$$

alcune delle quali possono mancare. Si scrive allora l'equazione:

$$L_{A_i} = a_1 L_{A_1} \cup \dots \cup a_n L_{A_n} \cup L_{A_1} \cup \dots \cup L_{A_n} \cup \varepsilon$$

L'ultimo termine scompare se manca l'alternativa  $A_i \rightarrow \varepsilon$ .

Questo sistema ha  $n$  equazioni con  $n$  incognite (i linguaggi generati dai corrispettivi nonterminali) e può essere risolto con il noto metodo di sostituzione o eliminazione gaussiana, applicando la seguente formula.

*Proprietà 2.72.* Identità di Arden

L'equazione

$$X = KX \cup L \tag{2.7}$$

dove  $K$  è un linguaggio non vuoto e  $L$  un linguaggio qualsiasi, ha una e una sola soluzione

$$X = K^* L \tag{2.8}$$

È chiaro che il linguaggio  $K^* L$  è soluzione della 2.7 poiché, sostituendolo a sinistra e a destra nell'equazione, si ottiene l'identità

$$K^* L = K K^* L \cup L$$

Viceversa si potrebbe dimostrare che la 2.7 non ha altre soluzioni al di fuori della 2.8.

*Esempio 2.73.* Equazioni insiemistiche

La grammatica

$$S \rightarrow sS \mid eA \quad A \rightarrow sS \mid \varepsilon$$

genera una lista di elementi (anche mancanti) e separati dal separatore  $s$ . Essa si trasforma nel sistema di equazioni:

$$\begin{cases} L_S &= sL_S \cup eL_A \\ L_A &= sL_S \cup \epsilon \end{cases}$$

Si sostituisce la seconda equazione nella prima

$$\begin{cases} L_S &= sL_S \cup e(sL_S \cup \epsilon) \\ L_A &= sL_S \cup \epsilon \end{cases}$$

poi (applicando la proprietà distributiva del concatenamento rispetto all'unione) si raccoglie  $L_S$  a suffisso comune

$$\begin{cases} L_S &= (s \cup es)L_S \cup e \\ L_A &= sL_S \cup \epsilon \end{cases}$$

e si risolve con l'identità di Arden la prima equazione:

$$\begin{cases} L_S &= (s \cup es)^*e \\ L_A &= sL_S \cup \epsilon \end{cases}$$

da cui anche  $L_A = s(s \cup es)^*e \cup \epsilon$ .

Si noti che è altrettanto agevole scrivere le equazioni per una grammatica in forma unilineare generica.

Un altro metodo per calcolare l'e.r. del linguaggio definito da una grammatica unilineare, passando attraverso un automa, sarà esposto nel capitolo 3.  
In conclusione si può affermare che ogni linguaggio unilineare è regolare.

## 2.7 Linguaggi regolari e liberi a confronto

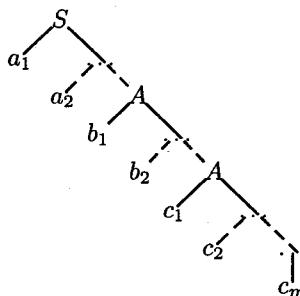
È importante comprendere i limiti della famiglia dei linguaggi regolari, allo scopo di decidere quali costrutti linguistici siano modellabili con le espressioni regolari e quali invece necessitino l'intera potenza delle grammatiche libere. Si vedrà ora un'importante proprietà strutturale della famiglia *REG*. Si sa (proprietà 2.37, p. 39) che, solo grazie alla ricorsione, una grammatica può generare un linguaggio infinito: una derivazione ricorsiva  $A \stackrel{+}{\Rightarrow} uAv$  può essere iterata un numero  $n$  di volte, producendo la stringa  $u^nAv^n$ . D'altra parte, ogni frase, purché abbastanza lunga, contiene necessariamente almeno una derivazione ricorsiva, quindi contiene certe sottostringhe che possono essere ripetute qualsiasi numero di volte.

Si formula ora quest'osservazione in modo più preciso, prima per le grammatiche unilineari, poi per quelle libere.

### Proprietà 2.74. Pompaggio

Sia data una grammatica unilineare  $G$ . Ogni sua frase  $x$  sufficientemente lunga, ossia di lunghezza superiore a una costante dipendente da  $G$ , può essere fattorizzata come  $x = tuv$ , dove la stringa  $u$  non è vuota, in modo tale che, per ogni  $n \geq 0$ , la stringa  $tu^n v$  appartiene al linguaggio. (Si dice che la frase può essere 'pompata' iniettando un numero arbitrario di volte la stringa  $u$ ).

Dimostrazione. Si consideri una grammatica strettamente lineare a destra avente  $k$  simboli nonterminali. Nell'albero d'una frase  $x$ , lunga  $k$  o più caratteri, vi è necessariamente un nonterminale  $A$  che compare almeno due volte



dove le tre sottostringhe della fattorizzazione sono:  $t = a_1 a_2 \dots$ ,  $u = b_1 b_2 \dots$  e  $v = c_1 c_2 \dots c_m$ . Esiste pertanto la derivazione ricorsiva:

$$S \stackrel{+}{\Rightarrow} tA \stackrel{+}{\Rightarrow} tuA \stackrel{+}{\Rightarrow} tuv$$

che permette di derivare anche le stringhe  $tuuv$  e  $tv$ .

Questa proprietà può essere sfruttata per dimostrare che certi linguaggi non sono regolari.

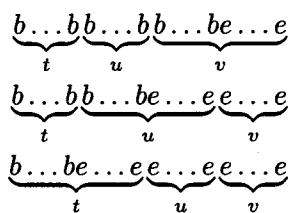
Un costrutto frequente nei linguaggi artificiali è quello che impone che due elementi  $begin \equiv b$  e  $end \equiv e$  compaiano in questo ordine e abbiano esattamente lo stesso numero di comparse.

*Esempio 2.75.* Linguaggio a due esponenti eguali.

Preso il linguaggio libero

$$L_1 = \{b^n e^n \mid n \geq 1\}$$

si supponga per assurdo che esso sia regolare. Presa una frase  $x = b^k e^k$ , con  $k$  abbastanza grande, la si suddivida in tre sottostringhe,  $x = tuv$ , con  $u$  non vuota. A seconda della posizione in cui cadono le divisioni, le stringhe  $t$ ,  $u$  e  $v$  possono essere così schematizzate:



Nel primo caso, iterando due volte la  $u$ , il numero delle  $b$  eccede quello delle  $e$ . Nel secondo, iterando due volte la  $u$ , la stringa  $tuuv$  contiene due sottostringhe  $be$ . Nel terzo, iterando la  $u$ , il numero delle  $e$  eccede quello delle  $b$ . In tutti i casi la proprietà 2.74 è contraddetta poiché le stringhe pommate non appartengano al linguaggio. Si è così dimostrato che questo linguaggio libero non è regolare.

L'esempio completa la dimostrazione dell'inclusione stretta delle due famiglie:

*Proprietà 2.76.* Ogni linguaggio regolare è libero, ma esistono linguaggi liberi che non sono regolari.

Che un esempio così semplice e fondamentale non sia regolare mostra l'insufficienza della famiglia regolare per modellare certe strutture sintattiche dei linguaggi tecnici. I linguaggi regolari non sono però inutili, perché descrivono molto efficacemente le parti più semplici e frequenti dei linguaggi tecnici: da un lato le sottostringhe che si collocano al livello detto lessicale (ad es. le costanti numeriche o gli identificatori), dall'altro molti costrutti che sono varianti delle liste, come ad es. le liste di parametri o le sequenze di istruzioni.

### Ruolo delle derivazioni autoincassate

Avendo assodato che i linguaggi regolari sono una famiglia più ristretta di quella dei linguaggi liberi, conviene porre l'attenzione alle particolarità che rendono non regolari certi linguaggi tipici, come ad es. i palindromi, il linguaggio di Dyck o il linguaggio con due esponenti eguali. Una più attenta osservazione rivela che le grammatiche di questi linguaggi presentano derivazioni ricorsive del tipo *autoincassato* (o autoinclusivo)

$$A \stackrel{+}{\Rightarrow} uAv \quad u \neq \varepsilon \wedge v \neq \varepsilon$$

Al contrario tali derivazioni sono assenti dalle grammatiche unilineari le cui ricorsioni, come osservato, stanno da un solo lato.

Ora è proprio l'assenza di derivazioni autoincassate che permette di risolvere le equazioni insiemistiche associate alle grammatiche unilineari, e conferisce ai linguaggi corrispondenti la struttura particolarmente semplice che si conosce. La maggiore potenza generativa delle grammatiche libere rispetto a quelle unilineari dipende essenzialmente dalla presenza di derivazioni autoincassate, come dice il seguente enunciato.

*Proprietà 2.77.* Una grammatica libera priva di derivazioni autoinclusive genera un linguaggio regolare.

*Esempio 2.78.* Grammatica non autoincassata.

La grammatica  $G$ :

$$S \rightarrow AS \mid bA \quad A \rightarrow aA \mid \varepsilon$$

pur non unilineare, non permette derivazioni autoincassate e  $L(G)$  è regolare, come si vede risolvendo le equazioni insiemistiche

$$\begin{cases} L_S = L_A L_S \cup b L_A \\ L_A = a L_A \cup \varepsilon \end{cases}$$

$$\begin{cases} L_S = L_A L_S \cup b L_A \\ L_A = a^* \end{cases}$$

$$L_S = a^* L_S \cup ba^*$$

$$L_S = (a^*)^* ba^*$$

### Linguaggi liberi di alfabeto unario

Non è vero il viceversa della precedente proprietà: infatti, pur in presenza di derivazioni autoincassate, il linguaggio generato può risultare in casi particolari regolare. Illustrando questo fatto, si coglie l'occasione per enunciare una curiosa proprietà dei linguaggi liberi aventi un alfabeto d'una sola lettera.

*Proprietà 2.79.* Il linguaggio definito da una grammatica libera, il cui alfabeto terminale  $\Sigma$  è unario,  $|\Sigma| = 1$  è regolare.

Si noti che le frasi  $x$  d'un linguaggio di alfabeto unario sono in corrispondenza biunivoca con i numeri interi, tramite  $x \leftrightarrow n$ , se e solo se  $|x| = n$ .

*Esempio 2.80.* La grammatica

$$G = \{S \rightarrow aSa \mid \varepsilon\}$$

ha la derivazione autoinclusiva  $S \Rightarrow aSa$ , ma  $L(G) = (aa)^*$  è regolare, e può essere definito con la grammatica lineare a destra  $\{S \rightarrow aaS \mid \varepsilon\}$ , ottenuta spostando a suffisso il nonterminale presente nel mezzo della prima regola.

#### 2.7.1 Limiti dei linguaggi liberi dal contesto

Passando ai linguaggi liberi, anche per essi si può vedere che le frasi abbastanza lunghe contengono necessariamente due sottostringhe che possono essere ripetute quante volte si vuole. Le ripetizioni danno luogo questa volta a strutture autoincassate. Tale proprietà impedisce alle grammatiche libere di generare i costrutti in cui tre o più parti sono ripetute lo stesso numero di volte.

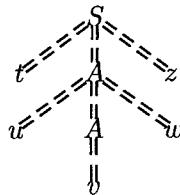
*Proprietà 2.81.* Linguaggio a tre esponenti.

Il linguaggio

$$L = \{a^n b^n c^n \mid n \geq 1\}$$

non è libero.

*Dimostrazione.* Per assurdo, si supponga che esista una grammatica  $G$  di  $L$ . Si immagini l'albero sintattico d'una frase  $x = a^n b^n c^n$ . In esso fissiamo l'attenzione sui cammini che vanno dalla radice (assioma  $S$ ) alle foglie terminali. Almeno uno dei cammini avrà lunghezza crescente con la lunghezza della frase  $x$ , e, poiché  $n$  può essere scelto grande a piacere, tale cammino attraverserà per forza due nodi con lo stesso simbolo nonterminale, diciasi  $A$ . La situazione è illustrata nello schema:



dove  $t, u, v, w, z$  sono stringhe terminali. Lo schema denota la derivazione:

$$S \stackrel{+}{\Rightarrow} tAz \stackrel{+}{\Rightarrow} t uAwz \stackrel{+}{\Rightarrow} tuvwz$$

Esiste dunque nella derivazione una sotto derivazione ricorsiva da  $A$  ad  $A$ , la quale potrà essere ripetuta un numero qualsiasi  $j$  di volte, producendo le stringhe del tipo

$$y = t \underbrace{u \dots u}_j v \underbrace{w \dots w}_j z$$

Si esaminano i casi possibili per le stringhe  $u, w$ :

- Entrambe le stringhe contengono soltanto una lettera, la stessa, dicasi  $a$ ; dunque, al crescere di  $j$ , la stringa  $y$  ha un numero di  $a$  che eccede il numero delle  $b$ , e non appartiene al linguaggio.
- La stringa  $u$  contiene almeno due lettere diverse, ad es.  $u = \dots a \dots b \dots$ . Allora, ripetendo due volte la derivazione ricorsiva, si ottiene  $uu = \dots a \dots b \dots a \dots b \dots$ , dove le lettere  $a$  e  $b$  sono mescolate, e quindi  $y$  esce dal linguaggio.  
Analogo è il caso in cui è la stringa  $w$  a contenere due lettere diverse.
- La stringa  $u$  contiene solo una lettera, dicasi  $a$  e la  $w$  solo una lettera diversa, dicasi  $b$ . Al crescere di  $j$ , la stringa  $y$  contiene un numero di  $a$  maggiore del numero delle  $b$ , quindi non è nel linguaggio.

Il ragionamento precedente, che sfrutta la possibilità di pompare le frasi del linguaggio mediante la ripetizione di una derivazione ricorsiva, è un valido strumento concettuale per dimostrare che certi linguaggi non appartengono alla famiglia LIB.

Certo il linguaggio a tre esponenti non ha rilevanza pratica, ma il prossimo caso mostra che le grammatiche libere non sono abbastanza potenti per definire certi costrutti tipici dei linguaggi tecnici.

### *Linguaggio delle repliche*

Una figura sintattica importante è la *replica*, che spesso si incontra nei linguaggi tecnici, ognqualvolta gli elementi di due liste devono essere in qualche relazione di corrispondenza. Si pensi ad es. alla concordanza tra la lista dei parametri formali e quella dei parametri attuali di un sottoprogramma; o per esemplificare con l'italiano, alla frase *gatti, rane, farfalle sono rispettivamente mammiferi, batraci, insetti.*

Nella forma più astratta, in cui le due liste sono fatte con gli stessi terminali, la replica è il linguaggio

$$L_{\text{replica}} = \{uu \mid u \in \Sigma^+\}$$

Sia  $\Sigma = \{a, b\}$ . Una frase, come  $x = abbbabbb = uu$ , è in un certo senso duale del palindromo  $y = abbbbbba = uu^R$ , dove la stringa  $u$  è specularmente rovesciata invece che replicata. La simmetria delle frasi  $L_{\text{replica}}$  è traslatoria, non speculare. Mentre il linguaggio dei palindromi è tra i più semplici linguaggi liberi, quello delle repliche non è libero. Ciò deriva dal fatto che per controllare la simmetria occorre una pila (LIFO last in first out) nel caso speculare, una coda (FIFO first in first out) nel caso traslatorio, e, come vedremo nel capitolo 4 gli algoritmi che riconoscono i linguaggi liberi sono dotati d'una pila di memoria, non d'una coda.

Per dimostrare che la replica non è libera, si sfrutta ancora il ragionamento del pompaggio, ma dopo avere filtrato il linguaggio in modo da renderlo simile a quello a tre esponenti.

Si osservi che un sottolinguaggio di  $L_{\text{replica}}$ , ottenuto mediante intersezione con un linguaggio regolare, è il linguaggio

$$L_{abab} = \{a^m b^n a^m b^n \mid m, n \geq 1\} = L_{\text{replica}} \cap a^+ b^+ a^+ b^+$$

Anticipando che l'intersezione d'un linguaggio libero con uno regolare è un linguaggio libero (come si dimostrerà a p. 158), basta dimostrare che  $L_{abab}$  non è libero, al fine di concludere che  $L_{\text{replica}}$  non è libero. Il ragionamento da fare è del tutto analogo a quello sviluppato nella prova che il linguaggio a tre esponenti (p. 76) non è libero.

### 2.7.2 Proprietà di chiusura di REG e LIB

Le operazioni linguistiche e insiemistiche combinano tra loro i linguaggi, ottenendone altri. Ciò può servire a estendere un linguaggio, a ridurlo, eliminando certe stringhe, o a modificarlo cambiando i caratteri terminali. Non tutte le operazioni però preservano l'appartenenza del linguaggio risultante alla stessa famiglia dei linguaggi cui l'operazione è applicata. Quando ciò non accade, il linguaggio risultante non potrà essere definito con lo stesso tipo di grammatica dei linguaggi operandi.

Concludendo il confronto tra linguaggi regolari e liberi, si completa ora il quadro delle loro proprietà di chiusura rispetto alle operazioni più comuni. Alcune proprietà sono state già presentate, altre sono di immediata giustificazione, altre ancora dovranno attendere i metodi della teoria degli automi per essere giustificate.

Si indicano con *LIB* e *REG* le due famiglie e con *L* e *R* un generico linguaggio libero o regolare. La tabella riassume le principali proprietà di chiusura:

| riflessione   | stella        | unione o concatenamento  | complemento         | intersezione                                    |
|---------------|---------------|--------------------------|---------------------|---|
| $R^R \in REG$ | $R^* \in REG$ | $R_1 \oplus R_2 \in REG$ | $\neg R \in REG$    | $R_1 \cap R_2 \in REG$                          |
| $L^R \in LIB$ | $L^* \in LIB$ | $L_1 \oplus L_2 \in LIB$ | $\neg L \notin LIB$ | $L_1 \cap L_2 \notin LIB$<br>$L \cap R \in LIB$ |

Commenti e esempi.

- Dove la tabella indica una non appartenenza (ad es.  $\neg L \notin LIB$ ) si deve intendere che esistono dei linguaggi che non appartengono alla famiglia (non già che il complemento di ogni linguaggio libero non sia libero).
- Il linguaggio riflesso di  $L(G)$  è generato dalla *grammatica riflessa*, quella ottenuta riflettendo specularmente le parti destre delle regole. Chiaramente, se la grammatica  $G$  è lineare a destra, la riflessa è lineare a sinistra e definisce ancora un linguaggio regolare.
- La stella, l'unione e il concatenamento di linguaggi liberi sono liberi. Siano  $G_1$  e  $G_2$  le grammatiche di  $L_1$  e  $L_2$ , siano  $S_1$  e  $S_2$  i loro assiomi, e si supponga, senza perdita di generalità, che i loro alfabeti nonterminali siano disgiunti,  $V_1 \cap V_2 = \emptyset$ . Per i tre casi si aggiungono a  $G_1$  e  $G_2$  le seguenti regole iniziali:

$$\begin{array}{ll} \text{Stella:} & S \rightarrow SS_1 \mid \varepsilon \\ \text{Unione:} & S \rightarrow S_1 \mid S_2 \\ \text{Concatenamento:} & S \rightarrow S_1 S_2 \end{array}$$

Per l'unione, se le due grammatiche sono lineari a destra, lo è anche la grammatica costruita. Al contrario le nuove regole introdotte per il concatenamento e la stella non sono lineari a destra, ma si possono anche scrivere grammatiche equivalenti e lineari a destra, poiché si sa (proprietà 2.23 p. 24) che la famiglia  $REG$  è chiusa rispetto a dette operazioni.

- La dimostrazione che il complemento di un linguaggio regolare è regolare si troverà nel capitolo 3 (p.136).
- L'intersezione di linguaggi libri non è in generale libera, come testimonia il noto linguaggio non libero con tre esponenti eguali (es. 2.81 di p. 76)

$$\{a^n b^n c^n \mid n \geq 1\} = \{a^n b^n c^+ \mid n \geq 1\} \cap \{a^+ b^n c^n \mid n \geq 1\}$$

dove i due linguaggi operandi sono facilmente definiti da grammatiche libere.

- Come conseguenza anche il complemento d'un linguaggio libero non è in generale libero. Infatti dall'identità di De Morgan  $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$ , se il complemento di un linguaggio libero fosse libero, essendo libera l'unione di due linguaggi libri, si avrebbe un assurdo.
- Invece l'intersezione d'un linguaggio libero con uno regolare risulta libera. Si rinvia a p. 158 la giustificazione, che richiede nozioni di teoria degli automi.

L'ultima proprietà ha utilità per rendere più selettiva una grammatica, **filtrandola** attraverso un linguaggio regolare che elimina certe frasi.

*Esempio 2.82.* Filtri regolari sul linguaggio di Dyck.

È interessante studiare come il linguaggio di Dyck (p. 44)  $L_D$  di alfabeto  $\Sigma = \{a, a'\}$  si restringe, intersecandolo con due linguaggi regolari:

$$\begin{aligned} L_1 &= L_D \cap \neg(\Sigma^* a' a \Sigma^*) = (aa')^* \\ L_2 &= L_D \cap \neg(\Sigma^* a' a \Sigma^*) = \{a^n a'^n \mid n \geq 0\} \end{aligned}$$

La prima intersezione filtra il linguaggio di Dyck riducendolo alle frasi prive della sottostringa  $a' a'$ , ossia prive di parentesi annidate. Il secondo filtro riduce il linguaggio alle frasi aventi un solo nido di parentesi. Entrambi sono linguaggi liberi, ma il primo è anche regolare.

### 2.7.3 Trasformazioni alfabetiche

Spesso si incontrano linguaggi tra loro molto simili che differiscono soltanto nella scelta di alcuni simboli terminali. Ad es. la moltiplicazione aritmetica può essere rappresentata in un linguaggio dal segno per, in un altro dall'asterisco oppure dal puntino.

→ Si chiama *traslitterazione o omomorfismo alfabetico* l'operazione linguistica che sostituisce a certi caratteri terminali altri caratteri.

**Definizione 2.83.** *Traslitterazione alfabetica (o omomorfismo)*<sup>23</sup>

*Siano dati i due alfabeti sorgente  $\Sigma$  e pozzo  $\Delta$ . Una traslitterazione alfabetica è una funzione:*

$$h : \Sigma \rightarrow \Delta \cup \{\varepsilon\}$$

*La traslitterazione o immagine omomorfa del carattere  $c \in \Sigma$  è data da  $h(c)$ . Se  $h(c) = \varepsilon$ , il carattere  $c$  è cancellato. La traslitterazione è priva di cancellature se, per ogni carattere sorgente  $c$ , risulta  $h(c) \neq \varepsilon$ .*

*La traslitterazione d'una stringa sorgente  $a_1 a_2 \dots a_n, a_i \in \Sigma$  è la stringa  $h(a_1)h(a_2)\dots h(a_n)$  prodotta dal concatenamento delle immagini dei singoli caratteri. La stringa vuota si trasforma nella stringa vuota.*

Di conseguenza vale la proprietà compositiva: la traslitterazione del concatenamento di due stringhe  $v$  e  $w$  è il concatenamento delle traslitterazioni individuali:

$$h(v.w) = h(v).h(w)$$

*Esempio 2.84.* Stampante.

Una stampante antiquata non possiede i caratteri greci, e quando deve stampare un testo, rimpiazza un carattere greco con il carattere speciale,  $\square$ . Inoltre il testo inviato alla stampante contiene dei caratteri di controllo (ad es. `start-text`, `end-text`) che non vengono stampati. La trasformazione del testo (trascrivendo le lettere maiuscole) è descritta dalla traslitterazione:

---

<sup>23</sup>Caso particolare delle funzioni di traduzione argomento del cap. 5.

$$\begin{aligned}
 h(c) &= c && \text{se } c \in \{a, b, \dots, z, 0, 1, \dots, 9\}; \\
 h(c) &= c && \text{se } c \text{ è un segno di punteggiatura o uno spazio bianco;} \\
 h(c) &= \square && \text{se } c \in \{\alpha, \beta, \dots, \omega\}; \\
 h(\text{start-text}) &= h(\text{end-text}) = \varepsilon.
 \end{aligned}$$

Un esempio di traslitterazione è:

$$\underbrace{h(\text{start-text} \text{ la cost. } \pi \text{ vale } 3.14 \text{ end-text})}_{\text{stringa sorgente}} = \underbrace{\text{la cost. } \square \text{ vale } 3.14}_{\text{stringa pozzo}}$$

Un caso speciale di omomorfismo alfabetico con cancellatura è la *proiezione*: essa è una funzione che cancella certi caratteri sorgente e lascia tutti gli altri invariati.

### *Traslitterazione a parole*

La traslitterazione è qualificata come *alfabetica* poiché l'immagine di ogni carattere dell'alfabeto sorgente è un carattere dell'alfabeto pozzo (o la stringa vuota). L'aggettivo cade, se nella traslitterazione l'immagine d'un carattere sorgente è una stringa non unitaria dell'alfabeto pozzo. Un esempio è la trasformazione di un'istruzione di assegnamento  $a \leftarrow b + c$  nella forma  $a := b + c$  tramite la traslitterazione (non alfabetica):

$$h(\leftarrow) = ' :=' \quad h(c) = c, \text{ per ogni altro } c \in \Sigma$$

Tali traslitterazioni sono dette *a parole*.

Un secondo esempio: le vocali con dieresi dell'alfabeto tedesco si traducono in stringhe di due caratteri, mediante la corrispondenza:

$$h(\ddot{a}) = ae, \quad h(\ddot{o}) = oe, \quad h(\ddot{u}) = ue$$

### *Sostituzione di linguaggio*

Generalizzando, si passa dalla traslitterazione a un'altra trasformazione, la *sostituzione* (introdotta nell'astrazione linguistica di p. 27), che a un carattere sorgente sostituisce un linguaggio specificato. La sostituzione è assai sfruttata in fase di progetto di un linguaggio, quando nella grammatica si lascia in sospeso un costrutto, designandolo con un simbolo (ad es. `(identificatore)`). Al completamento del progetto, il simbolo sarà sostituito dalla definizione del linguaggio corrispondente (ad es.  $(a \dots z)((a \dots z \mid 0 \dots 9)^*)$ ).

Formalmente, dato l'alfabeto sorgente  $\Sigma = \{a, b, \dots\}$ , una sostituzione  $h$  associa a ogni lettera un linguaggio  $h(a) = L_a, h(b) = L_b, \dots$  di alfabeto pozzo  $\Delta$ . L'applicazione della sostituzione  $h$  a una stringa sorgente  $a_1 a_2 \dots a_n, a_i \in \Sigma$  produce un insieme di stringhe, così calcolato:

$$h(a_1 a_2 \dots a_n) = \{y_1 y_2 \dots y_n \mid y_i \in L_{a_i}\}$$

In restrospettiva, una traslitterazione a parole è una sostituzione in cui ogni linguaggio immagine contiene una sola stringa; se poi la stringa è di lunghezza uno o zero, la traslitterazione è alfabetica.

### Chiusura rispetto a trasformazioni d'alfabeto

Sia  $L$  un linguaggio, libero o regolare e  $h$  una sostituzione, avente come immagine dei linguaggi della stessa famiglia. L'insieme delle stringhe immagine delle frasi di  $L$ , detto il linguaggio *immagine* o *pozzo*  $L' = h(L)$ , è un linguaggio della stessa famiglia? La risposta è positiva, come si mostrerà ora in modo costruttivo. Tale costruzione ha utilità pratica per chi deve modificare la definizione d'un linguaggio, perché consente di ottenere con minore sforzo la grammatica o l'espressione regolare del nuovo linguaggio, editando quella del vecchio.

→ **Proprietà 2.85.** La famiglia  $LIB$  è chiusa rispetto alla sostituzione di linguaggi della stessa famiglia (e quindi anche rispetto alla traslitterazione).

*Dimostrazione.* Poiché la traslitterazione a parole è un caso speciale di sostituzione, in cui ogni linguaggio inserito ha una sola frase, basta dimostrare l'enunciato per la sostituzione. Sia  $G$  la grammatica libera di  $L$  e  $h$  una sostituzione tale che, per ogni  $c \in \Sigma$ ,  $L_c$  sia un linguaggio libero, che si suppone definito dalla grammatica  $G_c$  di assioma  $S_c$ . Si suppone anche che gli alfabeti nonterminali delle grammatiche  $G, G_a, G_b, \dots$  siano disgiunti a coppie (altrimenti è sufficiente ridenominare i nonterminali).

Si mostra come ottenere la grammatica  $G'$  del linguaggio  $h(L)$ , applicando alle regole di  $G$  la traslitterazione alfabetica  $f$  così definita:

$$\begin{aligned} f(c) &= S_c, \text{ per ogni terminale } c \in \Sigma; \\ f(A) &= A, \text{ per ogni simbolo nonterminale } A \text{ di } G. \end{aligned}$$

La grammatica  $G'$  è così costruita:

- a ciascuna regola  $A \rightarrow \alpha$  di  $G$  si applica la traslitterazione  $f$ , che ha l'effetto di cambiare ogni carattere terminale nell'assioma della grammatica pozzo corrispondente;
- alle regole così prodotte si aggiungono quelle delle grammatiche  $G_a, G_b, \dots$

Chiaramente la grammatica così ottenuta genera il linguaggio  $h(L(G))$ .

La costruzione della grammatica  $G'$  si semplifica nel caso d'una traslitterazione. Basta sostituire a ogni carattere terminale  $c \in \Sigma$  della grammatica  $G$  l'immagine  $h(c)$ .

Per i linguaggi regolari vale la stessa proprietà.

→ **Proprietà 2.86.** La famiglia  $REG$  è chiusa rispetto alla sostituzione (e quindi anche rispetto alla traslitterazione) di linguaggi della stessa famiglia.

Lo stesso ragionamento impiegato nel caso dei linguaggi liberi può essere applicato all'espressione regolare del linguaggio sorgente, per produrre quella del linguaggio immagine.

*Esempio 2.87.* Grammatica traslitterata.

Il linguaggio iniziale  $i(i)^*$  definito dalle regole

$$S \rightarrow i; S | i$$

schematizza un programma composto da una serie di istruzioni  $i$  separate dal punto e virgola. Volendo esplodere le  $i$  nelle istruzioni d'assegnamento, si usa la traslitterazione a parole

$$g(i) = v \leftarrow e$$

dove  $v$  è una variabile e  $e$  un'espressione. La grammatica diviene:

$$S \rightarrow A; S | A \quad A \rightarrow v \leftarrow e$$

Per esplodere la definizione delle espressioni, si usa poi la sostituzione  $h(e) = L_E$ , dove il linguaggio è quello delle espressioni aritmetiche più volte presentato. Se esso è generato da una grammatica di assioma  $E$ , basta la seguente trasformazione per ottenere la grammatica del nuovo linguaggio:

$$S \rightarrow A; S | A \quad A \rightarrow v \leftarrow E \quad E \rightarrow \dots$$

Infine si potrà sostituire il simbolo  $v$  delle variabili con il linguaggio regolare degli identificatori, e così via.

#### 2.7.4 Grammatiche libere estese con espressioni regolari

Le espressioni regolari, pur essendo un formalismo meno potente delle grammatiche libere, risultano spesso più leggibili, grazie agli operatori di iterazione (stella e croce) e di scelta (unione). Per agevolare la comprensione e la concisione delle grammatiche, soprattutto nelle definizioni di linguaggi alquanto complessi, si consente l'uso di espressioni regolari nelle parti destre delle regole. Tali *grammatiche libere estese* o *EBNF*<sup>24</sup> sono preferite dai manuali dei linguaggi tecnici. Le loro regole possiedono una forma grafica suggestiva, i diagrammi sintattici, che saranno visti nel capitolo 4 come schemi di flusso degli algoritmi di analisi sintattica.

Poiché la famiglia *LIB* è chiusa rispetto alle operazioni regolari, la famiglia dei linguaggi definiti dalle grammatiche EBNF coincide con *LIB*.

Al fine di apprezzare la chiarezza delle grammatiche EBNF rispetto a quelle di base, si confrontano le definizioni di alcuni costrutti tipici dei linguaggi programmativi.

*Esempio 2.88.* Grammatica EBNF di linguaggio a blocchi: dichiarazioni.

Si consideri una lista di dichiarazioni di variabili:

*char testol, testo2; real temp, risult; int alfa, beta2, gamma;*

---

<sup>24</sup>Extended BNF.

quale si incontra con piccole varianti in molti linguaggi programmativi. Preso l'alfabeto  $\Sigma = \{c, i, r, v, ',', ';' \}$ , dove  $c, i, r$  stanno per *char*, *int*, *real* e  $v$  per un nome di variabile, il linguaggio delle liste di dichiarazioni è definito dalla espressione regolare  $D$ :

$$((c \mid i \mid r)v(v)^*;)^+$$

Gli operatori di iterazione sono vantaggiosi per generare le liste, ma non indispensabili: infatti sappiamo che ogni linguaggio regolare può essere generato da una grammatica libera (addirittura unilineare).

Le liste sono definite dalla grammatica:

$$D \rightarrow DE \mid E \quad E \rightarrow AN; \quad A \rightarrow c \mid i \mid r \quad N \rightarrow v, N \mid v$$

che presenta due regole ricorsive (la  $D$  e la  $N$ ). Oltre a essere più lunga, la grammatica riduce l'evidenza della struttura a due liste gerarchiche, che nella e.r. è palese. Inoltre vi è un'arbitrarietà nella scelta dei metasimboli  $A, E, N$  che può causare qualche inutile confusione, quando si raffrontano o combinano regole scritte da persone diverse, magari in più lingue nazionali.

**Definizione 2.89.** Una grammatica libera (o BNF) estesa  $G = (V, \Sigma, P, S)$  contiene esattamente  $|V|$  regole, ognuna nella forma  $A \rightarrow \eta$ , dove  $\eta$  è un'espressione regolare d'alfabeto  $V \cup \Sigma$ .

Per maggiore leggibilità o concisione, anche gli altri operatori derivati (croce, elevamento a potenza, opzionalità) possono essere utilizzati.

Il prossimo esempio completa il precedente, con l'aggiunta delle tipiche strutture a blocchi d'un linguaggio programmatico.

**Esempio 2.90.** Linguaggio simil-Algol.

Un blocco  $B$  si compone d'una parte dichiarativa (facoltativa)  $D$ , seguita da una parte imperativa  $I$ , racchiusa tra  $b$  (begin) e  $e$  (end):

$$B \rightarrow b[D]Ie$$

La parte dichiarativa  $D$  è come nell'esempio precedente:

$$D \rightarrow ((c \mid i \mid r)v(v)^*;)^+$$

La parte imperativa  $I$  è una lista di frasi  $F$  separate dal  $'.'$ :

$$I \rightarrow F(F)^*$$

Infine, una frase  $F$  è un assegnamento  $a$  oppure un blocco  $B$ :

$$F \rightarrow a \mid B$$

Come esercizio, anche se peggiorando la leggibilità, si può rendere la grammatica più compatta, sostituendo a  $D$  e  $I$  le loro definizioni:

$$B \rightarrow b[((c \mid i \mid r)v(v)^*;)^+]F(F)^*e$$

semplificabile in:

$$B \rightarrow b((c \mid i \mid r)v(v)^*;)^*F(F)^*e$$

Infine eliminando  $F$  si ottiene una grammatica  $G'$  fatta d'una sola regola:

$$B \rightarrow b((c \mid i \mid r)v(v)^*;)^*(a \mid B)(;(a \mid B))^*e$$

Ma da essa non si può ottenere una e.r. di alfabeto puramente terminale, perché il nonterminale  $B$  è ineliminabile, essendo necessario per la generazione dei blocchi annidati ( $bb\dots ee$ ), costrutto tipicamente non regolare, che necessita di derivazioni ricorsive autoincassate (in accordo con la proprietà 2.77, p. 75).

Nei manuali di riferimento dei linguaggi tecnici si preferiscono le regole estese. Tuttavia non conviene eccedere nella compattezza delle regole per non incorrere in oscurità. Infine spesso la modularizzazione della grammatica in più regole agevola l'assegnazione di semplici azioni semantiche a ciascuna di esse, come si vedrà nel capitolo 5.

### Derivazioni e alberi nelle grammatiche libere estese

La parte destra  $\alpha$  d'una regola estesa  $A \rightarrow \alpha$  di  $G$  è una e.r. che definisce un insieme in generale illimitato di stringhe. Esse possono essere pensate come le parti destre di un'insolita grammatica  $G'$ , equivalente a  $G$ , ma avente un numero illimitato di regole.

Ad es.  $A \rightarrow (aB)^+$  sta per l'insieme di regole

$$A \rightarrow aB \mid aBaB \mid \dots$$

La relazione di derivazione si può definire anche per le grammatiche estese, appoggiandosi al concetto di derivazione per le e.r., introdotto a p. 20.

Date le stringhe  $\eta_1, \eta_2 \in (\Sigma \cup V)^*$  si dice che  $\eta_1$  deriva (immediatamente)  $\eta_2$ , scritto  $\eta_1 \Rightarrow \eta_2$ , se le due stringhe si fattorizzano come:

$$\eta_1 = \alpha A \gamma, \quad \eta_2 = \alpha \vartheta \gamma$$

e esiste una regola  $A \rightarrow e$  tale che la e.r.  $e$  deriva in zero o più passi la stringa  $\vartheta$ :

$$e \xrightarrow{*} \vartheta \quad (\text{derivazione per e.r.})$$

Si noti che le stringhe  $\eta_1, \eta_2$  (e quindi anche  $\alpha \vartheta \gamma$ ) non contengono gli operatori delle e.r., né le parentesi; al contrario la stringa  $e$  è una e.r.

La derivazione precedente è sinistra se  $A$  è il nonterminale posto più a sinistra in  $\eta_1$ .

In modo naturale si potrebbero poi definire le derivazioni a più passi, e il linguaggio generato da una grammatica estesa, ma per brevità ci si limita all'esemplificazione.

*Esempio 2.91.* Derivazione estesa per espressioni aritmetiche.

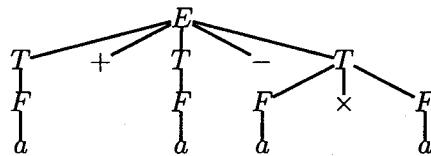
La grammatica estesa  $G$ :

$$E \rightarrow [+ | -]T((+ | -)T)^* \quad T \rightarrow F((\times | /)F)^* \quad F \rightarrow (a | '('E')')$$

genera le espressioni aritmetiche con i quattro operatori infissi, gli operatori  $\pm$  prefissi, le parentesi tonde e la variabile  $a$ . Le parentesi quadre racchiudono un costrutto opzionale. La derivazione sinistra

$$\begin{aligned} E &\Rightarrow T + T - T \Rightarrow F + T - T \Rightarrow a + T - T \Rightarrow a + F - T \Rightarrow a + a - T \Rightarrow \\ &\Rightarrow a + a - F \times F \Rightarrow a + a - a \times F \Rightarrow a + a - a \times a \end{aligned}$$

produce l'albero sintattico:



Si vede che il grado (cioè il numero di figli) d'un nodo risulta in generale illimitato negli alberi d'una grammatica EBNF. In tal modo, allargandosi l'albero, la sua altezza si riduce rispetto a quella d'una grammatica equivalente non estesa.

### Ambiguità nelle grammatiche estese

Una grammatica libera non estesa ambigua è ovviamente ambigua anche come grammatica estesa. Ma l'ambiguità può sorgere nelle grammatiche estese in un altro modo, peculiare delle e.r. Si ricorda da p. 22 che una e.r. è ambigua se nel linguaggio da essa definito vi è una frase generata con due derivazioni sinistre diverse.

Così la e.r.  $a^*b \mid ab^*$ , numerata come  $a_1^*b_2 \mid a_3b_4^*$ , è ambigua perché la frase  $ab$  è derivabile come  $a_1b_2$  oppure come  $a_3b_4$ . Di conseguenza anche la grammatica estesa

$$S \rightarrow a^*b \mid ab^*$$

risulta ambigua.

## 2.8 Grammatiche e famiglie di linguaggi più generali

La famiglia *LIB* dei linguaggi liberi ben si addice a molti dei costrutti più frequenti, come le strutture a parentesi e le liste a uno o più livelli, ma fallisce con altri costrutti pur semplici che si presentano nei linguaggi artificiali: si ricordano da p. 76 il linguaggio a tre esponenti e le repliche.

Per tale motivo, la teoria dei linguaggi formali ha esplorato continuamente altri tipi di grammatiche, più generali di quelle libere. Tali grammatiche, più

potenti delle libere ma anche tanto più difficili, non hanno quasi mai raggiunto il mondo delle applicazioni. Poiché alla base di tutti gli sviluppi teorici successivi sta la classificazione delle grammatiche dovuta al linguista Noam Chomsky, conviene presentarla brevemente a scopo di riferimento.

### 2.8.1 Classificazione di Chomsky

La tabella seguente raccoglie la storica classificazione delle grammatiche a struttura di frase (phrase structure) basata sulla forma delle regole di trascrizione (rewriting rules). Un fatto alquanto sorprendente è che piccole differenze nella forma delle regole causano grandi cambiamenti nelle proprietà algoritmiche della corrispondente famiglia di linguaggi.

Una regola di trascrizione ha al solito una parte sinistra e una parte destra, entrambe stringhe di alfabeto terminale  $\Sigma$  e nonterminale  $V$ . I quattro tipi sono così caratterizzati:

- una regola del *tipo 0* permette scrivere una stringa arbitraria di simboli terminali e non, al posto di una stringa arbitraria ma non vuota;
- una regola del *tipo 1* aggiunge alla forma delle regole del tipo 0 un vincolo sulla lunghezza: la parte destra deve essere almeno altrettanto lunga della parte sinistra;
- una regola del *tipo 2* coincide con la forma nota delle grammatiche libere: la parte sinistra deve essere un singolo nonterminale;
- una regola del *tipo 3* coincide con la forma nota delle grammatiche unilineari.

Per completezza, la tabella riporta anche il nome del modello di automa (ossia di algoritmo astratto) che riconosce le frasi del linguaggio.

| <i>grammatica</i>   | <i>forma delle regole</i>   | <i>famiglia del linguaggio</i>  | <i>tipo del riconoscitore</i>   |
|---|---|---|---|
| <i>Tipo 0</i>   | $\beta \rightarrow \alpha$ dove $\alpha, \beta \in (\Sigma \cup V)^*$ e $\beta \neq \epsilon$   | Ricorsivamente enumerabile  | Macchina di Turing  |
| <i>Tipo 1</i> o<br>contestuale<br>o dipendente<br>dal contesto<br>(context sensitive)             | $\beta \rightarrow \alpha$ dove $\alpha, \beta \in (\Sigma \cup V)^*$ e $ \beta  \leq  \alpha $   | Contestuale o<br>dipendente dal<br>contesto                                 | Macchina di Turing con nastro<br>di lunghezza limitata linearmente<br>dalla lunghezza della stringa da<br>riconoscere |
| <i>Tipo 2</i> o<br>libera dal<br>contesto<br>o non-<br>contestuale<br>(context-<br>free) o<br>BNF | $A \rightarrow \alpha$ dove $A$ è un non-terminale e $\alpha \in (\Sigma \cup V)^*$   | Linguaggi liberi (dal contesto)<br><i>LIB</i> o non contestuali o algebrici | Automa con memoria a pila   |
| <i>Tipo 3</i> o<br>unilineare<br>(lineare a destra o lineare<br>a sinistra)                       | Lineare a destra: $A \rightarrow uB$ ,<br>Lineare a sinistra: $A \rightarrow Bu$ , dove $A$ è un nonterminale, $u \in \Sigma^*$ e $B \in (V \cup \epsilon)$ | Regolari <i>REG</i> o razionali o a stati finiti                            | Automa finito   |

Le famiglie di linguaggi sono in relazione di contenimento stretto dall'alto verso il basso della tabella.

Le differenze tra le varie classi stanno, oltre che nella forma delle regole, nelle proprietà dei loro automi riconoscitori, che per i tipi 2 e 3 saranno studiati nei prossimi capitoli.

Dal punto di vista della memoria necessaria all'algoritmo che riconosce le frasi, si nota che, mentre per le classi 0, 1 e 2 essa è illimitata, per i linguaggi del tipo 3 basta una memoria finita.

Senza addentrarsi nello studio delle proprietà delle diverse classi, si citano alcuni altri punti.

Tutte e quattro le famiglie di linguaggi sono chiuse rispetto all'unione, al concatenamento, alla stella, all'inversione speculare, all'intersezione con linguaggi regolari. Ma le famiglie differiscono rispetto ad altre proprietà di chiusura: ad es. la chiusura risp. al complemento si ha per i tipi 1 e 3, ma non per i tipi 0 e 2.

Dal punto di vista della decidibilità, un salto algoritmico separa i linguaggi del tipo 0 da quelli del tipo 1, pur così simili nella struttura delle regole. Per i primi non esiste un algoritmo generale per decidere se una stringa appartiene al linguaggio (il problema è indecidibile o più precisamente semi-decidibile).

Invece per i linguaggi contestuali tale algoritmo esiste, anche se la sua complessità è non polinomiale.

Infine soltanto per il tipo 3 è decidibile se due grammatiche definiscono lo stesso linguaggio.

Si completa la presentazione con alcuni esempi di grammatiche e linguaggi del tipo 1 o contestuale.

*Esempio 2.92.* Grammatica contestuale del linguaggio a tre esponenti.

Un linguaggio non libero noto

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

è generato dalla seguente grammatica di tipo 1:

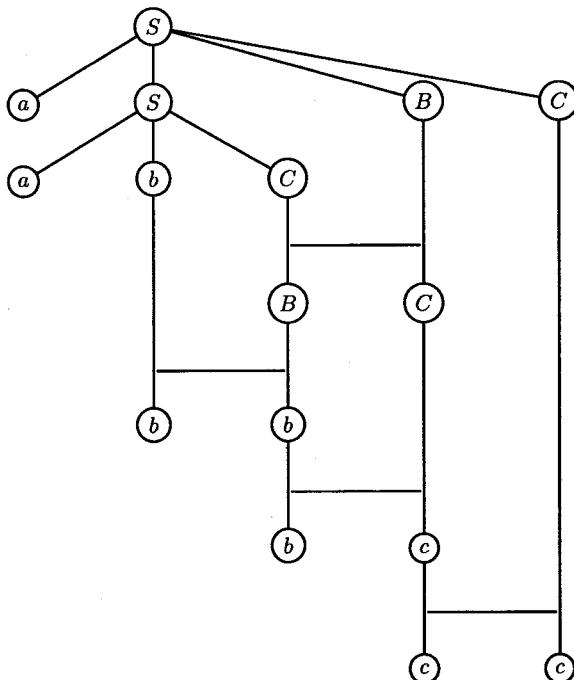
$$\begin{array}{lll} 1. S \rightarrow aSBC & 3. CB \rightarrow BC & 5. bC \rightarrow bc \\ 2. S \rightarrow abC & 4. bB \rightarrow bb & 6. cC \rightarrow cc \end{array}$$

Per le grammatiche dei tipi 1 e 0 la derivazione non è più rappresentabile come un albero, in quanto alla parte sinistra di una regola non corrisponde più un solo simbolo, bensì un insieme di simboli. Tuttavia si può disegnare la derivazione d'una frase sotto forma di un grafo, dove l'applicazione d'una regola come  $BA \rightarrow AB$  appare come sostituzione simultanea di tutti i simboli della parte sinistra con quelli della parte destra.

Per generare la frase  $aabbcc$  sembra ragionevole cercare di costruire una derivazione sinistra:

$$S \Rightarrow aSBC \Rightarrow aabCBC \Rightarrow aabcBC \Rightarrow \text{blocco!}$$

Purtroppo la derivazione si blocca, prima di riuscire a eliminare tutti i simboli nonterminali. Per generare la stringa voluta si deve usare una derivazione non sinistra, disegnata nel grafo seguente.



Intuitivamente la derivazione genera il numero voluto di lettere  $a$  mediante la regola 1 (del tipo 2 autoincassata) e poi la regola 2. I nonterminali  $B$  e  $C$  sono generati con il numero giusto di ripetizioni, ma mescolati tra loro. Per ottenere la stringa voluta, tutte le  $C$  devono essere spostate a suffisso, mediante la regola 3.  $CB \rightarrow BC$ . La prima applicazione della 3 riordina la forma di frase in modo che una  $B$  viene a essere contigua a una  $b$ , abilitando così la derivazione 4.  $bB \rightarrow bb$ . Si procede allo stesso modo alternando i passi 3 e 4, finché nella forma di frase non restano che delle  $C$  come nonterminali. Esse sono trasformate nei terminali  $c$ , mediante la regola  $bC \rightarrow bc$ , seguita da ripetute applicazioni della regola  $cC \rightarrow cc$ .

Si è constatato nel primo tentativo che, a differenza delle grammatiche libere, per quelle contestuali non vale la proprietà che il linguaggio generato mediante derivazioni sinistre coincide con quello generato mediante derivazioni qualsiasi.

Queste grammatiche consentono di trattare le repliche (o le liste con cordanze), una struttura sintattica di interesse pratico già introdotta per esemplificare i limiti della famiglia *LIB*.

*Esempio 2.93.* Grammatica contestuale delle repliche.

Il linguaggio  $L = \{ycy \mid y \in \{a, b\}^+\}$  contiene le frasi del tipo  $aabcaab$  in cui

vi è una marca centrale  $c$  e gli elementi di eguale posizione nel prefisso  $y$  e nel suffisso  $y$  devono essere eguali.

Per semplificare la scrittura delle regole, si suppone che le frasi siano terminate a destra dalla marca di fine stringa,  $\dashv$ .

La grammatica del tipo 1 è :

$$\begin{array}{lllll} S \rightarrow X \dashv & XA \rightarrow XA' & A'A \rightarrow AA' & A' \dashv \rightarrow a & B'a \rightarrow ba \\ X \rightarrow aXA & XB \rightarrow XB' & A'B \rightarrow BA' & B' \dashv \rightarrow b & B'b \rightarrow bb \\ X \rightarrow bXB & & B'A \rightarrow AB' & A'a \rightarrow aa & Xa \rightarrow ca \\ & & B'B \rightarrow BB' & A'b \rightarrow ab & Xb \rightarrow cb \end{array}$$

La grammatica segue questa strategia: dapprima genera un palindromo, quale  $aabXBA$ , dove  $X$  marca il centro della frase e la seconda metà è scritta con simboli maiuscoli. Poi la seconda metà  $B'AA$  viene invertita, trasformandola, in più passi, in  $A'A'B'$ . Infine i simboli maiuscoli apostrofati sono trascritti come  $aab$ , e il centro  $X$  è convertito in  $c$ .

Si mostra la derivazione della frase  $aabcaab$ , sottolineando per facilitare la lettura, la parte sinistra di ogni regola applicata:

$$\begin{aligned} S &\Rightarrow X \dashv \Rightarrow a\cancel{X}A \dashv \Rightarrow a\cancel{a}XAA \dashv \Rightarrow aab\cancel{X}BA \dashv \Rightarrow \\ &aab\cancel{X}B'AA \dashv \Rightarrow aab\cancel{X}AB'A \dashv \Rightarrow aab\cancel{X}A'B'A \dashv \Rightarrow \\ &aab\cancel{X}A'AB' \dashv \Rightarrow aab\cancel{X}AA'B' \dashv \Rightarrow aab\cancel{X}A'A'B' \dashv \Rightarrow \\ &aab\cancel{X}A'A'b \Rightarrow aab\cancel{X}A'ab \Rightarrow aab\cancel{X}aab \Rightarrow aabcaab \end{aligned}$$

Si può osservare che la strategia di generazione delle frasi, applicata in senso opposto, permetterebbe di riconoscere se una stringa data appartiene al linguaggio. L'algoritmo dovrebbe memorizzare su di un nastro le stringhe via via ottenute, a partire da quella data; tale formulazione algoritmica corrisponde al calcolo svolto da una macchina di Turing, che non esce mai con la testina dal segmento di nastro occupato dalla stringa data.

Gli esempi dovrebbero convincere il lettore della difficoltà di applicare questo genere di grammatiche alla definizione di linguaggi di maggiori dimensioni. Infatti la grammatica si comporta come un algoritmo in cui le regole interagiscono in modo intricato e poco comprensibile.

Detto diversamente, le grammatiche del tipo 1 e 0 possono essere viste come una particolare notazione per programmare algoritmi qualsiasi, anche di natura matematica, facendo ricorso al solo meccanismo della trascrizione delle stringhe. Un caso tipico<sup>25</sup> è l'algoritmo, in forma di grammatica contestuale, che genera i numeri primi nella notazione unaria, ovvero il linguaggio  $\{a^n \mid n \text{ numero primo}\}$ . Tali sviluppi della teoria dei linguaggi formali nella direzione degli insiemi di natura matematica non hanno però utilità per il

<sup>25</sup>Presentato in [43].

progetto dei linguaggi dell'informatica.

Ciononostante le grammatiche contestuali hanno avuto alcuni rari fautori tra i progettisti dei linguaggi artificiali e dei compilatori. Si cita il linguaggio Algol 68, interamente definito mediante una particolare classe di grammatiche contestuali, dette a due livelli.<sup>26</sup>

In conclusione si deve ammettere che, allo stato dell'arte della teoria delle grammatiche formali, resta insoddisfatta l'esigenza di definire certi costrutti sintattici utili e frequenti come le repliche, che sfuggono alla capacità generativa delle grammatiche libere. Il progettista dei compilatori dovrà quindi definire tali costrutti con altri metodi detti semanticici, che aggiungono dei vincoli di diversa natura alle regole grammaticali. Essi sono l'argomento del capitolo 5.

---

<sup>26</sup>Note anche come VW-grammatiche dal nome di Van Wijngarten[51]; vedasi anche Cleaveland e Uzgalis [12].

## Automi finiti e riconoscimento dei linguaggi regolari

### 3.1 Introduzione

L'impiego delle grammatiche e delle espressioni regolari è molto diffuso nella scrittura dei manuali e dei documenti di standardizzazione dei linguaggi. Ma per progettare un compilatore, cioè un algoritmo che verifica la correttezza d'una frase e la traduce, è necessario un approccio più algoritmico, esposto in questo capitolo per i linguaggi regolari e nel prossimo per quelli liberi.

Il problema di riconoscere se un testo appartiene a un certo linguaggio formale, prima di tradurlo o elaborarlo con modalità dipendenti dall'applicazione, sorge molto frequentemente. Un compilatore analizza il programma sorgente per controllare che sia valido; un programma di videoscrittura verifica che le parole digitate siano ortografiche e soddisfino la sintassi, e un'interfaccia interattiva a finestre, prima di accettare un dato inserito dall'utente ne controlla la validità.

Tale controllo, essenziale nel trattamento dei testi, è svolto da un *algoritmo di riconoscimento*, così detto perché riconosce se una stringa è corretta rispetto a una definizione formale di riferimento. Le procedure di riconoscimento si specificano e progettano impiegando dei modelli minimalisti, detti *macchine astratte* o *automi*. Così facendo si mettono in risalto le peculiarità delle varie classi di riconoscitori, e dunque delle famiglie di linguaggi riconosciuti, senza appesantire l'esposizione con inutili particolari realizzativi dipendenti dalle scelte tecniche di progetto dell'algoritmo.

Nel capitolo, dopo un cenno agli automi matematici più generali, si presentano gli automi con memoria finita, che riconoscono i linguaggi definiti da espressioni regolari. Molte esigenze di ricerca delle parole in un testo o di estrazione degli elementi lessicali sono risolte con tali modelli.

Il primo modello di automa è quello deterministico, di cui si espongono i metodi basilari di ripulitura e di minimizzazione.

Poi si introduce in modo motivato il modello non deterministico, e se ne mostra la corrispondenza con le grammatiche del tipo unilineare.

Si presentano poi gli algoritmi per effettuare la conversione tra automa rico-

noscitore e espressione regolare del linguaggio riconosciuto, giungendo così al punto di arrivo concettuale, l'identità dei due modelli. Ciò facendo si introduce la sottofamiglia dei linguaggi locali, riconoscibili con dei test locali sulla stringa.

Segue la considerazione degli operatori di complemento e intersezione delle espressioni regolari e del metodo di composizione di automi mediante prodotto cartesiano.

Il capitolo termina con il riepilogo delle relazioni esistenti tra automi finiti, espressioni regolari e grammatiche.

Gli automi finiti sono fondamentali per la compilazione e riappariranno più volte nei prossimi capitoli del libro, in particolare come traduttori, ossia macchine per tradurre un testo nel capitolo 5. Nello stesso capitolo saranno viste altre applicazioni degli automi finiti nell'analisi statica del flusso d'un programma da verificare o da ottimizzare.

## 3.2 Algoritmi di riconoscimento e automi

Gli *algoritmi di riconoscimento* (o di decisione) sono spesso considerati dagli studiosi della teoria della complessità del calcolo, e non solo per i problemi dei linguaggi artificiali. Ad es. il celebre problema di decidere se due grafi sono isomorfi si riformula in questi termini come problema di riconoscimento: il dominio del problema è costituito da coppie di grafi e l'algoritmo deve emettere la risposta *sì/no*, a seconda che i grafi della coppia siano o meno isomorfi. Questo è sempre il codominio d'un algoritmo di riconoscimento, anche se, a seconda del problema, il dominio cambia.

Passando ai linguaggi, il dominio è un insieme di stringhe d'un alfabeto  $\Sigma$ . L'applicazione dell'algoritmo di riconoscimento  $\alpha$  a una stringa data  $x$  si può indicare come  $\alpha(x)$ . Si dice che la stringa  $x$  è (risp. non è) *riconosciuta* o *accettata* se  $\alpha(x) = \text{sì}$  (risp.  $\alpha(x) = \text{no}$ ).

Il linguaggio riconosciuto,  $L(\alpha)$ , è allora l'insieme delle stringhe per cui:

$$L(\alpha) = \{x \in \Sigma^* \mid \alpha(x) = \text{sì}\}$$

Può succedere che per qualche stringa scorretta  $x \in (\Sigma^* \setminus L(\alpha))$  l'algoritmo non termini, ossia  $\alpha(x)$  non sia definita: in questo caso si dice che il problema di decidere l'appartenenza di  $x$  a  $L$  è semidecidibile, o anche che il linguaggio  $L$  è ricorsivamente enumerabile. Se un linguaggio fosse semidecidibile, non si potrebbe garantire che il compilatore non entri in ciclo quando analizza certe stringhe sorgente.

In pratica queste preoccupazioni, pur centrali per la teoria della computabilità,<sup>1</sup> non sono rilevanti per l'ingegneria del linguaggio, dove si evitano deliberatamente le famiglie di linguaggi semidecidibili. Tuttavia due problemi

---

<sup>1</sup> Il lettore interessato a un approfondimento può consultare un testo di informatica teorica (ad es. Bovet e Crescenzi [11], Floyd e Beugel [19], Hopcroft e Ullman [28], Mandrioli e Ghezzi [32], Mc Naughton [33]).

indecidibili, diversi dal riconoscimento, sono stati incontrati: quello di verificare se due grammatiche sono equivalenti e di decidere se una grammatica è ambigua.

Nella pratica il riconoscimento è soltanto la prima parte del lavoro della compilazione. Nel capitolo 5, scostandosi dai puri problemi di riconoscimento, si studierà la traduzione d'un linguaggio in un altro. Il codominio d'una traduzione è più ricco di  $\{sì, no\}$ , essendo esso stesso un linguaggio.

È noto che gli algoritmi di risoluzione dei problemi decidibili possono essere classificati in base alla loro complessità, misurata dalle risorse di calcolo, cioè della quantità di tempo (oppure di memoria), che richiedono. Quasi tutti i problemi di interesse per la compilazione hanno delle soluzioni di complessità bassa, lineare o al peggio polinomiale rispetto alle dimensioni del problema da risolvere.

Quando si studia la complessità di calcolo, invece di misurare il tempo di esecuzione si preferisce contare il numero di passi compiuti dall'algoritmo. La ragione è che così facendo si hanno delle misure che non dipendono dalla velocità dell'elaboratore, ma soltanto dalla natura dell'algoritmo. I passi elementari dell'algoritmo sono operazioni più o meno ricche, a seconda del modello di calcolatore (in senso astratto) cui si pensa: a un estremo, un passo è un'istruzione d'un linguaggio programmatico; all'altro estremo un passo è l'operazione d'un automa o macchina astratta, capace solo di leggere e scrivere un simbolo. L'uso d'una macchina astratta è preferito nella teoria e nell'ingegneria del linguaggio, tanto da caratterizzarle rispetto a altri campi dell'informatica. In questo campo è consuetudine presentare gli algoritmi di riconoscimento e traduzione sotto la veste di automi. I vantaggi sono dupli: evidenziare la correlazione tra le varie famiglie di automi riconoscitori e le corrispondenti famiglie di linguaggi e di grammatiche; e evitare un riferimento prematuro agli aspetti realizzativi del software. Ciò facendo non si perdono di vista gli sviluppi applicativi, perché è facile dare le indicazioni essenziali per trasformare un automa in un programma.

### 3.2.1 Automa generale

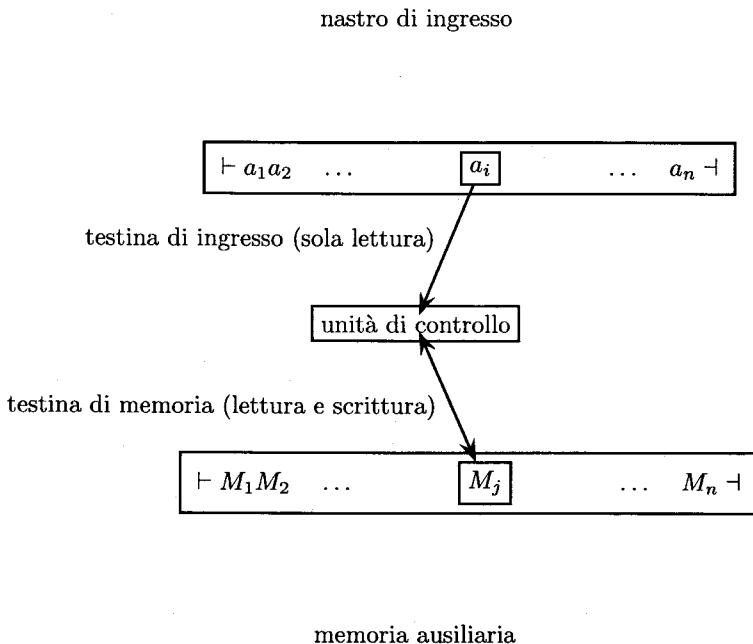
Un automa o macchina astratta è un dispositivo di calcolo ideale, dotato d'un insieme ridotto di operazioni elementari assai semplici. La varietà di questi dispositivi si è continuamente arricchita nel corso della storia della teoria dei linguaggi e degli automi, a partire dai classici studi di A. Turing degli anni 1930 sull'omonima macchina di calcolo. Nel libro si studieranno soltanto i pochi tipi di automi impiegati per l'elaborazione dei linguaggi tecnici.<sup>2</sup>

Lo schema d'un automa riconoscitore, nella sua forma più generale, comprende

---

<sup>2</sup>Per approfondire la teoria generale degli automi si rinvia il lettore a Salomaa [43], Hopcroft e Ullman [27, 28], Harrison [24]; per gli automi finiti un riferimento specifico è Sakarovitch [42].

tre parti: il nastro d'ingresso, l'unità di controllo, e la memoria ausiliaria:



L'unità di controllo ha una memoria limitata (si dice che ha un numero finito di stati), mentre la memoria ausiliaria ha in generale capacità illimitata. Sul nastro superiore sta scritta la stringa di ingresso (o *sorgente*), che può essere letta ma non modificata; esso è suddiviso in caselle, ciascuna contenente un simbolo dell'alfabeto terminale. Talvolta si impone che la stringa sia delimitata a sinistra e a destra da due caratteri riservati  $\vdash$  (marca d'inizio) e  $\dashv$  (marca di fine o terminatore). Anche la memoria ausiliaria può essere vista come un nastro, contenente simboli d'un alfabeto di memoria.

L'automa è un sistema che opera in istanti discreti svolgendo diverse azioni: leggere il *carattere*  $a_i$  *corrente*, spostare la testina di ingresso sul nastro a destra o sinistra d'una posizione, leggere il simbolo corrente  $M_j$  dalla memoria, e scrivere un simbolo al suo posto, spostare la testina di memoria, e cambiare lo stato della unità di controllo.

L'automa esamina la stringa sorgente compiendo una serie di mosse; ogni mossa dipende dai simboli presenti sotto le due testine e dallo stato del controllo. Una mossa può avere i seguenti effetti:

- spostamento della testina di ingresso d'una posizione a destra o a sinistra;

- scrittura d'un simbolo al posto del simbolo corrente nella memoria e spostamento della testina di memoria (d'una posizione a destra o sinistra);
- cambiamento di stato dell'unità di controllo.

Alcune di queste azioni possono mancare.

L'automa è *monodirezionale* se la testina d'ingresso si può spostare soltanto da sinistra verso destra: questo è il solo caso considerato, perché meglio risponde all'esigenza di tradurre un testo con una sola scansione.

Il comportamento futuro dell'automa è definito da tre componenti, costituenti la *configurazione* (istantanea): il suffisso non ancora letto della stringa sorgente, situato a destra della testina d'ingresso; il contenuto della memoria ausiliaria e la posizione della testina di memoria; lo stato della unità di controllo. L'automa è all'origine posto nella *configurazione iniziale*: la testina d'ingresso sta sul simbolo  $a_1$  che segue la marca d'inizio, l'unità di controllo è nello stato iniziale, e la memoria contiene l'informazione iniziale (solitamente un particolare simbolo).

Attraverso una serie di mosse (calcolo o computazione) la configurazione dell'automa evolve. Il cambiamento è *deterministico* se in ogni configurazione istantanea al più una mossa è possibile, *indeterministico* (o non deterministico) in caso contrario.

Un automa indeterministico è un modo astratto di rappresentare un algoritmo che in certe situazioni procede per tentativi, esaminando due o più strade alternative.

Una *configurazione* è *finale* se il controllo è in uno stato qualificato come finale e la testina d'ingresso è posta sul terminatore della stringa. In altre varianti, in alternativa al trovarsi in uno stato finale, la configurazione finale è caratterizzata da una particolare condizione imposta al contenuto della memoria: per esempio l'essere vuota o il contenere un simbolo specificato.

La *stringa sorgente*  $x$  è *accettata* se l'automa, partendo dalla configurazione iniziale con  $\vdash x \dashv$  in ingresso, esegue un calcolo che lo porta in una configurazione finale (un automa indeterministico potrebbe raggiungere la configurazione finale in più modi diversi).

L'insieme delle stringhe accettate dall'automa costituisce il *linguaggio accettato* (riconosciuto, definito) da esso.

Il calcolo termina, o perché l'automa ha raggiunto una configurazione finale, o perché non può svolgere alcuna mossa, nel senso che la mossa non è definita in quella configurazione istantanea; nel qual caso la stringa d'ingresso non è accettata con tale calcolo.

Due automi che accettano lo stesso linguaggio sono detti *equivalenti*. Naturalmente non è detto che due automi equivalenti siano dello stesso tipo, né che abbiano la stessa complessità di calcolo.

### *Macchina di Turing*

Lo schema precedente di automa riproduce sostanzialmente la macchina introdotta da A. Turing nel 1936 e oggi accettata come formalizzazione di ogni

procedura di calcolo sequenziale. La famiglia dei linguaggi accettati è detta *ricorsivamente enumerabile*.

Il linguaggio riconosciuto è detto *decidibile* se esiste una macchina di Turing che lo accetta e che si ferma per ogni stringa d'ingresso. La famiglia dei linguaggi decidibili è più ristretta di quella dei linguaggi ricorsivamente enumerabili.

Tale macchina ha interesse come riconoscitore di due delle famiglie di linguaggi della classificazione gerarchica di Chomsky (p. 87). Infatti i linguaggi generati dalle grammatiche del tipo 0 sono esattamente i linguaggi ricorsivamente enumerabili.

I linguaggi dipendenti dal contesto o del tipo 1 sono quelli riconosciuti da una macchina di Turing che usa un nastro di memoria di lunghezza proporzionale alla lunghezza della stringa da riconoscere.

La macchina di Turing non ha utilità pratica, ma è un riferimento mentale e un termine di confronto per gli automi più efficienti che si usano come riconoscitori dei linguaggi tecnici.

Riconsiderando la memoria ausiliaria, sono due i casi di interesse pratico per la compilazione. Se la memoria ausiliaria manca, l'automa generale diviene la ben nota macchina a stati finiti o automa finito, un modello fondamentale in ogni campo dell'informatica.

Se la memoria è organizzata a pila<sup>3</sup> si ha l'automa a pila, importante per la compilazione in quanto è il riconoscitore dei linguaggi liberi.

Le famiglie di linguaggi che interessano la compilazione e i corrispondenti riconoscitori sono dunque dei casi molto speciali rispetto ai linguaggi ricorsivamente enumerabili. Dal punto di vista della teoria della complessità del calcolo, il problema del riconoscimento, che può essere intrattabile per i linguaggi ricorsivamente enumerabili, ha sempre complessità polinomiale per quelli oggetto di studio. In particolare, i linguaggi regolari, riconosciuti dagli automi finiti, sono una sottoclasse dei linguaggi riconoscibili in tempo reale da un macchina di Turing; mentre i linguaggi liberi sono una sottoclasse dei linguaggi di complessità temporale polinomiale su tale macchina.

### 3.3 Introduzione agli automi finiti

Gli automi finiti sono certamente un dispositivo fondamentale per il calcolo. Da un lato la loro teoria ha raggiunto una grande profondità matematica, dall'altro, le applicazioni sono numerosissime: il progetto digitale, i sistemi di controllo, i protocolli di comunicazione, lo studio dell'affidabilità dei sistemi, ecc. La presente esposizione si focalizza sui pochi aspetti che interessano direttamente il progettista dei linguaggi e dei compilatori, limitandosi a qualche richiamo delle proprietà teoriche essenziali.

Nell'ordine si espongono i diagrammi stato-transizione e gli automi finiti visti

---

<sup>3</sup>Push-down stack, LIFO o Last In First Out.

come riconoscitori di linguaggi definiti da grammatiche unilineari. Si esamina poi il comportamento deterministico e non, e il passaggio dal secondo al primo. Segue la costruzione del riconoscitore finito d'un linguaggio descritto da un'espressione regolare, e viceversa. Si approfondiscono poi le proprietà dei linguaggi regolari presentando la chiusura rispetto al complemento e alla intersezione.

Conformemente allo schema generale, un automa finito possiede: il nastro d'ingresso contenente la stringa sorgente  $x \in \Sigma^*$ ; l'unità di controllo con la sua memoria limitata; la testina di lettura, inizialmente posta sul primo carattere di  $x$ , moventesi da sinistra a destra con una sola scansione, sino al termine di  $x$  (o al primo errore). Dopo la lettura d'un carattere l'automa aggiorna lo stato dell'unità di controllo. Alla fine della scansione, l'automa riconosce o meno  $x$ , a seconda dello stato in cui si trova.

Un *diagramma stato-transizione* è un grafo orientato rappresentante l'automa. I nodi del grafo sono gli stati dell'unità di controllo. Gli archi, etichettati con caratteri terminali, mostrano i cambiamenti di stato (*transizioni*) che avvengono alla lettura dei caratteri della stringa.

*Esempio 3.1.* Costanti numeriche decimali.

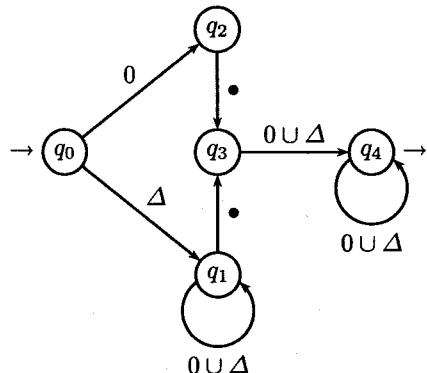
Il linguaggio  $L$  delle costanti numeriche decimali è così definito:

$\Sigma = \Delta \cup \{0, \bullet\}$ , dove  $\Delta = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$  è una cifra diversa dallo zero;

$$L = (0 \cup \Delta(0 \cup \Delta)^*) \bullet (0 \cup \Delta)^+$$

Il suo riconoscitore è descritto dal *diagramma* o anche dalla *tabella stato-transizioni*:

*diagramma stato-transizione*



*tabella stato-transizione*

| Stato<br>Presente | Carattere corrente |       |     |       |       |
|-------------------|--------------------|-------|-----|-------|-------|
|                   | 0                  | 1     | ... | 9     | •     |
| → $q_0$           | $q_2$              | $q_1$ | ... | $q_1$ | —     |
| $q_1$             | $q_1$              | $q_1$ | ... | $q_1$ | $q_3$ |
| $q_2$             | —                  | —     | ... | —     | $q_3$ |
| $q_3$             | $q_4$              | $q_4$ | ... | $q_4$ | —     |
| $q_4 \rightarrow$ | $q_4$              | $q_4$ | ... | $q_4$ | —     |

Per semplicità di disegno sono riuniti gli archi topologicamente eguali: ad es. l'arco  $(q_0 \rightarrow q_1)$  rappresenta un fascio di 9 archi, portanti le etichette 1, 2, ..., 9. Lo stato iniziale  $q_0$  dell'automa è individuato dalla freccia entrante; la freccia uscente distingue lo stato finale ( $q_4$ ), nel quale l'automa accetta la

stringa.

La matrice delle incidenze del grafo forma la tabella delle transizioni. Essa riporta per ogni coppia (stato presente, carattere corrente) il prossimo stato. Gli stati iniziali e finali sono distinti con le frecce.

La stringa sorgente  $0 \bullet 2\bullet$  fa transitare l'automa per gli stati  $q_0, q_2, q_3, q_4$ , nell'ultimo dei quali non è però lecita la lettura del carattere  $\bullet$ . Poiché il calcolo si arresta prima di aver esaurito la scansione della stringa, essa non è riconosciuta. Invece la stringa  $0 \bullet 21$  è accettata.

Se invece si volesse ridefinire il linguaggio per accettare anche le stringhe terminanti con  $\bullet$ , come  $305\bullet$ , basterebbe contrassegnare come finale anche lo stato  $q_3$ .

Un automa può dunque avere più stati finali, ma un solo stato iniziale (altrimenti non sarebbe deterministico).

### 3.4 Automa finito deterministico

Si precisano ora i concetti introdotti nell'esempio precedente.

Un *automa M finito deterministico* è costituito da cinque entità:

1.  $Q$ , l'*insieme degli stati* (finito e non vuoto)
2.  $\Sigma$ , l'*alfabeto di ingresso* (o terminale)
3. la *funzione di transizione*  $\delta : (Q \times \Sigma) \rightarrow Q$
4.  $q_0 \in Q$ , lo *stato iniziale*
5.  $F \subseteq Q$ , l'*insieme degli stati finali*.

La funzione  $\delta$  specifica le mosse dell'automa: il significato di  $\delta(q, a) = r$  è che  $M$ , trovandosi nello stato (presente)  $q$  e leggendo  $a$ , si porta nello stato (successivo)  $r$ . Se  $\delta(q, a)$  non è definita, l'automa si ferma, e si può pensare che entri nello stato d'errore o rifiuto.

Per elaborare una stringa  $x$  non vuota, l'automa esegue una serie di mosse.

Sia ad es.  $x = ab$ ; leggendo il primo carattere la mossa  $\delta(q_0, a) = q_1$  porta nello stato  $q_1$ , in cui si esegue la seconda mossa  $\delta(q_1, b) = q_2$ .

Per brevità, invece di scrivere  $\delta(\delta(q_0, a), b) = q_2$ , si combineranno le due mosse, scrivendo  $\delta(q_0, ab) = q_2$ , a indicare che la lettura della stringa  $ab$  porta nello stato  $q_2$ : il secondo argomento della funzione delta può dunque essere una stringa.

Nel caso della stringa vuota si impone che la macchina resti sempre nello stato in cui si trova:

$$\forall q \in Q : \delta(q, \varepsilon) = q$$

A seguito di queste estensioni, il dominio diviene  $(Q \times \Sigma^*)$  e la funzione è definita da:

$$\forall \text{ carattere } a \in \Sigma, \forall \text{ stringa } y \in \Sigma^* :$$

$$\delta^*(q \bullet ja) = \delta(\delta^*(q, y), a)$$

Nella rappresentazione grafica dell'automa, se risulta  $\delta(q, y) = q'$ , allora, e solo allora, esiste un cammino dallo stato  $q$  allo stato  $q'$ , etichettato dalla stringa  $y$ , percorrendo il quale l'automa scandisce la stringa  $y$ . Quando l'automa percorre tale cammino, si dice che esegue un *calcolo*.

Una stringa  $x$  è *riconosciuta* (o accettata) dall'automa se scandendola l'automa cammina dallo stato iniziale a uno stato finale,  $\delta(q_0, x) \in F$ .

Come caso particolare, la stringa vuota è riconosciuta se, e solo se, lo stato iniziale è anche finale.

Il *linguaggio riconosciuto* dall'automa  $M$  è

$$L(M) = \{x \in \Sigma^* \mid x \text{ è riconosciuta da } M\}$$

I linguaggi riconosciuti da automi del tipo considerato sono detti *riconoscibili a stati finiti*.

- » Due automi sono *equivalenti* se riconoscono lo stesso linguaggio.

*Esempio 3.2.* (Esempio 3.1 continuato).

L'automa  $M$  di p. 99 è definito da:

$$\begin{aligned} Q &= \{q_0, q_1, q_2, q_3, q_4\} \\ \Sigma &= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \bullet\} \\ q_0 &= q_0 \\ F &= \{q_4\} \end{aligned}$$

Esempi di transizioni:

$$\delta(q_0, 3 \bullet 1) = \delta(\delta(q_0, 3 \bullet), 1) = \delta(\delta(\delta(q_0, 3), \bullet), 1) = \delta(\delta(q_1, \bullet), 1) = \delta(q_3, 1) = q_4$$

Poiché  $q_4 \in F$ , la stringa  $3 \bullet 1$  è accettata. Invece, poiché  $\delta(q_0, 3 \bullet) = q_3$  non è finale, la stringa  $3 \bullet$  non appartiene al linguaggio, né vi appartiene  $02$  in quanto  $\delta(q_0, 02) = \delta(\delta(q_0, 0), 2) = \delta(q_2, 2)$ , che non è definita.

L'efficienza di calcolo d'un automa deterministico è la migliore possibile in assoluto: infatti l'algoritmo decide l'appartenenza della stringa in tempo reale, scandendola una sola volta da sinistra a destra.

### 3.4.1 Stato di errore e completamento dell'automa

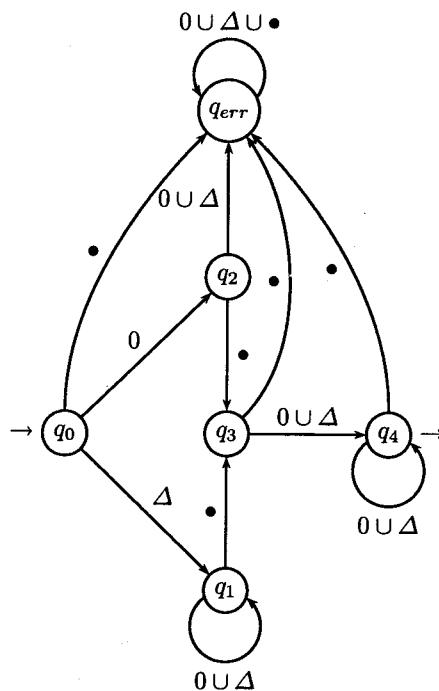
Se nello stato  $q$  la mossa non è definita per il carattere  $a$ , si può dire che l'automa alla lettura di tale carattere cade nello stato d'errore,  $q_{err}$ , dal quale non potrà più uscire, qualsiasi carattere sia successivamente letto. Per questa ragione tale stato è anche chiamato il *pozzo*.

- » La funzione di transizione può essere sempre *completata* con lo stato di errore, senza modificare il linguaggio accettato, ponendo:

»  $\forall$  stato  $q \in Q$  e  $\forall$  carattere  $a \in \Sigma$  se  $\delta(q, a)$  è indefinita, ponì  $\delta(q, a) = q_{err}$

$$\forall \text{ carattere } a \in \Sigma \text{ ponì } \delta(q_{err}, a) = q_{err}$$

Il riconoscitore delle costanti decimali, nella versione completata con lo stato di errore, è:



Se un calcolo cade nello stato di errore non ne esce, né può raggiungere lo stato finale per riconoscere una stringa. Di conseguenza, l'automa completato accetta lo stesso linguaggio dell'automa originale. Convenzionalmente lo stato di errore è sempre sottinteso e non occorre disegnarlo.

### 3.4.2 Automa pulito

Un automa potrebbe contenere delle parti inutili, che non danno contributo al linguaggio riconosciuto e vanno normalmente eliminate. I concetti seguenti valgono anche per gli automi indeterministici e in generale per ogni tipo di automa.

Uno stato  $q$  è *raggiungibile* da uno stato  $p$  se esiste un calcolo che porta l'automa da  $p$  a  $q$ . Uno stato è *accessibile* se è raggiungibile dallo stato iniziale, è *post-accessibile* se da esso uno stato finale è raggiungibile. Uno stato accessibile e post-accessibile è detto *utile*. In altro modo, uno stato utile giace su un cammino tra lo stato iniziale e uno stato finale.

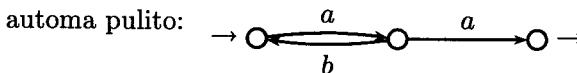
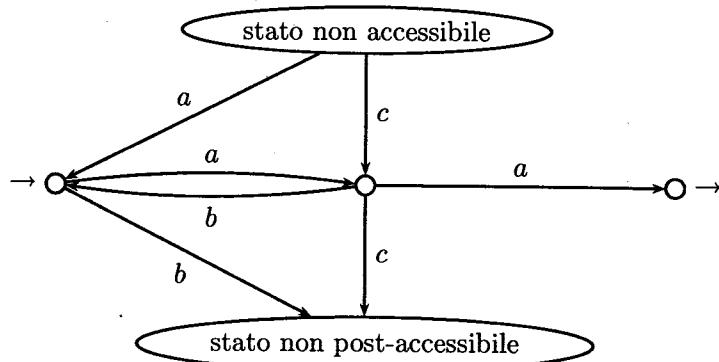
Un automa è *pulito* se ogni suo stato è utile.

Proprietà 3.3. Ogni automa finito ammette un automa pulito equivalente.

Per pulire l'automa, individuati gli stati non utili, li si sopprime con tutti gli archi incidenti, nel modo ora illustrato.

*Esempio 3.4.* Eliminazione degli stati inutili.

automa non pulito:



### 3.4.3 Automa minimo

Ogni linguaggio può essere riconosciuto da tanti automi equivalenti pur se diversi rispetto al numero di stati o alla funzione di transizione. Ma l'automa più piccolo è unico, nel senso precisato dalla seguente affermazione.

*Proprietà 3.5.* Per ogni linguaggio a stati finiti, il riconoscitore deterministico minimo rispetto al numero degli stati è unico (a meno d'una ridenominazione degli stati).

Prima di descrivere un algoritmo di minimizzazione<sup>4</sup> di un automa, è necessaria una nuova definizione.

Si suppone che l'automa sia pulito. Tuttavia esso può contenere due stati che, ai fini del riconoscimento, possono essere fusi in un solo stato. Come individuarli? Si definisce una relazione binaria, detta di *indistinguibilità* o di Nerode, tra coppie di stati.

**Definizione 3.6.** Gli stati  $p$  e  $q$  sono indistinguibili se, e solo se, per ogni stringa  $x \in \Sigma^*$ , o entrambi gli stati  $\delta(p, x)$  e  $\delta(q, x)$  sono finali, o nessuno dei due lo è.

La relazione complementare è chiamata distinguibilità.

Parafrasando la definizione, due stati  $p, q$  sono indistinguibili se, partendo rispettivamente dall'uno e dall'altro e scandendo la stessa stringa  $x$  non è possibile giungere a due stati, uno finale, l'altro non.

Si osserva che:

---

<sup>4</sup>Il problema della minimizzazione è stato risolto anche con altri algoritmi. Si veda la rassegna in [52].

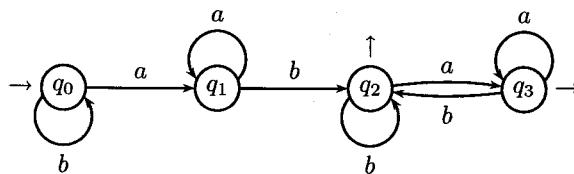
1. lo stato pozzo  $q_{err}$  è distinguibile da ogni altro stato  $p$ , poiché certamente esiste una stringa  $x$  per cui risulta  $\delta(p, x) \in F$ , mentre per ogni stringa  $x$ , è  $\delta(q_{err}, x) = q_{err}$ ;
2.  $p$  e  $q$  sono distinguibili se  $p$  è finale e  $q$  no, poiché  $\delta(p, \varepsilon) \in F$  e  $\delta(q, \varepsilon) \notin F$ ;
3.  $p$  e  $q$  sono distinguibili se, per qualche carattere  $a$ , gli stati prossimi  $\delta(p, a)$  e  $\delta(q, a)$  sono distinguibili.

In particolare  $p$  è distinguibile da  $q$  se gli insiemi delle etichette delle frecce uscenti rispettivamente da  $p$  e da  $q$  sono diversi. Infatti in tale caso, esiste un carattere  $a$ , tale che la mossa dallo stato  $p$  conduce nello stato  $p'$ , mentre la mossa da  $q$  non è definita (ossia conduce nello stato di errore); per la condizione 3. tali stati sono distinguibili.

L'indistinguibilità è una relazione transitiva e riflessiva, le cui classi di equivalenza possono essere calcolate con un semplice procedimento che si descrive con un esempio.

*Esempio 3.7.* Classi di equivalenza degli stati indistinguibili.

Si consideri l'automa  $M$ :



Si costruisce ora la tabella di dimensioni  $|Q| \times |Q|$  della relazione di indistinguibilità. È inutile riempire le caselle poste sopra la diagonale principale, essendo la relazione simmetrica.

Si marcherà la casella  $(p, q)$  con una  $X$  se la coppia di stati è distinguibile. Si inizializzano con  $X$  le caselle per le quali uno solo degli stati è finale:

|       |       |       |       |   |
|-------|-------|-------|-------|---|
| $q_1$ |       | -     | -     | - |
| $q_2$ | $X$   | $X$   | -     | - |
| $q_3$ | $X$   | $X$   |       | - |
| $q_0$ | $q_1$ | $q_2$ | $q_3$ |   |

Quindi si esamina ogni casella  $(p, q)$  non marcata e, per ogni carattere terminale  $a$ , si rilevano gli stati successivi  $r = \delta(p, a)$  e  $s = \delta(q, a)$ .

Se la casella  $(r, s)$  è già marcata  $X$ , cioè  $r$  è distinguibile da  $s$ , allora lo sono anche  $p$  e  $q$ , e la casella  $(p, q)$  è da marcare con  $X$ .

Se invece la casella  $(r, s)$  non è marcata con  $X$ , si inserisce nella casella  $(p, q)$  la coppia  $(r, s)$ , come promemoria che in futuro, se si marcherà  $(r, s)$ , si dovrà marcare anche  $(p, q)$ .

Le coppie di stati prossimi sono indicate nella tabella di sinistra. Il risultato

**del passo** è la tabella di destra, che ha la marca nella casella  $(1,0)$  poiché la coppia  $(0,2) \equiv (2,0)$  era già marcata:

|       |               |       |               |       |
|-------|---------------|-------|---------------|-------|
| $q_1$ | $(1,1),(0,2)$ | -     | -             | -     |
| $q_2$ | $X$           | $X$   | -             | -     |
| $q_3$ | $X$           | $X$   | $(3,3),(2,2)$ | -     |
|       | $q_0$         | $q_1$ | $q_2$         | $q_3$ |

|       |       |       |               |       |
|-------|-------|-------|---------------|-------|
| $q_1$ | $X$   | -     | -             | -     |
| $q_2$ | $X$   | $X$   | -             | -     |
| $q_3$ | $X$   | $X$   | $(3,3),(2,2)$ | -     |
|       | $q_0$ | $q_1$ | $q_2$         | $q_3$ |

Tutte le caselle sono marcate e l'algoritmo termina.

Le caselle non marcate con  $X$  designano le coppie indistinguibili, nell'es. la coppia  $(q_2, q_3)$ .

Una classe d'equivalenza della relazione di indistinguibilità contiene tutti gli stati tra loro indistinguibili.

Nell'esempio le classi d'equivalenza sono  $[q_0], [q_1], [q_2, q_3]$ .

È interessante esaminare anche un caso in cui la funzione di transizione non è totale. Si modifichi l'automa  $M$  cancellando l'autoanello  $\delta(q_3, a) = q_3$ , ossia ridefinendo la funzione come  $\delta(q_3, a) = q_{err}$ . Di conseguenza gli stati  $q_2, q_3$  diventano distinguibili, poiché  $\delta(q_2, a) = q_3, \delta(q_3, a) = q_{err}$ . Le classi di equivalenza sono tutte unitarie.

### Costruzione dell'automa minimo

L'automa minimo  $M'$  equivalente a quello dato  $M$  ha come stati le classi di equivalenza della relazione di indistinguibilità. Per definire la funzione di transizione basta dire che vi è in  $M'$  un arco tra due classi di equivalenza

$$[\dots, p_r, \dots] \xrightarrow{b} [\dots, q_s, \dots]$$

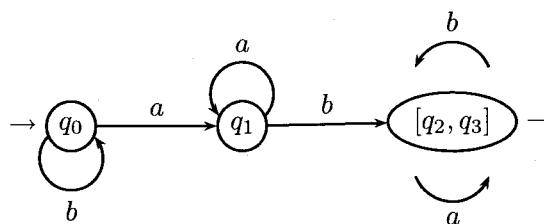
se, e solo se, in  $M$  vi è un arco

$$p_r \xrightarrow{b} q_s$$

tra due stati, ordinatamente appartenenti alle due classi di equivalenza; si noti che lo stesso arco di  $M'$  deriva in generale da più archi di  $M$ .

*Esempio 3.8.* (Esempio 3.7 continuato).

L'automa minimo  $M'$  è:



Questo è l'automa più piccolo equivalente a quello dato. Infatti sarebbe facile constatare che, fondendo insieme due stati non equivalenti rispetto all'indistinguibilità, si ottiene una macchina che riconosce un linguaggio più ampio dell'originale.

L'esistenza dell'algoritmo di minimizzazione dimostra la proprietà 3.5 d'unicità dell'automa minimo equivalente a un automa dato, sulla quale si basa un test per verificare se due automi sono equivalenti: si minimizzano quindi si confrontano, per vedere se sono identici, a meno d'un cambiamento di nome degli stati.<sup>5</sup>

Nelle applicazioni, è ovvio che ragioni di economia fanno preferire l'automa minimo, ma il risparmio può essere trascurabile, soprattutto per i casi che interessano la compilazione.

Addirittura in certe situazioni la minimizzazione degli stati del riconoscitore è da evitare: se l'automa è arricchito con certe azioni per il calcolo d'una funzione di uscita (funzione semantica), come si vedrà nel capitolo 5, due stati, che sarebbero indistinguibili per il riconoscitore, potrebbero essere associati a azioni diverse, con la conseguenza che, fondendoli insieme, si confonderebbero le azioni da compiere.

Infine si anticipa che la proprietà d'unicità dell'automa minimo non vale in generale per gli automi non deterministici di prossima introduzione.

### 3.4.4 Dall'automa alla grammatica

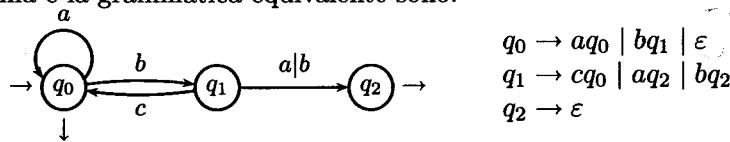
Sarà facile constatare che gli automi a stati finiti e le grammatiche unilineari (o del tipo 3 di Chomsky) di p. 70 definiscono esattamente la stessa famiglia di linguaggi.

Per primo si mostra come costruire una grammatica lineare a destra equivalente a un automa. La grammatica  $G$  ha come simboli nonterminali gli stati  $Q$  dell'automa, come assioma lo stato iniziale, per ogni mossa  $q \xrightarrow{a} r$  ha la regola  $q \rightarrow ar$ , e, se lo stato  $q$  è finale, ha anche la regola terminale  $q \rightarrow \epsilon$ .

È immediato vedere che vi è corrispondenza biunivoca tra i calcoli dell'automa e le derivazioni della grammatica: una stringa  $x$  è accettata dall'automa se, e solo se, esiste la derivazione  $q_0 \xrightarrow{*} x$ .

*Esempio 3.9.* Dall'automa alla grammatica lineare a destra.

L'automa e la grammatica equivalente sono:



La frase  $bca$ , riconosciuta in 3 passi dall'automa, deriva dall'assioma in  $3 + 1$  passi:

<sup>5</sup>Esistono tuttavia algoritmi più efficienti per decidere l'equivalenza di due automi senza prima minimizzarli.

$$q_0 \Rightarrow bq_1 \Rightarrow bcq_0 \Rightarrow bcaq_0 \Rightarrow bca\epsilon$$

Si osservi che la grammatica possiede regole vuote, ma essa può essere convertita nella forma non annullabile. Un modo è di applicare la trasformazione di p. 60.

Nell'es. l'insieme dei nonterminali annullabili è  $Null = \{q_0, q_2\}$ , da cui si costruiscono le regole equivalenti:

$$q_0 \rightarrow aq_0 \mid bq_1 \mid a \mid \epsilon \quad q_1 \rightarrow cq_0 \mid aq_2 \mid bq_2 \mid a \mid b \mid c$$

Ma  $q_2$ , avendo solo la regola vuota, è da eliminare, e la conseguente ripulitura produce la grammatica

$$q_0 \rightarrow aq_0 \mid bq_1 \mid a \mid \epsilon \quad q_1 \rightarrow cq_0 \mid a \mid b \mid c$$

Si noti che la regola nulla  $q_0 \rightarrow \epsilon$  non può essere tolta perché  $q_0$  è l'assioma, ossia la stringa vuota sta nel linguaggio.

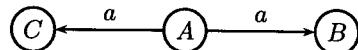
In questa versione della grammatica è facile vedere che, a una mossa entrante in uno stato finale  , possono corrispondere le due regole  $q \rightarrow ar \mid a$ , una terminale, l'altra di transizione verso lo stato  $r$ .

Se il passaggio dall'automa alla grammatica equivalente è stato immediato, quello inverso, dalla grammatica all'automa, richiederà la modifica della definizione della macchina per incorporarvi il concetto di indeterminismo.

### 3.5 Automi indeterministici

In una grammatica lineare a destra possono trovarsi due alternative

$$A \rightarrow aB \mid aC \quad \text{dove } a \in \Sigma, A, B, C \in V$$

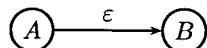


inizianti con lo stesso carattere  $a$ . La precedente corrispondenza, tra regole e mosse dell'automa, porta a tracciare due frecce con la stessa etichetta, dallo stato  $A$  a due stati diversi  $B$  e  $C$ .

Nello stato  $A$ , leggendo il carattere  $a$ , la macchina può scegliere in quale degli stati andare, e il suo comportamento non è deterministico. Formalmente la funzione di transizione prende due valori  $\delta(A, a) = \{B, C\}$ .

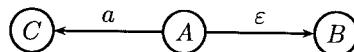
Similmente una regola grammaticale di copiatura

$$A \rightarrow B \quad \text{dove } B \in V$$



porta a disegnare la transizione dallo stato  $A$  allo stato  $B$ , senza lettura d'un carattere, o, che è lo stesso, con la lettura della stringa vuota.

Una mossa che avviene senza leggere un carattere d'ingresso è detta *spontanea* o *mossa epsilon*. Anche le mosse spontanee possono rendere indeterministico il funzionamento, come nel caso



dove nello stato  $A$  l'automa può scegliere di andare spontaneamente (cioè senza lettura) in  $B$ , oppure di leggere un carattere  $a$ , se esso è  $a$ , di andare in  $C$ .

### 3.5.1 Motivazioni dell'indeterminismo

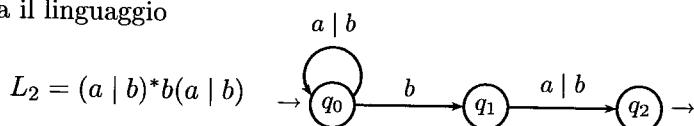
Si è appena visto che la corrispondenza tra grammatiche e automi introduce in modo naturale le transizioni con più stati d'arrivo e le mosse spontanee, le due principali situazioni indeterministiche. Ma, poiché il concetto di indeterminismo potrebbe apparire lontano dalla pratica, è opportuno motivarlo ulteriormente.

#### Concisione

La definizione di un linguaggio mediante una macchina non deterministica, può risultare più leggibile e compatta, come mostra il prossimo esempio.

*Esempio 3.10.* Penultimo carattere.

Sono da definire le stringhe, come  $abaabb$ , aventi una  $b$  nella penultima posizione, ossia il linguaggio



L'automa indeterministico  $N_2$  opera nel seguente modo: data una stringa, cerca di trovare un cammino, dallo stato iniziale a quello finale, etichettato con la stringa data; in caso di successo, la stringa è accettata. Così la stringa  $baba$  è riconosciuta dal calcolo

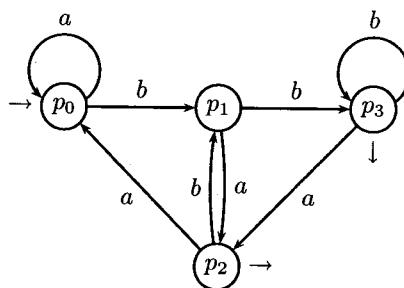
$$q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2$$

pur se altri calcoli possibili, come

$$q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0$$

falliscono perché non conducono allo stato finale.

Lo stesso linguaggio è accettato dall'automa deterministico  $M_2$ :



Non solo questa macchina ha un numero maggiore di stati, ma essa non rende altrettanto evidente la condizione che il penultimo carattere sia  $b$ .

Generalizzando l'esempio, dal linguaggio  $L_2$  al linguaggio  $L_k$  tale che il  $k$ -ultimo,  $k \geq 2$ , carattere sia  $b$ , si vede subito che l'automa indeterministico ha  $k+1$  stati, mentre si potrebbe dimostrare che il numero di stati dell'automa deterministico minimo è dato da una funzione che cresce esponenzialmente con  $k$ : una dimostrazione che l'indeterminismo può rendere molto più concise le definizioni.

### Dualità sinistra - destra

Un'altra situazione che conduce all'indeterminismo nasce nel passaggio dal riconoscitore deterministico d'un linguaggio  $L$  al riconoscitore del linguaggio speculare  $L^R$ , quello che scandisce il testo dalla fine all'inizio (come talvolta è richiesto). Tale riconoscitore si ottiene facilmente nel seguente modo: si scambiano gli stati iniziali e finali<sup>6</sup> e si inverte il senso delle frecce. Entrambe le operazioni possono causare situazioni indeterministiche

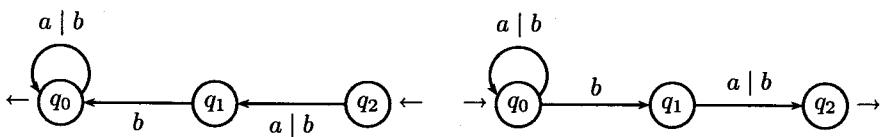
*Esempio 3.11.* Il linguaggio avente  $b$  come penultimo carattere ( $L_2$  dell'es. precedente) è l'immagine riflessa del linguaggio in cui il secondo carattere è  $b$

$$L' = \{x \in \{a, b\}^* \mid b \text{ è il secondo carattere di } x\} \quad L_2 = (L')^R$$

$L'$  è riconosciuto dall'evidente automa deterministico (a sinistra):

---

<sup>6</sup>Se l'automa ha più stati finali, gli stati iniziali sono multipli, come si vedrà.



Applicando all'automa deterministico la trasformazione speculare, si ottiene l'automa indeterministico (a destra), che coincide con quello di p. 108.

Come ultima motivazione, si anticipa che il passaggio attraverso gli automi indeterministici è utilizzato nella costruzione del riconoscitore del linguaggio definito da un'espressione regolare.

### 3.5.2 Riconoscimento indeterministico

Si precisano qui i concetti del calcolo non deterministico a stati finiti, ignorando per ora le mosse spontanee.

Un *automa indeterministico*  $N$  a stati finiti, senza mosse spontanee, è definito da

- l'insieme degli stati  $Q$
- l'alfabeto terminale  $\Sigma$
- due sottoinsiemi di  $Q$ : l'insieme  $I$  degli stati *iniziali* e l'insieme  $F$  di quelli *finali*
- la relazione di transizione  $\delta$ , sottoinsieme del prodotto cartesiano  $Q \times \Sigma \times Q$ .

Note. Questa macchina può avere più d'uno stato iniziale. Nella rappresentazione grafica una transizione  $(q_1, a, q_2)$  è disegnata come un arco etichettato con  $a$ , dal primo nodo al secondo.

Al solito, un *calcolo* della macchina è una serie di transizioni tali che l'origine di ciascuna coincide con la destinazione della precedente

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \dots \xrightarrow{a_n} q_n$$

L'*origine* del calcolo è  $q_0$ , il *termine* è  $q_n$  e la *lunghezza* è il numero di transizioni  $n$ .

I calcoli di lunghezza 1 sono le transizioni. L'*etichetta* del calcolo è il concatenamento  $a_1 a_2 \dots a_n$  dei caratteri letti a ogni transizione.

Il calcolo, in forma abbreviata, è denotato dalla scrittura  $q_0 \xrightarrow{a_1 a_2 \dots a_n} q_n$ .

Una stringa  $x$  è *riconosciuta* (o accettata) dall'automa se essa è l'*etichetta* d'un calcolo che ha origine in uno stato iniziale, termina in uno stato finale e ha  $x$  come etichetta.

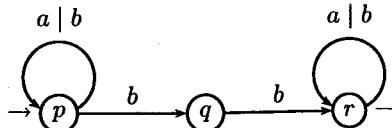
Convenzionalmente, ogni stato è origine e termine d'un calcolo di lunghezza 0, avente come etichetta la stringa vuota  $\varepsilon$ . Di conseguenza la stringa vuota

è accettata dall'automa se, e soltanto se, uno stato iniziale è anche finale. Il *linguaggio*  $L(N)$  riconosciuto dall'automa  $N$  è l'insieme delle stringhe riconosciute

$$L(N) = \{x \in \Sigma^* \mid q \xrightarrow{x} r \text{ con } q \in I, r \in F\}$$

*Esempio 3.12.* Ricerca d'una parola in un testo.

Data una parola  $y$ , per riconoscere se un testo la contiene, basta sottoporlo all'automa che accetta il linguaggio  $(a \mid b)^*y(a \mid b)^*$ . Si presenta per la parola  $y = bb$  l'automa:



La stringa  $abbb$  è l'etichetta di più calcoli originanti nello stato iniziale:

$$\begin{array}{ll} p \xrightarrow{a} p \xrightarrow{b} p \xrightarrow{b} p \xrightarrow{b} p & p \xrightarrow{a} p \xrightarrow{b} p \xrightarrow{b} p \xrightarrow{b} q \\ p \xrightarrow{a} p \xrightarrow{b} p \xrightarrow{b} q \xrightarrow{b} r & p \xrightarrow{a} p \xrightarrow{b} q \xrightarrow{b} r \xrightarrow{b} r \end{array}$$

I primi due calcoli non trovano la parola cercata. Gli ultimi due la trovano rispettivamente nelle posizioni  $\underline{ab}$   $\underline{bb}$  e  $a \underline{bb} b$ .

### Funzione di transizione

Le mosse dell'automa possono essere definite tramite una funzione di transizione, pur di ammettere funzioni a più valori.

Per un automa indeterministico  $N = (Q, \Sigma, \delta, I, F)$ , privo di mosse spontanee, la funzione  $\delta$  è definita nel dominio e codominio:

$$\delta : (Q \times (\Sigma \cup \varepsilon)) \rightarrow \text{parti finite di } Q$$

Per un carattere terminale  $a$ , il significato di

$$\delta(q, a) = [p_1, p_2, \dots, p_k]$$

è che la macchina, dallo stato presente  $q$ , leggendo  $a$ , può arbitrariamente andare in uno dei  $k$  stati  $p_1, \dots, p_k$ .

Si definisce poi la funzione anche per una stringa  $y$  qualsiasi:

$$\forall q \in Q : \delta(q, \varepsilon) = [q]$$

$$\forall q \in Q, y \in \Sigma^* : \delta(q, y) = [p \mid q \xrightarrow{y} p]$$

ovvero  $p \in \delta(q, y)$  se vi è un calcolo etichettato  $y$ , da  $q$  a  $p$ .

Mediante la funzione di transizione, si scrive la seguente definizione del linguaggio accettato dall'automa  $N$ :

$$L(N) = \{x \in \Sigma^* \mid \exists q \in I : \delta(q, x) \cap F \neq \emptyset\}$$

cioè l'insieme calcolato dalla funzione di transizione deve contenere uno stato finale, affinché la stringa  $x$  sia accettata.

*Esempio 3.13.* (Esempio 3.12 continuato.)

$$\delta(p, a) = [p], \quad \delta(p, ab) = [p, q], \quad \delta(p, abb) = [p, q, r]$$

### 3.5.3 Automi con mosse spontanee

Un altro comportamento indeterministico è quello d'un automa che cambia stato, pur senza leggere un carattere, compiendo così una mossa spontanea. Essa nel diagramma stato-transizione è rappresentata da un arco etichettato  $\varepsilon$  (o  $\varepsilon$ -arco).

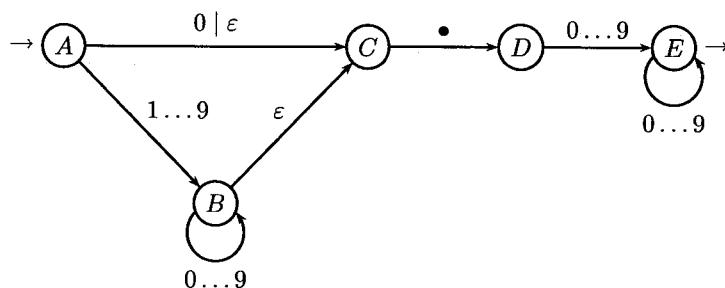
Con l'uso di  $\varepsilon$ -archi è facile costruire i riconoscitori di linguaggi ottenuti per composizione regolare di altri linguaggi. Il prossimo esempio mostra il caso dell'unione e del concatenamento.

*Esempio 3.14.* Costanti decimali.

Il linguaggio comprende le costanti quali ad es.  $90 \bullet 01$ . La parte intera a sinistra del punto decimale può essere 0 o scomparire (es.  $\bullet 01$ ); ma non può contenere zeri non significativi. Il linguaggio è prodotto dalla e.r.

$$L = (0 \mid \varepsilon \mid N) \bullet (0 \dots 9)^+ \text{ dove è } N = (1 \dots 9)(0 \dots 9)^*$$

Il seguente automa riproduce direttamente la struttura della definizione.



Si noti che, scegliendo la mossa spontanea da  $A$  a  $C$ , scompare la parte intera; l'altra mossa spontanea, da  $B$  a  $C$ , impone che la parte intera  $N$  sia seguita dal punto decimale. La stringa  $34 \bullet 5$  è accettata dal calcolo

$$A \xrightarrow{3} B \xrightarrow{4} B \xrightarrow{\varepsilon} C \xrightarrow{\bullet} D \xrightarrow{5} E$$

La presenza di mosse spontanee non modifica il concetto di riconoscimento: una stringa  $x$  è *riconosciuta* dall'automa con mosse spontanee se essa è l'etichetta d'un calcolo che ha origine in uno stato iniziale, termina in uno stato finale e ha  $x$  come etichetta.

Ora però la lunghezza del calcolo (ossia il numero degli archi del cammino percorso nel grafo) può superare quella della stringa, a causa della presenza

di  $\epsilon$ -archi. Di conseguenza il numero di passi dell'algoritmo può superare la lunghezza della stringa sorgente, e di conseguenza l'automa non funziona più in tempo reale. Esso però ha sempre una complessità di calcolo lineare, perché si può escludere che vengano eseguiti dei cicli di mosse spontanee.

### Unicità dello stato iniziale

Nelle definizioni l'automa indeterministico può avere più stati iniziali, ma è facile ottenere un automa equivalente con stato iniziale unico. Basta aggiungere il nuovo stato  $q_0$  iniziale, che sarà l'unico, e le  $\epsilon$ -mosse che da esso vanno agli stati ex iniziali dell'automa da trasformare. Chiaramente un calcolo del nuovo automa accetta una stringa se, e soltanto se, anche il vecchio automa la accetta.

La mosse spontanee così aggiunte potranno poi essere eliminate, come si vedrà a p. 119.

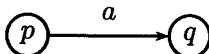
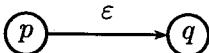
#### 3.5.4 Corrispondenza tra automa e grammatica

Si riepilogano le corrispondenze tra automi indeterministici, anche con mosse spontanee, e grammatiche unilineari, confermando che la differenza tra i due modelli è una mera variante descrittiva.

Confrontando dunque una grammatica e un automa, si esplicitano le condizioni che rendono equivalenti le loro regole e mosse rispettive.

Denotiamo con  $G = (V, \Sigma, P, S)$  la grammatica lineare a destra e con  $N = (Q, \Sigma, \delta, q_0, F)$  l'automa. Possiamo supporre, per quanto detto sopra, che lo stato iniziale sia unico.

Si suppone inizialmente che le regole della grammatica siano strettamente unilineari (p. 71). Ecco la corrispondenza:

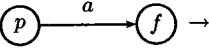
|   | <i>Grammatica lineare a destra</i>                      | <i>Automa finito</i>  |
|---|---|---|
| 1 | Alfabeto nonterminale $V$                               | Insieme degli stati $Q = V$   |
| 2 | Assioma $S = q_0$                                       | Stato iniziale $q_0 = S$  |
| 3 | $p \rightarrow aq$ , dove $a \in \Sigma$ e $p, q \in V$ |                |
| 4 | $p \rightarrow q$ , dove $p, q \in V$                   |                |
| 5 | $p \rightarrow \epsilon$                                | Stato finale  |

Gli stati  $Q$  dell'automa sono in corrispondenza biunivoca con i simboli non-terminali  $V$  della grammatica. Lo stato iniziale corrisponde all'assioma.

Si noti (riga 3) che una coppia di alternative  $p \rightarrow aq \mid ar$  corrisponde a due mosse non deterministiche. Una regola di copiatura (riga 4) corrisponde a una mossa spontanea. Uno stato è finale (riga 5) se corrisponde a un nonterminale avente una regola vuota.

È facile constatare che ogni derivazione della grammatica corrisponde a un calcolo dell'automa, e viceversa; di conseguenza i due modelli definiscono lo stesso linguaggio.

**Proprietà 3.15.** Un linguaggio è riconosciuto da un automa finito se, e solo se, è generato da una grammatica unilineare.

Se la grammatica possiede anche delle regole terminali del tipo  $p \rightarrow a$ , dove  $a \in \Sigma$ , l'automa conterrà anche uno stato finale  $f$ , distinto da quelli prodotti dalla riga 5 della tabella, e la mossa 

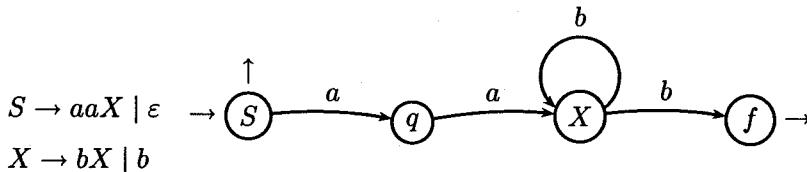
**Esempio 3.16.** Equivalenza tra grammatiche lineari a destra e automi.

La grammatica corrispondente all'automa delle costanti decimali di p. 112 è:

$$\begin{array}{ll} A \rightarrow 0C \mid C \mid 1B \mid \dots \mid 9B & B \rightarrow 0B \mid \dots \mid 9B \mid C \\ C \rightarrow \bullet D & D \rightarrow 0E \mid \dots \mid 9E \\ E \rightarrow 0E \mid \dots \mid 9E \mid \epsilon & \end{array}$$

dove  $A$  è l'assioma.

Come secondo esempio, nella grammatica lineare a destra



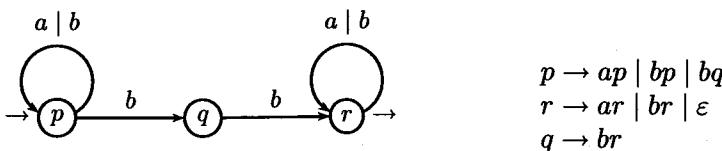
anche se la regola  $S \rightarrow aaX$  non è strettamente lineare, un semplice accorgimento ne permette la traduzione (mostrata a destra) in una serie di due archi: basta creare uno stato intermedio  $q$  dopo la prima  $a$ . Inoltre si crea il nuovo stato finale  $f$  per riprodurre l'effetto della regola  $X \rightarrow b$ .

### 3.5.5 Ambiguità dell'automa

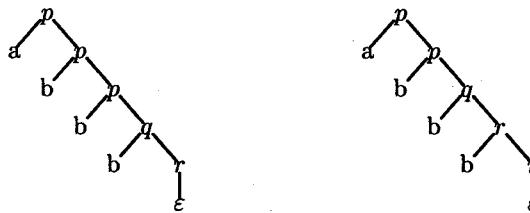
Un *automa* è detto *ambiguo* se accetta una frase con due calcoli diversi. Grazie alla corrispondenza biunivoca tra i calcoli e le derivazioni della grammatica, si può dire che un automa è ambiguo se, e soltanto se, la grammatica lineare a destra corrispondente è essa stessa ambigua, ossia se genera una frase con due alberi sintattici diversi.

*Esempio 3.17.* Ambiguità dell'automa e della grammatica.

Nell'esempio 3.12 di p. 111 si è visto che l'automa



riconosce la frase  $abbb$  con due calcoli. La corrispondente grammatica (a destra) genera la stessa frase con i due alberi:



### 3.5.6 Grammatica lineare a sinistra e automa

Poiché si sa che la famiglia  $REG$  è pure definita dalle grammatiche lineari a sinistra, si possono enunciare le regole di corrispondenza tra le grammatiche di tale tipo e gli automi finiti, seguendo un principio di dualità tra sinistra e destra.

Ora le regole della grammatica  $G$  hanno le forme:

$$A \rightarrow Ba, \quad A \rightarrow B, \quad A \rightarrow \epsilon$$

Si consideri il linguaggio speculare  $L^R = (L(G))^R$ : esso è generato dalla grammatica riflessa, denotata  $G_R$ , ottenuta (p. 79) cambiando le regole della prima forma in  $A \rightarrow aB$ , mentre le altre due forme non mutano. Poiché evidentemente la grammatica riflessa  $G_R$  è lineare a destra, si sa costruire il riconoscitore  $N_R$  di  $L^R$ . Non resta che trasformarlo, invertendo le frecce e scambiando stati finali e iniziali, per ottenere il riconoscitore del linguaggio originale  $L$ .

*Esempio 3.18.* Dalla grammatica lineare a sinistra all'automa.

Data la grammatica  $G$ :

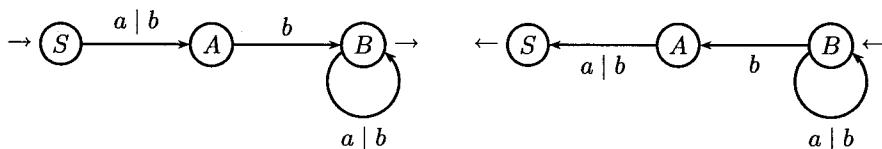
$$S \rightarrow Aa \mid Ab \quad A \rightarrow Bb \quad B \rightarrow Ba \mid Bb \mid \epsilon$$

Invertendo specularmente le regole si ottiene la grammatica riflessa  $G_R$ :

$$S \rightarrow aA \mid bA \quad A \rightarrow bB \quad B \rightarrow aB \mid bB \mid \epsilon$$

dalla quale si costruisce l'automa  $N^R$  e, con l'inversione delle frecce e lo scambio tra stati iniziali e finali, il riconoscitore  $N$  del linguaggio  $L(G)$ :

*Riconoscitore  $N_R$  di  $(L(G))^R$*       *Riconoscitore  $N$  di  $L(G)$*



Per inciso, il linguaggio è quello in cui il penultimo carattere è  $b$ .

### 3.6 Dall'automa all'espressione regolare direttamente: il metodo BMC

Poiché un automa finito equivale a una grammatica lineare a destra, e questa genera un linguaggio regolare, l'e.r. del linguaggio riconosciuto dall'automa può essere ottenuta risolvendo le equazioni lineari con il procedimento di p. 71, ma si offrirà qui anche un percorso diretto, il metodo di eliminazione di Brzozowski e McCluskey (BMC).

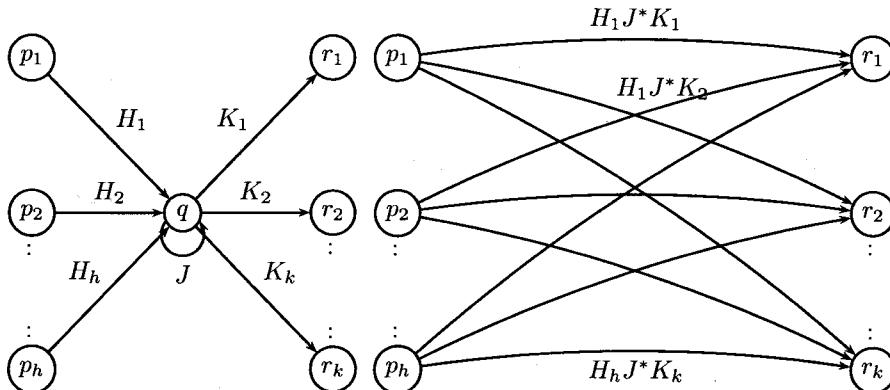
Si suppone che lo stato iniziale  $i$  sia unico e privo di archi entranti, e lo stato finale  $t$  sia unico e privo di archi uscenti. Se così non fosse, basterebbe creare il nuovo stato  $i$  e collegarlo con mosse spontanee agli stati ex-iniziali dell'automa; e similmente per il nuovo stato finale  $t$ .

Gli stati diversi da  $i$  e da  $t$  sono detti *interni*.

Si costruirà un automa equivalente a quello dato, in cui però gli archi possono essere etichettati non soltanto con caratteri terminali, ma anche con linguaggi regolari. Esso è chiamato *automa generalizzato*.

L'idea è di eliminare uno alla volta gli stati interni, aggiungendo delle mosse compensatorie, che preservano il linguaggio riconosciuto, etichettate con e.r.; ciò fino a quando restano soltanto gli stati iniziale e finale. A quel punto l'etichetta dell'arco  $i \rightarrow t$  è la e.r. del linguaggio.

Si veda, a sinistra, uno stato interno  $q$  con gli archi incidenti; a destra, lo stesso frammento di automa, dopo l'eliminazione dello stato  $q$  e l'aggiunta di archi etichettati con le stesse stringhe prodotte dall'attraversamento di  $q$ :



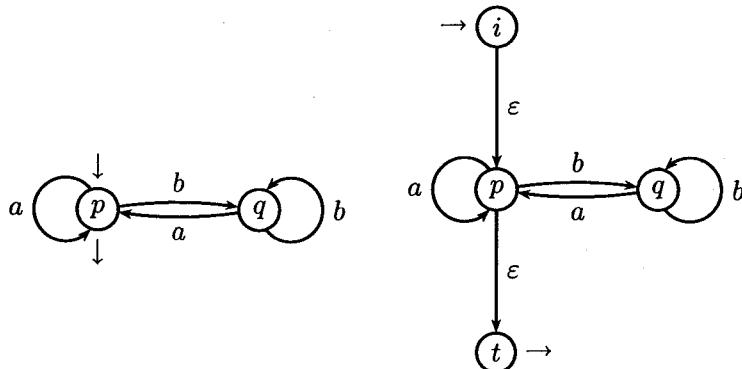
Per non complicare la figura, alcune etichette sono omesse, chiarendo che, per ogni coppia di stati  $p_i, r_j$ , vi è l'arco  $p_i \xrightarrow{H_i J^* K_j} r_j$ .

Si noti che alcuni stati  $p_i, r_j$  potrebbero essere coincidenti.

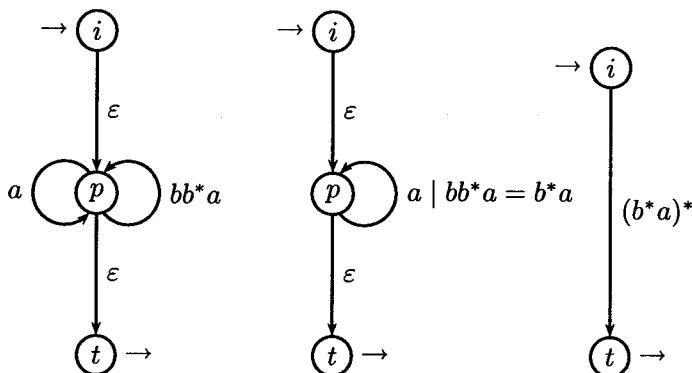
È evidente che l'insieme delle stringhe, che possono essere lette dall'automa originale passando da uno stato  $p_i$  a uno stato  $r_j$ , coincide con il linguaggio che etichetta l'arco  $p_i \rightarrow r_j$  dell'automa trasformato.

Al fine di calcolare la e.r. del linguaggio riconosciuto da un automa, si applica la trasformazione tante volte, quanti sono gli stati interni, ottenendo al termine un automa con i soli stati iniziale e finale, un solo arco e la e.r. del linguaggio come sua etichetta.

*Esempio 3.19.* (Sakarovitch). L'automa dato è mostrato prima e dopo la normalizzazione:



Si applica l'algoritmo BMC nell'ordine  $q, p$ :



L'ordine di eliminazione degli stati interni è irrilevante, ma, come già nella risoluzione dei sistemi di equazioni lineari, ordini diversi possono produrre e.r. di diversa complessità, pur se equivalenti. Così nell'esempio precedente l'ordine  $p, q$  produrrebbe la e.r.  $(a^*b)^+a^+ \mid a^*$ .

### 3.7 Eliminazione dell'indeterminismo

Pur se in certe situazioni la formulazione più diretta del linguaggio desiderato è quella non deterministica, la versione finale del riconoscitore del linguaggio deve quasi sempre essere deterministica, per ragioni di efficienza. Fanno eccezione quei casi in cui il costo da minimizzare è il tempo di costruzione dell'automa invece del tempo di riconoscimento delle frasi, come ad es. quando l'automa viene usato in videoscrittura per ricercare nel testo una sola volta una parola specificata dall'utente.

Si mostrerà ora, attraverso un procedimento costruttivo, che ogni automa indeterministico può essere trasformato in uno deterministico equivalente, e, come corollario, che ogni grammatica unilineare ammette una grammatica equivalente non ambigua.

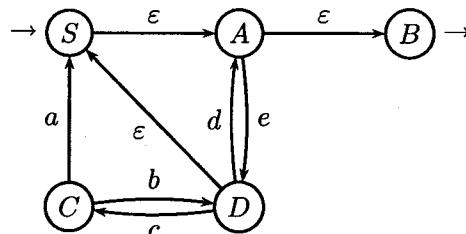
La trasformazione di un automa indeterministico in uno deterministico procede in due fasi:

1. Eliminazione delle mosse spontanee, ottenendo un automa in generale indeterministico: poiché le mosse spontanee corrispondono alle regole di copiatura della grammatica, si può applicare la trasformazione grammaticale che elimina queste ultime (p. 61).
2. Sostituzione di più transizioni non deterministiche con una sola che porta in un nuovo stato: questa fase è detta costruzione delle parti finite, perché i nuovi stati introdotti sono in corrispondenza con i sottoinsiemi dell'insieme degli stati.

### *Eliminazione delle mosse spontanee*

Per la prima fase si passa subito all'esemplificazione, essendo l'applicazione della trasformazione grammaticale che elimina le regole di copiatura.

*Esempio 3.20.* Dato l'automa



si devono eliminare le regole di copiatura<sup>7</sup> dalla grammatica equivalente:

$$\begin{array}{lll} S \rightarrow A & A \rightarrow B \mid eD & B \rightarrow \epsilon \\ C \rightarrow aS \mid bD & D \rightarrow S \mid cC \mid dA & \end{array}$$

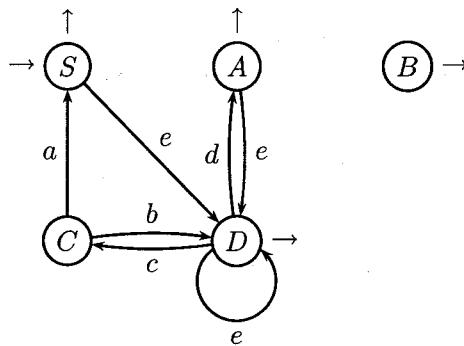
Calcolati gli insiemi delle copie

|          | <i>copia</i>      |
|----------|-------------------|
| <i>S</i> | <i>S, A, B</i>    |
| <i>A</i> | <i>A, B</i>       |
| <i>B</i> | <i>B</i>          |
| <i>C</i> | <i>C</i>          |
| <i>D</i> | <i>D, S, A, B</i> |

si ottiene la grammatica e il corrispondente automa senza mosse spontanee:

<sup>7</sup>Pur in presenza di  $\epsilon$ -regole, l'algoritmo di p. 61 resta valido, poiché in una grammatica lineare ogni parte destra contiene non più d'un nonterminale.

$$\begin{array}{l} S \rightarrow \varepsilon \mid eD \\ C \rightarrow aS \mid bD \end{array} \quad \begin{array}{l} A \rightarrow \varepsilon \mid eD \\ D \rightarrow \varepsilon \mid eD \mid cC \mid dA \end{array} \quad \begin{array}{l} B \rightarrow \varepsilon \end{array}$$



dove lo stato  $B$ , irraggiungibile dallo stato iniziale, è da eliminare.

Si noti che nel grafo dell'automa un intero cammino (ad es.  $D \xrightarrow{\varepsilon} S \xrightarrow{\varepsilon} A \xrightarrow{e} D$ ), composto di una catena di mosse spontanee concluse da una mossa di lettura, è sostituito da una mossa diretta ( $D \xrightarrow{e} D$ ). Se, dopo l'eliminazione delle mosse spontanee, l'automa risulta indeterministico, si applica la fase seguente.

### 3.7.1 Costruzione delle parti finite raggiungibili

Dato  $N$ , un automa indeterministico privo di mosse spontanee, si costruisce l'automa deterministico equivalente  $M'$ .

La costruzione si basa su quest'idea: se in  $N$  vi sono le mosse

$$p \xrightarrow{a} p_1, \quad p \xrightarrow{a} p_2, \quad \dots, \quad p \xrightarrow{a} p_k$$

poiché, dopo la lettura di  $a$ , non si sa in quale degli stati successori  $p_1, p_2, \dots, p_k$  la macchina  $N$  si trovi, si crea in  $M'$  un nuovo stato collettivo, denominato

$$[p_1, p_2, \dots, p_k]$$

per indicare l'incertezza tra i  $k$  stati.

Si costruiscono poi gli archi uscenti dagli stati collettivi, nel modo seguente. Se uno stato collettivo contiene gli stati  $p_1, p_2, \dots, p_k$ , per ognuno di essi si considerano in  $N$  gli archi uscenti, etichettati con la stessa lettera  $a$

$$p_1 \xrightarrow{a} [q_1, q_2, \dots], \quad p_2 \xrightarrow{a} [r_1, r_2, \dots], \quad \text{ecc.}$$

e si uniscono gli stati di arrivo

$$[q_1, q_2, \dots] \cup [r_1, r_2, \dots] \cup \dots$$

ottenendo lo stato collettivo di arrivo della transizione

$$[p_1, p_2, \dots, p_k] \xrightarrow{a} [q_1, q_2, \dots, r_1, r_2, \dots, \dots]$$

Se tale stato ancora non esiste in  $M'$ , lo si crea.

Algoritmo 3.21. Algoritmo dell'insieme delle parti finite.

L'automa  $M'$  deterministico equivalente a  $N$  ha:

1. gli stati  $Q' = \mathcal{P}(Q)$ , l'insieme delle parti finite di  $Q$
2. gli stati finali  $F' = \{p' \in Q' \mid p' \cap F \neq \emptyset\}$ , quelli contenenti uno stato finale di  $N$
3. lo stato iniziale<sup>8</sup>  $[q_0]$
4. la funzione di transizione  $\delta'$ :  
per ogni  $p' \in Q'$  e per ogni  $a \in \Sigma$

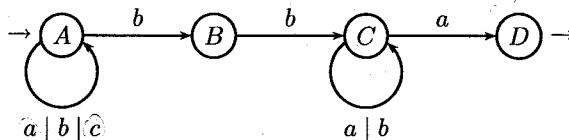
$$p' \xrightarrow{a} [s \mid q \in p' \wedge (\text{l'arco } q \xrightarrow{a} s \text{ è in } N)]$$

Al passo 4., se l'arco  $q \xrightarrow{a} q_{err}$  porta nello stato d'errore, esso non va aggiunto allo stato collettivo; infatti i calcoli che portano nel pozzo non riconoscono alcuna stringa e possono essere ignorati.

Poiché gli stati di  $M'$  sono i sottoinsiemi di  $Q$ , la cardinalità di  $Q'$  è, nel caso peggiore, un esponenziale di quella di  $Q$ , a comprova della maggiore dimensione della macchina deterministica. Si ricordi l'esplosione esponenziale del numero degli stati nel linguaggio in cui il  $k$ -ultimo carattere è fissato (p. 108).

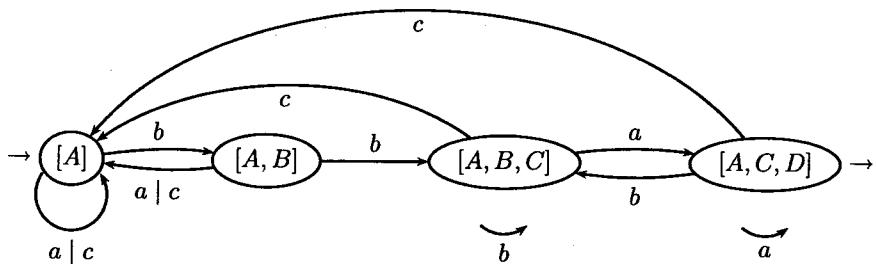
L'algoritmo può essere migliorato:  $M'$  spesso ha degli stati irraggiungibili dallo stato iniziale, dunque inutili. Invece di eliminarli, ripulendo l'automa a posteriori, conviene evitarne la creazione: si disegnano soltanto gli stati che sono raggiungibili dallo stato iniziale.

Esempio 3.22. Dato l'automa indeterministico



ecco l'automa deterministico equivalente:

<sup>8</sup>Se l'automa dato  $N$  ha più stati iniziali, lo stato iniziale di  $M'$  è l'insieme di tutti gli stati iniziali.



Spiegazioni: essendo  $\delta(A, b) = \{A, B\}$  si crea lo stato collettivo  $[A, B]$ , da cui partono le transizioni

$$[A, B] \xrightarrow{a} (\delta(A, a) \cup \delta(B, a)) = [A]$$

$$[A, B] \xrightarrow{b} (\delta(A, b) \cup \delta(B, b)) = [A, B, C]$$

Poi si creano le transizioni uscenti dal nuovo stato collettivo  $[A, B, C]$ :

$$[A, B, C] \xrightarrow{a} (\delta(A, a) \cup \delta(B, a) \cup \delta(C, a)) = [A, C, D], \text{ stato collettivo finale}$$

$$[A, B, C] \xrightarrow{b} (\delta(A, b) \cup \delta(B, b) \cup \delta(C, b)) = [A, B, C], \text{ autoanello}$$

ecc.

Si termina quando il passo 4, applicato agli stati di  $Q'$  finora creati, non crea nuovi stati.

Si noti che non tutti i sottoinsiemi di  $Q$  corrispondono a stati raggiungibili: ad es. lo stato  $[A, C]$  è inutile.

Per giustificare la validità del procedimento, si dimostra che una stringa  $x$  è riconosciuta da  $M'$  se, e solo se, è accettata da  $M$ .

Se un calcolo di  $M$  accetta  $x$ , esiste un cammino etichettato  $x$  dallo stato iniziale  $q_0$  a uno stato finale  $q_f$ . L'algoritmo garantisce allora che anche in  $M'$  esiste un cammino etichettato  $x$  da  $[q_0]$  a uno stato  $[\dots, q_f, \dots]$  contenente  $q_f$ . Viceversa se  $x$  è l'etichetta d'un calcolo valido di  $M'$ , da  $[q_0]$  a uno stato finale  $p \in F'$ , allora per costruzione  $p$  contiene almeno uno stato finale  $q_f$  di  $M$ . Per costruzione esiste allora anche in  $M$  un cammino di etichetta  $x$  da  $q_0$  a  $q_f$ . In conclusione vale la seguente proprietà fondamentale.

**Proprietà 3.23.** Ogni linguaggio a stati finiti è riconosciuto da un automa deterministico.

Questa proprietà testimonia che l'algoritmo di riconoscimento dei linguaggi a stati finiti opera in tempo reale, ossia richiede un numero di transizioni eguale o minore del numero dei caratteri della stringa (minore se un errore si manifesta prima del completamento della lettura della stringa).

Come corollario si ha che, per ogni linguaggio riconosciuto da un automa a stati finiti, esiste una grammatica unilineare priva di ambiguità, quella che

corrisponde in modo naturale (p. 106) all'automa deterministico. Di conseguenza per i linguaggi regolari si dispone d'un procedimento per eliminare l'ambiguità, attraverso la costruzione del riconoscitore deterministico e della grammatica a esso equivalente.

## 3.8 Dall'espressione regolare all'automa riconoscitore

Se per definire un linguaggio si usa un'espressione regolare, invece d'una grammatica unilineare, resta da esporre un procedimento per costruire l'automa. Il problema è assai diffuso nelle applicazioni, come la compilazione o l'analisi dei documenti, e tanti algoritmi sono stati inventati. Essi si diversificano rispetto alle proprietà (deterministico o indeterministico con o senza mosse spontanee) e alle dimensioni dell'automa prodotto, e rispetto alla complessità algoritmica della costruzione.

Si esporranno due metodi di costruzione. Il primo, noto come metodo di Thompson o strutturale, decomponne l'espressione nelle sue sottoespressioni, fino a giungere agli elementi atomici dell'alfabeto. Costruisce poi i riconoscitori delle varie sottoespressioni e li connette in reti di riconoscitori che realizzano gli operatori (unione, concatenamento, stella, croce) presenti nella e.r..

Il secondo metodo, noto come algoritmo di Glushkov, McNaughton e Yamada, costruisce un riconoscitore indeterministico senza mosse spontanee, ma di dimensioni maggiori di quello costruito col metodo precedente.

Con questi metodi si concluderà il percorso espositivo che mostra che le famiglie dei linguaggi regolari, a stati finiti e unilineari (tipo 3 di Chomsky) sono identiche.

Entrambi i metodi possono essere combinati con gli algoritmi di determinizzazione allo scopo di produrre direttamente macchine deterministiche.

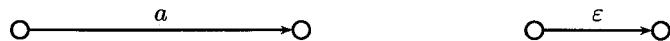
### 3.8.1 Metodo strutturale o di Thompson

Sia data un'espressione regolare; analizzandola nelle sottoespressioni, si costruirà ora in modo sistematico l'automa che riconosce il linguaggio.

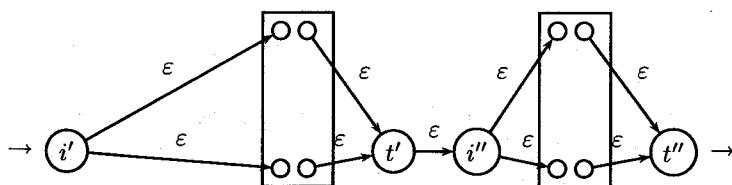
In questa costruzione ogni automa deve avere un solo stato iniziale e un solo stato finale. Se così non fosse, si aggiunge il nuovo stato iniziale  $i$  e lo si collega ai vecchi stati iniziali tramite  $\epsilon$ -archi. Dualmente, se vi fossero più stati finali, si introduce un nuovo stato finale  $t$ , e i vecchi stati finali sono collegati al nuovo stato tramite  $\epsilon$ -archi.

Il procedimento di Thompson [49] sfrutta le regole di corrispondenza tra e.r. e automi riconoscitori di seguito schematizzate.

*Espressioni regolari atomiche:*

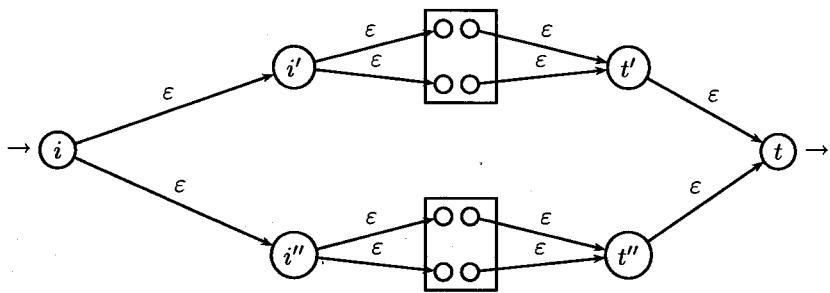


Concatenamento di due e.r.:

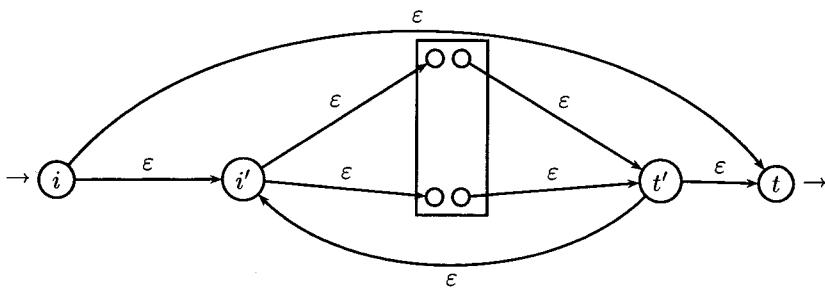


dove il rettangolo schematizza un automa di cui si evidenziano soltanto gli stati iniziali (colonna di sinistra) e finali (colonna di destra).

Unione di due e.r.:



*Stella d'una e.r.:*



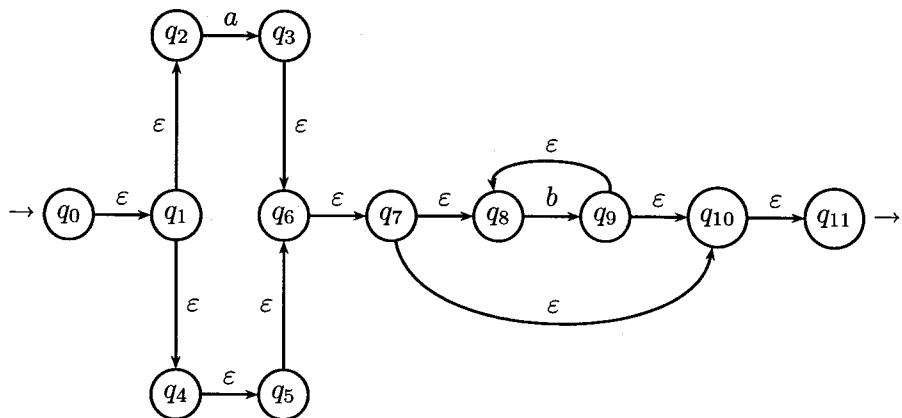
In generale il risultato della costruzione è un automa indeterministico, con archi spontanei.

Per giustificare il procedimento di Thompson, basta osservare che esso è una formulazione operativa delle proprietà di chiusura dei linguaggi regolari rispetto alle operazioni di concatenamento, d'unione e di stella, enunciate nel capitolo precedente (p. 24).

*Esempio 3.24.* Per l'e.r.:

$$(a \cup \epsilon).b^*$$

si disegna l'automa costruito con il metodo strutturale:



Si noti che diversi stati (ad es.  $q_0$  e  $q_{11}$ ) sono ridondanti; potrebbero essere fusi con  $q_1$  e  $q_{10}$  rispettivamente. Esistono versioni perfezionate dell'algoritmo di Thompson che evitano tali ridondanze durante la costruzione.

Altre versioni abbinano l'algoritmo con quello per la eliminazione delle mosse spontanee. Volendo realizzare con un programma questo procedimento, è necessario decomporre una e.r. nelle sue sottoespressioni, produrre i riconoscitori dei linguaggio ad esse associati, e poi collegarli mediante le mosse spontanee. Questo è un tipico problema di analisi e traduzione guidata dalla sintassi o struttura della e.r., affrontabile con i metodi del capitolo 5.

### 3.8.2 Algoritmo di Glushkov, Mc Naughton e Yamada

Un altro metodo classico, GMY, costruisce un automa i cui stati sono in corrispondenza diretta con i caratteri terminali presenti nell'espressione. È necessario preparare il terreno<sup>9</sup> con la definizione d'una particolare sottofamiglia dei linguaggi regolari.

#### Linguaggi localmente testabili e automi locali

Vi sono linguaggi regolari semplicissimi da riconoscere, perché basta controllare la presenza di certe sottostringhe. Un esempio è l'insieme delle stringhe che iniziano con  $b$ , finiscono con  $a$  o  $b$ , e contengono le sottostringhe  $ba$ ,  $ab$ .

**Definizione 3.25.** *Dato un linguaggio  $L$  di alfabeto  $\Sigma$ , l'insieme degli inizi è*

$$Ini(L) = \{a \in \Sigma \mid a\Sigma^* \cap L \neq \emptyset\} \quad (\text{ossia le lettere iniziali di } L)$$

*l'insieme delle fini è*

$$Fin(L) = \{a \in \Sigma \mid \Sigma^* a \cap L \neq \emptyset\} \quad (\text{ossia le lettere finali di } L)$$

*l'insieme dei digrammi è*

$$Dig(L) = \{x \in \Sigma^2 \mid \Sigma^* x \Sigma^* \cap L \neq \emptyset\} \quad (\text{le sottostringhe di lunghezza 2})$$

*e il suo complemento*

$$\overline{Dig}(L) = \Sigma^2 \setminus Dig(L)$$

**Esempio 3.26.** Linguaggio localmente testabile.

Per il linguaggio  $L_1 = (abc)^+$  si ha

$$Ini(L_1) = a \quad Fin(L_1) = c \quad Dig(L_1) = \{ab, bc, ca\}$$

e quindi

$$\overline{Dig}(L_1) = \{aa, ac, ba, bb, cb, cc\}$$

<sup>9</sup>Seguendo il percorso concettuale di Berstel e Pin [10].

Si osservi che i tre insiemi caratterizzano esattamente le frasi del linguaggio, nel senso che vale l'identità

$$L_1 \equiv \{x \mid \text{Ini}(x) \in \text{Ini}(L_1) \wedge \text{Fin}(x) \in \text{Fin}(L_1) \wedge \text{Dig}(x) \subseteq \text{Dig}(L_1)\} \quad (3.1)$$

o equivalentemente

$$L_1 \equiv \{x \mid \text{Ini}(x) \in \text{Ini}(L_1) \wedge \text{Fin}(x) \in \text{Fin}(L_1)\} \setminus \Sigma^* \overline{\text{Dig}}(L_1) \Sigma^* \quad (3.2)$$

**Definizione 3.27.** *Un linguaggio  $L$  è detto locale (o localmente testabile) se per esso vale l'identità 3.1 (o 3.2).*

Chiaramente per qualsiasi linguaggio, purché non contenente la stringa vuota, la formula 3.1 (o 3.2) vale sempre, pur di sostituire il segno di inclusione a quello di identità: infatti ogni frase inizia (risp. termina) necessariamente con un carattere di  $\text{Ini}$  (risp. di  $\text{Fin}$ ) e i suoi digrammi sono inclusi in quelli del linguaggio. Ma tali condizioni possono essere insufficiente per escludere alcune stringhe non valide.

**Esempio 3.28.** Linguaggio non locale.

Nel linguaggio  $L_2 = b(aa)^+b$  si ha

$$\text{Ini}(L_2) = \text{Fin}(L_2) = \{b\} \quad \text{Dig}(L_2) = \{aa, ab, ba\} \quad \overline{\text{Dig}}(L_2) = \{bb\}$$

Le frasi di  $L_2$  hanno lunghezza pari, mentre l'insieme delle frasi, che iniziano e terminano con  $b$  e non contengono il digramma  $bb$ , comprende anche stringhe di lunghezza dispari, come  $baaab$ . Quindi tale linguaggio include strettamente  $L_2$ .

Il riconoscimento d'un linguaggio locale è particolarmente semplice. Scandendo la stringa da sinistra a destra, si controlla che il primo carattere stia in  $\text{Ini}$ , ogni coppia di caratteri adiacenti non stia in  $\overline{\text{Dig}}$  e l'ultimo carattere stia in  $\text{Fin}$ . Il controllo può essere effettuato facendo scorrere una finestra mobile, larga due posizioni, da sinistra a destra sulla stringa, operazione facilmente attuabile da un automa finito deterministico.

#### Algoritmo di costruzione del riconoscitore d'un linguaggio locale

L'automa riconoscitore d'un linguaggio locale, definito dagli insiemi  $\text{Ini}, \text{Fin}, \text{Dig}$ , è definito da:

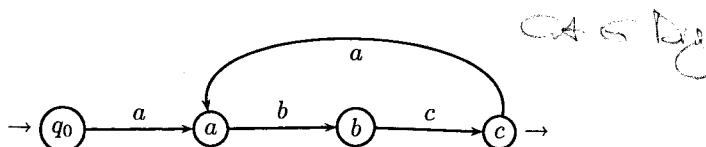
- gli stati  $q_0 \cup \Sigma$ ;
- gli stati finali  $\text{Fin}$ ;
- gli archi  $q_0 \xrightarrow{a} q$  se  $a \in \text{Ini}$ ;
- gli archi  $a \xrightarrow{b} q$  se  $ab \in \text{Dig}$ ;
- se la stringa vuota appartiene al linguaggio, lo stato iniziale  $q_0$  è anche finale.

Le seguenti proprietà caratterizzano i *riconoscitori dei linguaggi locali*:

1. gli stati non iniziali sono in corrispondenza biunivoca con l'alfabeto
2. nessuna freccia entra nello stato iniziale  $q_0$
3. tutte le frecce etichettate con la stessa lettera  $a$  entrano nello stesso stato, quello denominato  $a$ .

Intuitivamente, l'automa si trova nello stato  $b$  se, e solo se, l'ultimo carattere letto è  $b$ .

*Esempio 3.29.* Ecco il riconoscitore del linguaggio locale  $(abc)^+$  dell'es. 3.26:



Grazie alle proprietà elencate, le etichette delle frecce sono pleonastiche, perché sono identiche a quelle dei nodi di destinazione.

### Composizione di linguaggi locali di alfabeti disgiunti

Per applicare l'idea della finestra scorrevole a e.r. generiche, occorre un altro passo concettuale, basato sulla seguente osservazione: le operazioni fondamentali preservano la proprietà dei linguaggi locali, a condizione che i loro alfabeti terminali siano disgiunti.

*Proprietà 3.30.* Dati i linguaggi locali  $L'$  e  $L''$ , di alfabeti disgiunti,  $\Sigma' \cap \Sigma'' = \emptyset$ , risultano locali anche i linguaggi unione  $L' \cup L''$ , concatenamento  $L'.L''$  e stella  $L'^*$  (e anche croce).

Dimostrazione: è immediato costruire per il linguaggio risultante un riconoscitore del tipo locale, combinando quelli dei linguaggi componenti. Siano  $q'_0, q''_0$  i loro rispettivi stati iniziali e  $F', F''$  gli insiemi degli stati finali.

Per l'unione  $L' \cup L''$ :

- lo stato iniziale  $q_0$  è ottenuto fondendo insieme gli stati iniziali  $q'_0$  e  $q''_0$ ;
- gli stati finali sono quelli dei due automi,  $F' \cup F''$ ;
- si noti che, se la stringa vuota appartiene a (almeno) uno dei due linguaggi, si aggiunge  $q_0$  agli stati finali.

Per il concatenamento  $L'.L''$ :

- lo stato iniziale è  $q'_0$
- gli archi sono quelli di  $L'$  uniti a:
  - gli archi di  $L''$ , tranne quelli aventi origine nello stato iniziale  $q''_0$ ;

- in sostituzione di questi ultimi, gli archi  $q' \xrightarrow{a} q''$ , aventi origine in uno stato finale  $q' \in F'$  e destinazione in uno stato  $q''$ , tali che esista per  $L''$  l'arco  $q'' \xrightarrow{a} q''$ .
- gli stati finali sono quelli di  $F''$ , se  $\varepsilon \notin L''$ ;  
gli stati finali  $F'$  uniti a  $F''$ , altrimenti.

Per la stella  $L'^*$ :

- si aggiunge  $q'_0$  agli stati finali;
- per ogni stato finale  $q \in F'$ , si aggiunge l'arco  $q \xrightarrow{a} r$ , se l'automa aveva l'arco  $q'_0 \xrightarrow{a} r$ , con origine nello stato iniziale.

Questa dimostrazione è costruttiva e produce il riconoscitore d'un linguaggio ottenuto unendo, concatenando o iterando dei linguaggi locali, purché di alfabeti disgiunti.

### Procedimento GMY

Si vedrà ora come trasformare una generica e.r. in un linguaggio locale, attraverso un opportuno cambiamento di alfabeto.

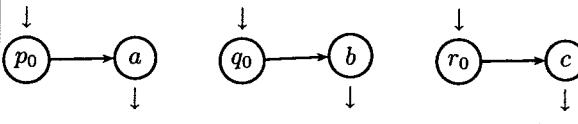
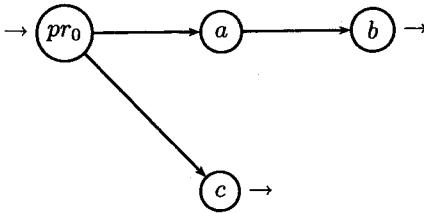
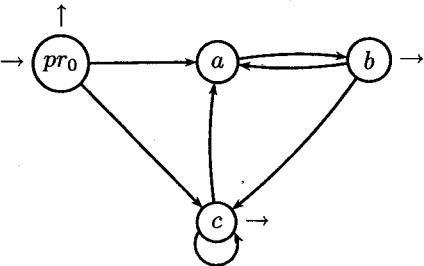
In una e.r. un carattere può ovviamente comparire più volte. La e.r. è detta *lineare* se nessun carattere è ripetuto. Ad es.  $(abc)^*$  è lineare, mentre  $(ab)^*a$  non è lineare, perché il carattere  $a$  compare più volte (pur essendo locale il linguaggio).

*Proprietà 3.31.* Il linguaggio definito da una e.r. lineare è locale.

Dimostrazione. Per l'ipotesi di linearità, le sottoespressioni della e.r. sono di alfabeti disgiunti. Poiché la e.r. è ottenuta per composizione dei linguaggi locali associati alle sue sottoespressioni, il linguaggio da essa definito è locale, grazie alla Proprietà 3.30.

*Esempio 3.32.* Per la e.r. lineare  $(ab \cup c)^*$  seguono i passi della costruzione del riconoscitore, partendo dalle sottoespressioni atomiche:



| <i>sottoespressione</i>                           | <i>automi componenti</i>  |
|---|---|
| <i>elementi</i><br>$a, b, c$<br>•                 |   |
| <i>concatenamento<br/>e unione</i><br>$ab \cup c$ |    |
| <i>stella</i><br>$(ab \cup c)^*$                  |  |

Avendo accertato che il linguaggio d'una e.r. lineare è locale, il problema di costruirne il riconoscitore si riconduce al calcolo degli insiemi dei caratteri iniziali *Ini*, finali *Fin* e dei digrammi *Dig* del linguaggio.

*Calcolo degli insiemi locali d'un linguaggio regolare*

Date le e.r.  $e, e'$ , le seguenti regole, da applicare ai caratteri terminali e agli operatori, calcolano i tre insiemi.

È necessario prima accertare se la e.r. definisce anche la stringa vuota, nel qual caso essa è detta *annullabile*. La funzione  $Null(e)$  assume il valore  $\varepsilon$ , se  $\varepsilon \in L(e)$ ; altrimenti assume il valore  $\emptyset$ . Essa è calcolata mediante le regole:

$$\begin{array}{ll} Null(\varepsilon) = \varepsilon & Null(\emptyset) = \emptyset \\ Null(a) = \emptyset, \text{ per ogni terminale } a & Null(e \cup e') = Null(e) \cup Null(e') \\ Null(e.e') = Null(e) \cap Null(e') & Null(e^*) = \varepsilon \\ Null(e^+) = Null(e) & \end{array}$$

Ad es. si ha:

$$Null((a \cup b)^* ba) = Null((a \cup b)^*) \cap Null(ba) = \varepsilon \cap (Null(b) \cap Null(a)) = \varepsilon \cap \emptyset = \emptyset$$

Le seguenti regole calcolano i tre insiemi locali:

| <i>Ini</i>                                | <i>Fin</i>                                |
|---|---|
| $Ini(\emptyset) = \emptyset$              | $Fin(\emptyset) = \emptyset$              |
| $Ini(\varepsilon) = \emptyset$            | $Fin(\varepsilon) = \emptyset$            |
| $Ini(a) = \{a\}$ , per ogni terminale $a$ | $Fin(a) = \{a\}$ , per ogni terminale $a$ |
| $Ini(e \cup e') = Ini(e) \cup Ini(e')$    | $Fin(e \cup e') = Fin(e) \cup Fin(e')$    |
| $Ini(e.e') = Ini(e) \cup Null(e)Ini(e')$  | $Fin(e.e') = Fin(e') \cup Fin(e)Null(e')$ |
| $Ini(e^*) = Ini(e^+) = Ini(e)$            | $Fin(e^*) = Fin(e^+) = Fin(e)$            |

Nota: le regole per il calcolo di *Ini* e di *Fin* sono duali, nel senso che coincidono se si sostituisce alla e.r. la sua riflessa.

| <i>Digrammi</i>                                      |
|--|
| $Dig(\emptyset) = \emptyset$                         |
| $Dig(\varepsilon) = \emptyset$                       |
| $Dig(a) = \emptyset$ , per ogni terminale $a$        |
| $Dig(e \cup e') = Dig(e) \cup Dig(e')$               |
| $Dig(e.e') = Dig(e) \cup Dig(e') \cup Fin(e)Ini(e')$ |
| $Dig(e^*) = Dig(e^+) = Dig(e) \cup Fin(e)Ini(e)$     |

Terminati questi calcoli, si può costruire il riconoscitore d'una e.r. lineare, come sotto illustrato.

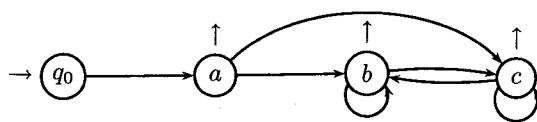
*Esempio 3.33.* Riconoscitore di e.r. lineare.

La e.r.  $a(b \cup c)^*$  non è annullabile, e si pone  $Null = \emptyset$ .

Calcolati gli insiemi locali

$$Ini = a \quad Fin = \{b, c\} \cup a = \{a, b, c\} \quad Dig = \{ab, ac\} \cup \{bb, bc, cb, cc\}$$

si costruisce, con l'algoritmo di p. 127, l'automa locale:



dove al solito le etichette degli archi sono il nome dello stato d'arrivo.

#### *Numerazione della espressione regolare*

Per consentire l'applicazione del procedimento a una una e.r. generica, la si trasforma in una e.r. lineare sostituendo i caratteri terminali con altri tutti distinti, ottenuti numerando progressivamente i caratteri della e.r..

Così l'espressione

$$(ab)^*a \text{ diventa } (a_1b_2)^*a_3$$

La seconda e.r. è lineare nell'alfabeto dei caratteri numerati, e quindi definisce un linguaggio locale.

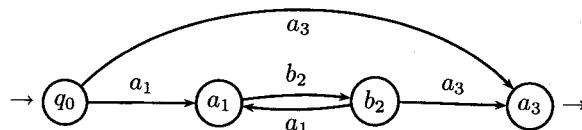
Costruito nel modo noto il riconoscitore della e.r. numerata, quello della e.r. originale si ottiene infine, cancellando i numeri dalle etichette degli archi.<sup>10</sup> Conviene riepilogare i passi dell'algoritmo.

#### *Algoritmo 3.34. GMY*

L'algoritmo si articola in quattro passi:

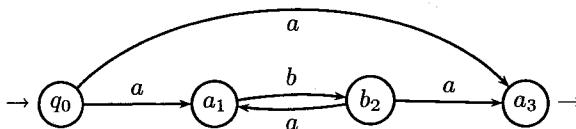
1. Numera l'e.r. e ottenendo una e.r. lineare  $e'$ ;
2. Calcola per  $e'$  l'annullabilità e gli insiemi locali *Ini*, *Fin*, *Dig*;
3. Costruisci il riconoscitore del linguaggio locale caratterizzato da tali insiemi (come spiegato a p. 127);
4. Cancella la numerazione dalle etichette degli archi.

*Esempio 3.35.* Per la e.r.  $(ab)^*a$  si calcola la e.r. numerata  $(a_1b_2)^*a_3$  e se ne costruisce il riconoscitore:



Cancellando i numeri dagli archi, si ottiene il riconoscitore del linguaggio desiderato:

<sup>10</sup>Questo è un esempio di traslitterazione (omomorfismo) definito a p. 80.



Si osservi che l'automa è indeterministico, privo di mosse spontanee, e ha tanti stati quanti i caratteri terminali presenti nella e.r., più uno.

### 3.8.3 Costruzione del riconoscitore deterministico di Berry e Sethi

Data una e.r., per ottenere l'automa deterministico che la riconosce, si potrebbe trasformare l'automa costruito da GMY, mediante l'algoritmo delle parti finite; ma conviene presentare un algoritmo diretto.<sup>11</sup>

Sia  $e$  la e.r. di alfabeto  $\Sigma$  e sia  $e'$  quella numerata, di alfabeto  $\Sigma_N$ , con gli insiemi locali  $Ini, Fin, Dig$  e la funzione  $Null$ .

Per convenienza, invece dei diagrammi, si usa l'insieme dei *séguiti* d'un carattere nella e.r. numerata, così definito:

$$Seg(a_i) = \{b_j \mid a_i b_j \in Dig(e')\}$$

Inoltre, per convenzione, lo speciale carattere  $\dashv$  (pensabile come terminatore della stringa) sta nei séguiti dei caratteri finali:

$$\dashv \in Seg(a_i), \text{ per ogni } a_i \in Fin(e')$$

(ciò equivale a considerare la e.r.  $e' \dashv$  invece di  $e'$ ). Il terminatore non ha seguito,  $Seg(\dashv) = \emptyset$ .

*Algoritmo 3.36. Algoritmo BS (Berry e Sethi)*

Ogni stato dell'automa è denotato da un sottoinsieme di  $\Sigma_N \cup \dashv$ .

L'algoritmo esamina ogni stato, per costruire gli archi uscenti e i loro stati di arrivo, applicando una regola simile a quella dell'algoritmo delle parti finite. Dopo l'esame, lo stato è marcato come visitato, per evitare di riesaminarlo.

Lo stato iniziale è  $Ini(e' \dashv)$ . Uno stato è finale se contiene  $\dashv$ .

Inizialmente l'insieme degli stati  $Q$  contiene il solo stato iniziale.

$$Q := \{Ini(e' \dashv)\}$$

while esiste in  $Q$  uno stato  $q$  non visitato do

    segna  $q$  come visitato

    per ogni carattere  $b \in \Sigma$

        do

$$q' := \bigcup_{\text{carattere numerato } b' \in q} Seg(b')$$

        se  $q'$  non è vuoto né appartiene a  $Q$ ,

            aggiungilo come nuovo stato non visitato, ponendo

<sup>11</sup>Per approfondimenti si veda [7].

```


$$Q := Q \cup \{q'\}$$

aggiungi l'arco  $q \xrightarrow{b} q'$ 
end do
end do

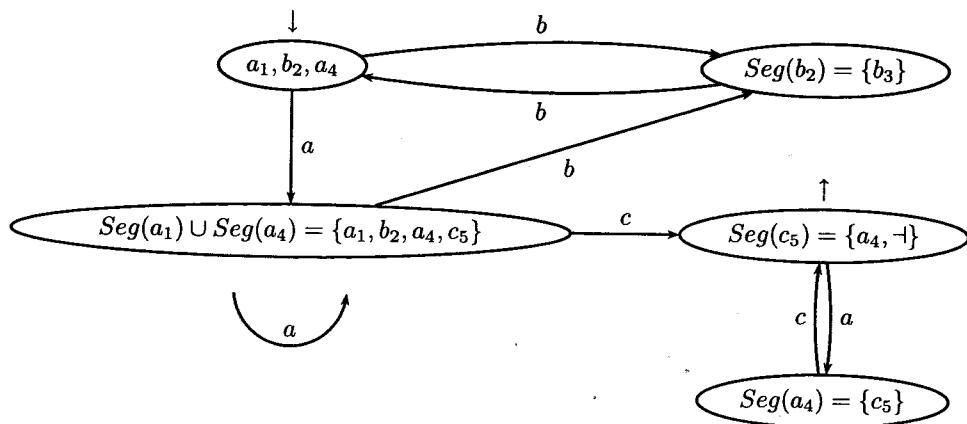
```

*Esempio 3.37.* Data l'e.r.

$(a \mid bb)^*(ac)^+$ , numerata  $(a_1 \mid b_2 b_3)^*(a_4 c_5)^+ \dashv$ , si calcola l'insieme  $Ini(e') = \{a_1, b_2, a_4\}$ , poi la funzione *seguiti*

|       | seguiti         |
|-------|-----------------|
| $a_1$ | $a_1, b_2, a_4$ |
| $b_2$ | $b_3$           |
| $b_3$ | $a_1, b_2, a_4$ |
| $a_4$ | $c_5$           |
| $c_5$ | $a_4, \dashv$   |

e infine l'automa deterministico:



Gli automi costruiti con questi procedimenti contengono spesso più stati del necessario, e, se necessario, possono essere minimizzati nella maniera nota.

#### *Applicazione alla determinizzazione d'un automa*

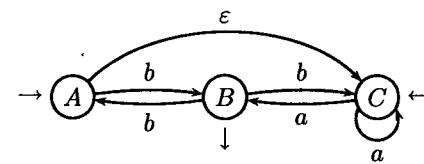
L'algoritmo BS può essere usato, in alternativa a quello delle parti finite, per costruire un automa deterministico  $M$  equivalente a un automa indeterministico  $N$ , che può contenere  $\epsilon$ -archi.

Si procede nel modo seguente.

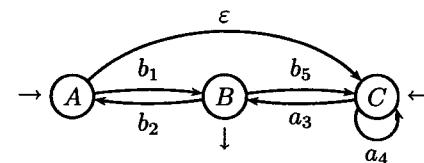
1. Si numerano in modo distinto le etichette degli archi non spontanei di  $N$ . L'automa numerato  $N'$  ottenuto riconosce un linguaggio locale.
2. Si calcolano gli insiemi locali  $Ini$ ,  $Fin$ ,  $Seguiti$  per l'automa  $N'$ . Il calcolo si svolge con regole analoghe a quelle usate per una e.r..
3. Si applica la costruzione di BS, per ottenere l'automa deterministico  $M$ .

È sufficiente un esempio per illustrare l'applicazione.

*Esempio 3.38.* Dato l'automa indeterministico  $N$



si numerano le etichette terminali, ottenendo l'automa  $N'$ :



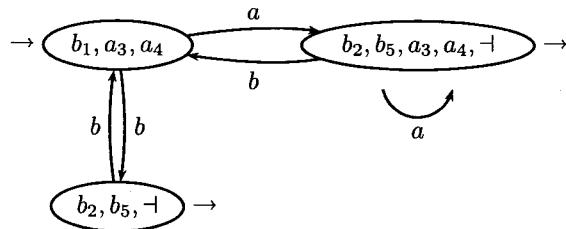
il cui linguaggio è locale. La stringa vuota non appartiene al linguaggio. Si calcola l'insieme degli inizi

$$Ini(L(N')) = \{b_1, a_3, a_4\}$$

notando che è  $\varepsilon a_4 = a_4$ ,  $\varepsilon a_3 = a_3$ ; poi l'insieme dei seguiti

|       | seguiti            |
|-------|--------------------|
| $b_1$ | $b_2, b_5, \dashv$ |
| $b_2$ | $b_1, a_3, a_4$    |
| $a_3$ | $b_2, b_5, \dashv$ |
| $a_4$ | $a_3, a_4$         |
| $b_5$ | $a_3, a_4$         |

Infine l'algoritmo BS costruisce l'automa deterministico  $M$ :



### 3.9 Espressioni regolari con complemento e intersezione

Lo studio delle operazioni sui linguaggi regolari è qui completato considerando il complemento, l'intersezione e la differenza insiemistica, che erano stati lasciati in sospeso. Infatti si incontrano situazioni in cui queste operazioni rendono più facile o più concisa l'espressione del linguaggio.

*Proprietà 3.39.* Chiusura di  $REG$  rispetto a complemento e intersezione.  
Siano  $L$  e  $L'$  linguaggi regolari. Allora anche il complemento  $\neg L$  e l'intersezione  $L \cap L'$  sono linguaggi regolari.

Si inizia dalla costruzione del riconoscitore del complemento  $\neg L = \Sigma^* \setminus L$ . Si può supporre che il riconoscitore  $M$  di  $L$  sia deterministico, con stato iniziale  $q_0$ , stati  $Q$ , stati finali  $F$ , e con funzione di transizione  $\delta$ .

*Algoritmo 3.40.* Costruzione del riconoscitore deterministico  $\overline{M}$  del complemento.

Per primo si completa l'automa  $M$ , aggiungendo lo stato d'errore o pozzo  $p$  e gli archi che in esso entrano:

1. Crea un nuovo stato  $p \notin Q$ , il *pozzo*; gli stati di  $\overline{M}$  sono  $Q \cup \{p\}$
2. la funzione di transizione  $\bar{\delta}$  è:
  - a)  $\bar{\delta}(q, a) = \delta(q, a)$ , dove  $\delta(q, a) \in Q$ ;
  - b)  $\bar{\delta}(q, a) = p$ , dove  $\delta(q, a)$  non è definita;
  - c)  $\bar{\delta}(p, a) = p$ , per ogni carattere  $a \in \Sigma$
3. gli stati finali sono  $\overline{F} = (Q \setminus F) \cup \{p\}$ .

Si noti che gli stati finali e non finali sono stati scambiati.

Per primo si osservi che, se un calcolo di  $M$  riconosce la stringa  $x \in L(M)$ , allora il corrispondente calcolo di  $\overline{M}$  termina in uno stato non finale, ossia  $x \notin L(\overline{M})$ .

Poi, se un calcolo di  $M$  non riconosce  $y$ , due sono i casi possibili: il calcolo termina in uno stato  $q$  non finale, o il calcolo termina nello stato pozzo  $p$ . In entrambi i casi il calcolo corrispondente di  $\overline{M}$  termina in uno stato finale, ossia  $y \in L(\overline{M})$ .

Da ultimo, per dimostrare che l'intersezione di due linguaggi regolari è regolare, basta invocare la nota identità di De Morgan:

$$L \cap L' = \neg(\neg L \cup \neg L')$$

poiché, sapendo che i linguaggi  $\neg L$  e  $\neg L'$  sono regolari, la loro unione è regolare.

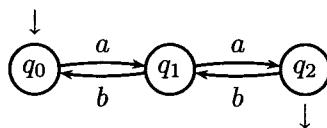
Come corollario, anche la *differenza insiemistica* di due linguaggi regolari è regolare grazie all'identità

$$L \setminus L' = L \cap \neg L'$$

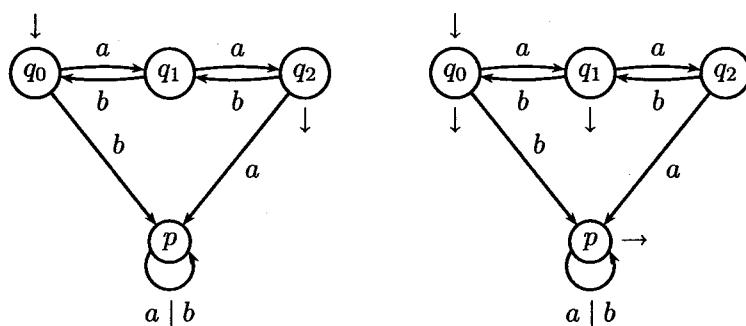
*Esempio 3.41.* Automa del complemento.

La seguente figura mostra l'automa dato  $M$ , quello intermedio completato con il pozzo, e il riconoscitore  $\overline{M}$  del complemento.

automa originale  $M$

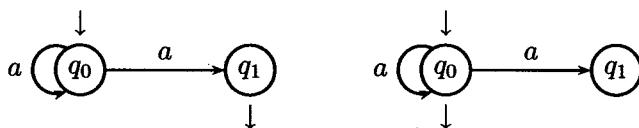


dopo il completamento con il pozzo automa  $\overline{M}$  del complemento



Per la validità del risultato, è essenziale che l'automa da trasformare sia deterministico, altrimenti il linguaggio accettato dall'automa costruito potrebbe non essere disgiunto da quello originale, violando la proprietà del complemento,  $L \cap \neg L = \emptyset$ . Come controesempio, si veda

*automa originale      pseudo-automa del complemento*



dove l'automa dello pseudo-complemento accetta erroneamente la stringa  $a$ , appartenente al linguaggio originale.

Per ultimo osserviamo che la costruzione può produrre automi che hanno stati inutili, o comunque automi non minimi.

### 3.9.1 Prodotto di automi

Una manipolazione frequente nelle applicazioni della teoria degli automi è il *prodotto (cartesiano)*, che consiste nella simulazione di due automi mediante un unico automa, avente come insieme di stati il prodotto cartesiano degli stati delle due macchine. Se ne vedrà ora l'impiego per costruire il riconoscitore del linguaggio intersezione di due linguaggi regolari.

Per inciso, la dimostrazione della chiusura della famiglia *REG* rispetto all'intersezione, basata sull'identità di De Morgan (p. 137), già fornisce un procedimento per riconoscere l'intersezione: dati i riconoscitori finiti deterministici di

due linguaggi, si costruiscono quelli dei loro complementi, e poi il riconoscitore dell'unione, (applicando il metodo di Thompson di p. 123). Da quest'ultimo (dopo averlo reso deterministico) si costruisce infine l'automa del complemento.

In modo più diretto, l'intersezione di due linguaggi regolari può essere riconosciuta dalla macchina prodotto cartesiano dei due automi dati  $M'$  e  $M''$ . Per semplicità si suppone che essi siano privi di mosse spontanee, ma non necessariamente deterministici.

La macchina prodotto  $M$  ha come insieme degli stati il prodotto cartesiano degli stati dei due automi  $Q' \times Q''$ . Pertanto ogni stato è una coppia  $\langle q', q'' \rangle$ , dove la prima (seconda) componente è uno stato della prima (seconda) macchina.

In tale stato si definisce l'arco uscente

$$\langle q', q'' \rangle \xrightarrow{a} \langle r', r'' \rangle$$

se, e solo se, esistono gli archi  $q' \xrightarrow{a} r'$  in  $M'$  e  $q'' \xrightarrow{a} r''$  in  $M''$ .

Ossia tale arco sta in  $M$  se, e solo se, la sua proiezione sulla prima (risp. seconda) componente sta in  $M'$  (risp. in  $M''$ ).

Gli stati iniziali  $I$  di  $M$  sono il prodotto  $I = I' \times I''$  degli stati iniziali delle macchine componenti; e quelli finali sono il prodotto degli stati finali,  $F = F' \times F''$ . Per giustificare la validità della costruzione, si consideri una stringa  $x$  appartenente all'intersezione: essa è accettata da un calcolo di  $M'$  e da un calcolo di  $M''$ , dunque anche da un calcolo di  $M$  che passa attraverso le coppie di stati, rispettivamente visitati dai due calcoli.

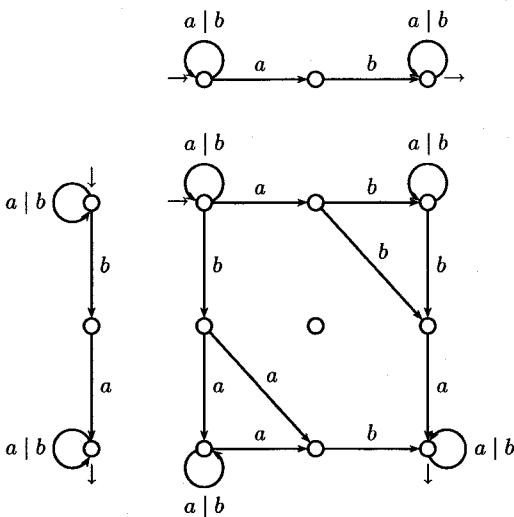
Viceversa se  $x$  non sta nell'intersezione, almeno uno dei calcoli di  $M'$  e  $M''$  non raggiunge uno stato finale, dunque neanche il calcolo di  $M$  raggiunge uno stato finale.

*Esempio 3.42.* Intersezione e macchina prodotto (Sakarovitch).

Il riconoscitore delle stringhe che contengono almeno una sottostringa  $ab$  e una sottostringa  $ba$  è specificato molto naturalmente mediante l'intersezione dei linguaggi  $L', L''$  seguenti

$$L' = (a \mid b)^* ab(a \mid b)^* \quad L'' = (a \mid b)^* ba(a \mid b)^*$$

e quindi il prodotto cartesiano dei riconoscitori dei due linguaggi:



Le coppie del prodotto cartesiano non raggiungibili dallo stato iniziale possono al solito essere omesse.

Il metodo del prodotto cartesiano può essere applicato anche a altre operazioni diverse dall'intersezione; così, nel caso dell'unione, sarebbe facile modificare la macchina prodotto in modo che essa riconosca una stringa, se almeno una delle macchine componenti la riconosce. Tuttavia la macchina prodotto ha un numero di stati maggiore dell'automa, costruito con il metodo strutturale di Thompson: il prodotto delle cardinalità invece della somma.

### Espressioni regolari estese e ristrette

Una *espressione regolare* è detta *estesa* se contiene, in aggiunta alle operazioni di base (unione, concatenamento e stella), anche il complemento, l'intersezione o la differenza insiemistica.

Ad es. le stringhe, che contengono almeno una sottostringa *aa* e non terminano con *bb*, sono definite dalla e.r. estesa

$$((a \mid b)^* aa(a \mid b)^*) \cap \neg((a \mid b)^* bb)$$

Segue un esempio più applicativo, per dimostrare l'utilità delle e.r. estese.

#### *Esempio 3.43.* Identifieri.

Gli identifieri validi possono contenere le lettere *a...z*, le cifre *0...9* (non in prima posizione) e il trattino *'-'* (non in prima né in ultima posizione). Non sono permessi due trattini consecutivi.

Una frase è: *dopo - 2ndo - test*.

La seguente e.r. estesa prescrive che le stringhe (i) inizino con una lettera, (ii) non contengano due tratti consecutivi e (iii) non terminino con un tratto:

$$\begin{aligned}
 & \underbrace{(a \dots z)^+ (a \dots z \mid 0 \dots 9 \mid -)^*}_{(i)} \cap \\
 & \quad \neg ((a \dots z \mid 0 \dots 9 \mid -)^* -- (a \dots z \mid 0 \dots 9 \mid -)^*) \cap \\
 & \quad \underbrace{\neg ((a \dots z \mid 0 \dots 9 \mid -)^* -)}_{(ii)} \\
 & \quad \quad \quad \underbrace{\neg ((a \dots z \mid 0 \dots 9 \mid -)^* -)}_{(iii)}
 \end{aligned}$$

### Linguaggi senza stella

L'uso degli operatori di complemento e di intersezione non aggiunge nulla alla famiglia dei linguaggi regolari, perché questi operatori possono essere eliminati riducendoli a quelli di base (unione, concatenamento, stella).

Invece, l'eliminazione della stella dagli operatori permessi causa una perdita nella famiglia dei linguaggi definibili, che costituiscono una sottoclassificazione di *REG*, detta la famiglia dei linguaggi *aperiodici* o *senza stella* (star free) o *senza contatore* (non-counting).

Poiché tale sottoclassificazione è raramente considerata nell'ambito della compilazione, basti qui un cenno.

Si considerino gli operatori d'unione, concatenamento, intersezione e complemento. Partendo dagli elementi terminali dell'alfabeto e dall'insieme vuoto, si può allora definire una famiglia di linguaggi mediante delle *espressioni regolari senza stella*, facenti uso esclusivo di detti operatori. Tali e.r. differiscono da quelle classiche perché possono contenere l'intersezione e il complemento, ma non la stella (né la croce).

Un linguaggio è detto *senza stella* se esiste una e.r. senza stella che lo definisce.

Si nota subito che il linguaggio universale di alfabeto  $\Sigma$ , essendo il complemento dell'insieme vuoto,  $\Sigma^* = \neg\emptyset$ , è un linguaggio senza stella.

In effetti, si è già incontrata, senza saperlo, una famiglia di linguaggi senza stella: i linguaggi locali (p. 126) sono infatti sempre definibili senza fare uso della stella. Si ricorda che un linguaggio locale è definito da tre insiemi caratteristici: gli inizi, le fini e i digrammi permessi (o vietati). Tale proprietà è facilmente espressa dalla intersezione di tre linguaggi senza stella.

*Esempio 3.44.* E.r. senza stella di un linguaggio locale.

Le frasi del linguaggio locale  $(abc)^+$  dell'esempio di p. 126, iniziano con  $a$ , finiscono con  $c$  e non contengono i digrammi  $\{aa \mid ac \mid ba \mid bb \mid cb \mid cc\}$ . Il linguaggio è allora definito dalla e.r. senza stella:

$$(a \neg\emptyset) \cap (\neg\emptyset c) \cap (\neg(\neg\emptyset (aa \mid ac \mid ba \mid bb \mid cb \mid cc) \neg\emptyset))$$

La famiglia dei linguaggi senza stella è strettamente inclusa in quella dei linguaggi regolari. Non ne fanno parte i linguaggi caratterizzati da una proprietà

di conteggio che giustifica l'altro nome “senza contatore” della famiglia. Fuoriesce dalla famiglia senza stella il linguaggio regolare

$$\{x \in \{a \mid b\}^* \mid |x|_a \text{ è pari}\}$$

facilmente riconosciuto da un automa finito, avente un ciclo che conta modulo 2 le  $a$  incontrate. Per questo linguaggio non esiste una e.r. che non faccia uso della stella o della croce.

Nella casistica dei linguaggi inventati dall'uomo, l'operazione di conteggiare (nel senso intuitivamente spiegato) certe lettere o sottostringhe non sembra essere mai richiesta, al fine di controllare la correttezza. Per ragioni, che forse hanno a che fare con l'organizzazione della mente umana o forse con la robustezza della comunicazione, nessun linguaggio tecnico discrimina le frasi valide da quelle scorrette, sulla base d'una congruenza aritmetica ossia d'un conteggio ciclico. Si immagini la stranezza d'un linguaggio di programmazione, in cui la validità d'un blocco di istruzioni dipendesse dalla classe di congruenza del numero delle istruzioni presenti: blocco valido se il numero è multiplo pongasi di tre, scorretto altrimenti.

Nello studio dei linguaggi regolari utilizzati nella comunicazione uomo-macchina, sarebbe dunque sufficiente restringere l'attenzione ai linguaggi senza stella o contatore ciclico.

Ma ciò non vale in altri ambiti dell'informatica, anzi è ovvio che molti circuiti logici dei microprocessori svolgono operazioni di conteggio; basti pensare che uno dei più semplici componenti, il circuito bistabile o flip-flop, conta modulo due gli impulsi, ossia riconosce linguaggio  $(11)^*$ , che non è senza stella.<sup>12</sup>

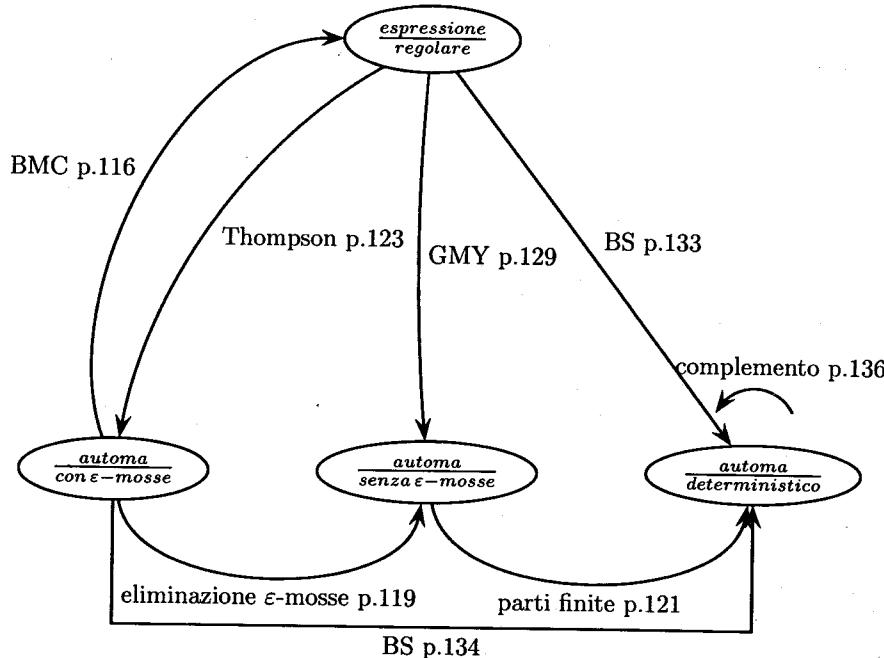
### 3.10 Riepilogo: relazioni tra linguaggi regolari, automi e grammatiche

A conclusione dello studio dei linguaggi regolari e degli automi finiti, è utile ricapitolare le relazioni e le trasformazioni tra i modelli formali di questa famiglia di linguaggi.

La prossima figura delinea, sotto forma di grafo di flusso, i procedimenti che trasformano espressioni regolari in automi e viceversa, nonché le conversioni tra vari tipi di automi:

---

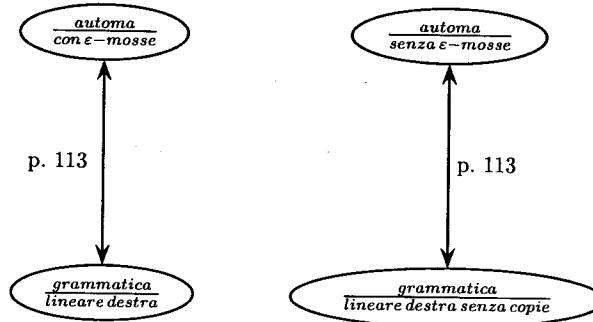
<sup>12</sup>Per la teoria dei linguaggi senza stella si veda McNaughton e Papert [34].



Ad es. si legge nella figura che l'algoritmo GMY converte una e.r. in un automa privo di mosse spontanee.

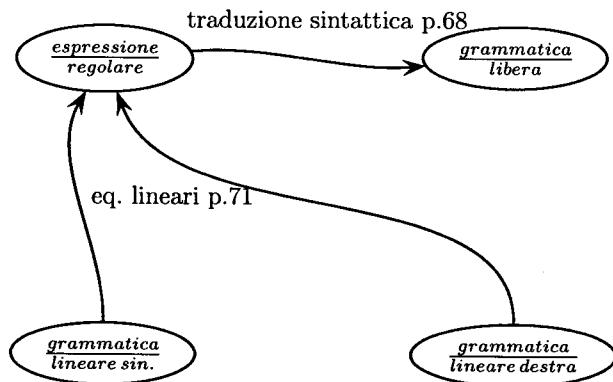
Dove non è indicato il metodo di conversione, la corrispondenza tra i due modelli è immediata.

La prossima figura mostra le corrispondenze dirette tra grammatiche lineari a destra e automi finiti:



Non sono riportate le relazioni tra gli automi e le grammatiche lineari a sinistra, perché sono identiche a quelle con le grammatiche lineari a destra, grazie al principio di dualità sinistra-destra.

Infine si ricordano le relazioni tra espressioni regolari e grammatiche:



Il fatto fondamentale che emerge dal quadro è che le famiglie dei linguaggi regolari, dei linguaggi accettati da automi finiti (deterministici e non) e dei linguaggi generati da grammatiche unilineari (o del tipo 3) coincidono.  
Infine si ricorda che i linguaggi regolari sono una sottofamiglia assai ristretta di quelli liberi.

## Riconoscimento e parsificazione delle frasi

### 4.1 Introduzione

I linguaggi liberi, per essere riconosciuti, richiedono risorse di memoria maggiori di quelli regolari. Nel capitolo si studiano i relativi algoritmi, visti prima come automi astratti con una pila di memoria, poi come procedure di analisi sintattica o parsificazione<sup>1</sup>, che costruiscono l'albero sintattico delle frasi. Per inciso, la parsificazione ha poco interesse per i linguaggi regolari poiché la loro struttura sintattica può essere predeterminata (lineare a destra o a sinistra), mentre per i linguaggi liberi essa prende forme diverse.

Come per le grammatiche unilineari, così per quelle libere vi è corrispondenza tra le regole grammaticali e le mosse dell'automa, che però ora possiede, oltre agli stati, una memoria a pila. Una differenza vistosa tra le due famiglie è che per i linguaggi liberi non sempre si può ottenere un automa deterministico. Inoltre la presenza di due memorie, la pila e gli stati, introduce una varietà di modi di funzionamento che complica il quadro teorico, rispetto al caso degli automi finiti.

Dopo la presentazione generale degli automi a pila, si studieranno quelli deterministici che definiscono la famiglia molto utile *DET* dei linguaggi liberi deterministici, di cui si vedranno le principali proprietà.

Il resto del capitolo presenta gli algoritmi di parsificazione: prima gli algoritmi veloci, operanti in tempo lineare, poi un algoritmo parsificatore di tipo generale.

I parsificatori in tempo lineare studiati sono di due tipi: discendenti ( $LL(k)$ ) e ascendenti ( $LR(k)$ ), a seconda dell'ordine in cui l'albero sintattico è costruito. Entrambi i tipi sono diffusamente impiegati nella compilazione. Tali algoritmi presuppongono che il linguaggio sia deterministico, e impongono ulteriori limitazioni sulla forma delle regole della grammatica.

La successiva presentazione dei metodi di costruzione dei parsificatori procede attraverso un percorso espositivo omogeneo, che poggia sulla rappresentazio-

---

<sup>1</sup>Dal latino *pars*, *partis*, nel senso di dividere la frase nelle sue parti.

ne della grammatica come rete ricorsiva di automi. Tale unificazione mira a evidenziare le simiglianze tra gli algoritmi e le rispettive condizioni di applicabilità, evitando fastidiose ripetizioni di concetti molto simili.

Per i parsificatori discendenti è poi esposta la realizzazione mediante procedure ricorsive, che ha il pregio di permettere una facile costruzione manuale. Per completezza il capitolo tratta anche la costruzione dei parsificatori deterministicici per le grammatiche estese con espressioni regolari.

L'ultimo metodo presentato è quello di Earley, adatto a parsificare qualsiasi grammatica, ma di complessità maggiore pur se ancora polinomiale. Il quadro complessivo copre dunque tutta la gamma degli algoritmi normalmente usati nella compilazione e nel trattamento dei documenti.

Il capitolo termina con una discussione dei criteri di scelta dei parsificatori.

#### 4.1.1 Automi a pila

Gli algoritmi di riconoscimento usati dai compilatori sono essenzialmente degli automi finiti dotati d'una memoria ausiliaria organizzata come una pila illimitata di simboli

$$A_1 A_2 \dots \overset{\text{cima}}{\overbrace{A_k}}$$

La stringa di ingresso o sorgente, delimitata a destra dal terminatore, è scritta

$$a_1 a_2 \dots \overset{\text{carattere corrente}}{\overbrace{a_i}} \dots a_n \dashv$$

Le operazioni applicabili alla pila sono:

*impilamento*:  $\text{push}(B)$  pone il simbolo  $B$  in cima, cioè alla destra di  $A_k$ ; più operazioni di impilamento possono essere combinate e rappresentate da  $\text{push}(B_1, B_2, \dots, B_n)$

*test di pila vuota*:  $\text{empty}$ , predicato vero solo se  $k = 0$ ;

*disimpilamento*:  $\text{pop}$ , purché la pila non sia vuota, toglie da essa il simbolo  $A_k$ .

Conviene immaginare che la pila abbia dipinto sul fondo un simbolo speciale, spesso scritto  $Z_0$ , detto il *fondo*. Esso potrà essere soltanto letto (né tolto né impilato). La sua presenza in cima alla pila significa che essa è vuota.

Come in un automa finito, i caratteri della stringa sono esaminati da sinistra a destra, dalla testina di lettura. Il carattere, che sta sotto la testina in un certo momento, è detto *corrente*. In un dato istante la *configurazione* dell'automa è caratterizzata da: il carattere corrente, lo stato corrente, il contenuto della pila. Con una mossa l'automa:

- legge il carattere corrente avanzando con la testina, oppure compie una mossa spontanea senza muovere la testina;
- legge il simbolo in cima e lo toglie dalla pila, oppure legge  $Z_0$  se la pila è vuota;

- in funzione dei valori dello stato corrente, del carattere corrente e del simbolo letto dalla pila, calcola il nuovo stato e eventualmente impila uno o più simboli.

#### 4.1.2 Dalla grammatica all'automa a pila

Si mostra che le regole grammaticali possono essere viste come istruzioni d'una macchina a pila indeterministica che riconosce il linguaggio; essa è tanto semplice da non usare gli stati ma soltanto la pila come memoria.

Intuitivamente, l'automa opera in modo *predittivo* o finalizzato (goal oriented), usando la pila come una agenda delle future azioni da compiere. I simboli della pila sono i caratteri nonterminali e terminali della grammatica. Se la pila contiene, dalla cima al fondo, i simboli  $A_k \dots A_1$ , la macchina eseguirà per prima l'azione associata a  $A_k$ , poi la  $A_{k-1}$ , e così via fino all'ultima azione  $A_1$ . L'azione associata a  $A_k$  ha l'obiettivo di riconoscere se in ingresso, a partire dal carattere corrente  $a_i$ , vi è una stringa  $w$  derivabile da  $A_k$ . In caso positivo, l'azione farà avanzare la testina di lettura di  $|w|$  posizioni. Naturalmente l'obiettivo può articolarsi ricorsivamente in sotto obiettivi, se per riconoscere  $A_k$  è necessario riconoscere altri simboli terminali o non. Inizialmente l'obiettivo è l'assioma della grammatica: compito del riconoscitore è infatti riconoscere se la stringa sorgente deriva dall'assioma.

*Algoritmo 4.1.* Dalla grammatica all'automa a pila indeterministico con un solo stato.

Data la grammatica libera  $G = (V, \Sigma, P, S)$ , si precisa ora la corrispondenza tra le regole e le mosse dell'automa. Nella tabella la lettera  $b$  sta per un terminale, le lettere  $A, B$  per un nonterminale e  $A_i$  per un simbolo qualsiasi.

| <i>regola</i>                                  | <i>mossa</i>   | <i>commento</i>  |
|--|--|--|
| 1 $A \rightarrow BA_1 \dots A_n$<br>$n \geq 0$ | if <i>cima</i> = <i>A</i> then pop;<br>push( <i>A<sub>n</sub> ... A<sub>1</sub>B</i> ) end if  | Per riconoscere <i>A</i> si devono riconoscere <i>BA<sub>1</sub> ... A<sub>n</sub></i>                                 |
| 2 $A \rightarrow bA_1 \dots A_n$<br>$n \geq 0$ | if <i>car_corr</i> = <i>b</i> $\wedge$ <i>cima</i> = <i>A</i> then pop;<br>push ( <i>A<sub>n</sub> ... A<sub>1</sub></i> ); avanza testina lettura | <i>b</i> era il primo carattere atteso ed è stato letto; restano da riconoscere <i>A<sub>1</sub> ... A<sub>n</sub></i> |
| 3 $A \rightarrow \epsilon$                     | if <i>cima</i> = <i>A</i> then pop   | È stata riconosciuta la stringa vuota che deriva da <i>A</i>   |
| 4 per ogni carattere <i>b</i> $\in \Sigma$     | if <i>car_corr</i> = <i>b</i> $\wedge$ <i>cima</i> = <i>b</i> then pop; avanza testina lettura   | <i>b</i> era il primo carattere atteso ed è stato letto  |
| 5 — — —  | if <i>car_corr</i> = $\perp$ $\wedge$ pila è vuota then accetta; alt.  | La stringa è stata scandita per intero e non restano in agenda altri obiettivi   |

La forma della regola determina la mossa. Se la parte destra inizia con un terminale (2), la mossa è condizionata dalla lettura dello stesso. Invece altre regole (1 e 3) danno luogo a mosse spontanee, che non leggono il carattere corrente. Poi vi sono le mosse (4) che controllano che un terminale, quando affiora in cima alla pila (essendo stato precedentemente impilato da una mossa del tipo 1 o 2) coincida con il carattere corrente. Infine la mossa (5) accetta la stringa se la pila è vuota alla lettura del terminatore.

Inizialmente la pila contiene soltanto il simbolo di fondo pila  $Z_0$  e 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.

L'automa riconosce la stringa se, esiste un calcolo che termina con la mossa 5.

Si dice che questo modello d'automa riconosce le frasi a *pila vuota*.

Si noti che l'automa non ha bisogno di stati diversi, cioè la pila è l'unica memoria. Più avanti però si aggiungeranno gli stati, al fine di rendere deterministico l'automa.

*Esempio 4.2.* Le mosse del riconoscitore predittivo del linguaggio

$$L = \{a^n b^m \mid n \geq m \geq 1\}$$

sono riportate accanto alle regole della grammatica:

| <i>Regola</i>         | <i>Mossa</i>  |
|-----------------------|---|
| 1 $S \rightarrow aS$  | if $car\_corr = a \wedge cima = S$ then pop; push( $S$ ); avanza  |
| 2 $S \rightarrow A$   | if $cima = S$ then pop; push( $A$ )                               |
| 3 $A \rightarrow aAb$ | if $car\_corr = a \wedge cima = A$ then pop; push( $bA$ ); avanza |
| 4 $A \rightarrow ab$  | if $car\_corr = a$ and $cima = A$ then pop; push( $b$ ); avanza   |
| 5                     | if $car\_corr = b \wedge cima = b$ then pop; avanza               |
| 6                     | if $car\_corr = \dashv \wedge$ pila è vuota then accetta; alt.    |

La scelta tra le mosse 1 e 2 non è deterministica, poiché la 2 può essere scelta anche quando  $a$  è il carattere corrente; e similmente la scelta tra 3 e 4.

Non è difficile convincersi che l'automa così costruito riconosce una stringa se, e solo se, la grammatica la genera. Infatti per ogni calcolo che termina con successo esiste una derivazione corrispondente, e viceversa; in altre parole l'automa simula le derivazioni sinistre della grammatica.

Ad es. alla derivazione:

$$S \Rightarrow A \Rightarrow aAb \Rightarrow aabb$$

corrisponde la seguente traccia del calcolo, terminante con esito positivo:

$$\begin{array}{c|c} \text{Pila} & x \\ \hline Z_0S & aabb \dashv \end{array}$$

$$\underline{Z_0A} | aabb \dashv$$

$$\underline{Z_0bA} | abb \dashv$$

$$\underline{Z_0bb} | bb \dashv$$

$$\underline{Z_0b} | b \dashv$$

$$\underline{Z_0} | \dashv$$

Ma l'algoritmo non può sapere a priori quale sarà la derivazione giusta, e deve esplorare tutte le possibilità, anche quelle che falliranno, come la seguente:

$$\begin{array}{c|c} \text{Pila} & x \\ \hline Z_0S & aabb \dashv \end{array}$$

$$\underline{Z_0S} | abb \dashv$$

$$\underline{Z_0S} | bb \dashv$$

$$\underline{Z_0A} | bb \dashv$$

errore

Si noti anche che una stringa è accettata per mezzo di diverse computazioni se, e solo se, essa possiede diverse derivazioni sinistre, ossia se è ambigua per la grammatica.

Le regole della tabella di p. 148 sono a ben vedere bidirezionali, e possono essere applicate in senso inverso, per passare da un automa a pila del tipo considerato, alla grammatica che genera lo stesso linguaggio.

Mettendo insieme la trasformazione diretta e inversa, un risultato teorico importante è stato acquisito.

*Proprietà 4.3.* La famiglia dei linguaggi liberi coincide con quella dei linguaggi riconosciuti a pila vuota da un automa a pila indeterministico, avente un solo stato.

Si sottolinea che la costruzione precedente non vale per gli automi a pila aventi più stati, per i quali si dovranno utilizzare metodi un po' più articolati.

L'obbiettivo di questo capitolo potrebbe sembrare raggiunto poiché si dispone ora d'un procedimento per costruire il riconoscitore del linguaggio definito da una grammatica. Purtroppo l'automa non è deterministico e esplora tutte le vie, con una complessità di calcolo non polinomiale rispetto alla lunghezza della stringa sorgente. Seguiranno quindi altri algoritmi più efficienti.

### Complessità di calcolo dell'automa a pila

Si calcola ora un limite superiore sul numero di passi necessari nel caso peggiore per riconoscere una stringa, con l'automa a pila sopra descritto. Per semplicità si suppone che la grammatica  $G$  del linguaggio sia nella forma di Greibach (p. 66) in cui, come si ricorda, ogni regola inizia con un terminale e non contiene altri terminali. Pertanto non vi sono mosse spontanee (tipi (1) e (3) della tabella di p. 148) e l'automa non impila mai un carattere terminale.

Data una stringa  $x$  di lunghezza  $n$ , la derivazione  $S \xrightarrow{*} x$ , se esiste, ha esattamente  $n$  passi, e altrettante mosse compie l'automa per riconoscere  $x$ . Sia  $K$  il massimo tra i numeri delle alternative  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$  dei non-terminali della grammatica. A ogni passo della derivazione, per espandere il nonterminale può essere scelta una tra  $k \leq K$  alternative, quindi il numero delle possibili derivazioni di lunghezza  $n$  è al più  $K^n$ . Poiché nel caso peggiore, l'algoritmo può essere obbligato a esaminare tutte queste possibilità, prima di trovare quella giusta, la complessità di calcolo è esponenziale.

Tuttavia questo risultato è migliorabile; alla fine del capitolo un ragionamento più fine condurrà a un algoritmo di riconoscimento di complessità polinomiale, valido per ogni grammatica libera.

#### 4.1.3 Definizione dell'automa a pila

Nell'esporre i vari modelli della macchina a pila, si cercherà di ridurre i dettagli, contando sulla capacità del lettore, di adattare a questi automi i concetti

già studiati per quelli finiti.

Per definire un automa  $M$  a pila occorrono sette entità:

1.  $Q$ , l'insieme finito degli stati dell'unità di controllo;
2.  $\Sigma$ , l'alfabeto d'ingresso;
3.  $\Gamma$ , l'alfabeto della pila;
4.  $\delta$ , la funzione di transizione;
5.  $q_0 \in Q$ , lo stato iniziale;
6.  $Z_0 \in \Gamma$ , il simbolo iniziale della pila;
7.  $F \subseteq Q$ , l'insieme degli stati finali.

L'automa così definito è in generale indeterministico. Il dominio e codominio della funzione di transizione sono espressi da prodotti cartesiani tra insiemi:

| <i>dominio</i>                                       | <i>codominio</i>                                 |
|--|--|
| $Q \times (\Sigma \cup (\varepsilon)) \times \Gamma$ | le parti finite dell'insieme $Q \times \Gamma^*$ |

Le mosse, cioè i valori di  $\delta$ , possono essere così classificate e descritte:

- mossa con lettura:

$$\delta(q, a, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_n, \gamma_n)\}$$

con  $n \geq 1$ ,  $a \in \Sigma$ ,  $Z \in \Gamma$  e con  $p_i \in Q$ ,  $\gamma_i \in \Gamma^*$

L'automa nello stato  $q$  con  $Z$  in cima alla pila, leggendo  $a$ , può entrare in uno degli stati  $p_i$ ,  $1 \leq i \leq n$ , dopo aver eseguito in successione le operazioni pop, push( $\gamma_i$ ).

Note: la scelta dell'azione  $i$ -esima tra le  $n$  possibili non è deterministica; l'avanzamento della testina di ingresso è automatico; il simbolo in cima è sempre disimpilato; la stringa impilata può essere vuota.

- mossa spontanea:

$$\delta(q, \varepsilon, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_n, \gamma_n)\}$$

con le stesse indicazioni di prima. L'automa, nello stato  $q$  con  $Z$  in cima alla pila, senza leggere alcun carattere di ingresso, entra in uno degli stati  $p_i$ ,  $1 \leq i \leq n$ , dopo aver eseguito in successione le operazioni pop, push( $\gamma_i$ ).

Nella definizione è evidente che il comportamento può essere indeterministico, sia perché il codominio è un insieme di possibili scelte, sia per la presenza di mosse spontanee.

Si chiama *configurazione istantanea* della macchina  $M$  una terna  $(q, y, \eta) \in Q \times \Sigma^* \times \Gamma^+$ , che descrive:

- $q$ , lo stato attuale del controllo;
- $y$ , la parte (suffisso) ancora da leggere della stringa  $x$ ;
- $\eta$ , il contenuto della pila.

La configurazione *iniziale* è  $(q_0, x, Z_0)$  o  $(q_0, x \dashv, Z_0)$ , se si usa il terminatore. Il passaggio o transizione da una configurazione a un'altra, si indica con  $(q, y, \eta) \mapsto (p, z, \lambda)$ . Un calcolo è una catena di zero più transizioni, indicato con  $\mapsto^*$ . Al solito, si scrive la croce al posto della stella per denotare un calcolo di almeno una transizione.

Le mosse, con lettura e senza, danno luogo alle seguenti transizioni:

| <i>Conf. presente</i> | <i>Conf. successiva</i> | <i>Mossa applicata</i>   |
|-----------------------|-------------------------|--|
| $(q, az, \eta Z)$     | $(p, z, \eta\gamma)$    | mossa con lettura:<br>$\delta(q, a, Z) = \{(p, \gamma), \dots\}$         |
| $(q, az, \eta Z)$     | $(p, az, \eta\gamma)$   | mossa spontanea:<br>$\delta(q, \varepsilon, Z) = \{(p, \gamma), \dots\}$ |

Benché ogni mossa cancelli il simbolo in cima, esso può essere impilato di nuovo dalla stessa mossa, se non deve essere rimosso dalla pila.

Una stringa  $x$  è *riconosciuta* (o accettata) dalla macchina  $M$  mediante *stato finale* se esiste un calcolo che legge per intero la stringa e termina in uno stato finale:

$$(q_0, x, Z_0) \xrightarrow{*} (q, \varepsilon, \lambda), \quad q \text{ è uno stato finale}, \lambda \in \Gamma^*$$

Al solito il linguaggio riconosciuto è l'insieme delle stringhe accettate.

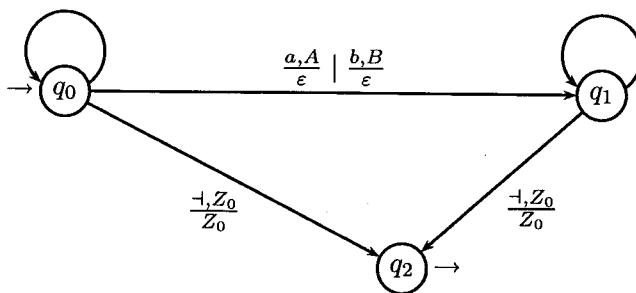
Si noti che, all'atto del riconoscimento, la pila contiene una stringa  $\lambda$ , sulla quale non è posta alcuna condizione, e che talora potrà essere la stringa vuota.

### Diagramma stato-transizione per automi a pila

Si possono rappresentare graficamente i diagrammi stato-transizione, benché l'effetto sia di lettura meno immediata rispetto agli automi finiti, dovendosi appesantire la rappresentazione con le operazioni sulla pila, come mostra il prossimo esempio.

*Esempio 4.4.* Il linguaggio  $L = \{uu^R \mid u \in \{a,b\}^*\}$  dei palindromi (p. 31) di lunghezza pari è accettato mediante stato finale dal riconoscitore a pila:

$$\frac{a,A}{AA} \mid \frac{a,B}{BA} \mid \frac{a,Z_0}{Z_0A} \mid \frac{b,A}{AB} \mid \frac{b,B}{BB} \mid \frac{b,Z_0}{Z_0B} \qquad \frac{a,A}{\varepsilon} \mid \frac{b,B}{\varepsilon}$$



L'alfabeto della pila ha tre simboli:  $Z_0$ , il simbolo iniziale,  $A$  e  $B$  che memorizzano rispettivamente l'avvenuta lettura d'una  $a$  e d'una  $b$ . Ad es. l'arco  $q_0 \xrightarrow{\frac{a,B}{BA}} q_0$  denota la mossa con lettura  $(q_0, BA) \in \delta(q_0, a, B)$ .

Il funzionamento indeterministico si manifesta in  $q_0$ , dove, leggendo  $a$  dall'ingresso e  $A$  dalla pila, la macchina può restare in  $q_0$  impilando  $AA$ , oppure andare in  $q_1$  senza impilare.

Data la stringa  $x = aa$  si hanno, tra altri possibili, i seguenti due calcoli.

| Pila    | $x$         | Stato | Commento  |
|---------|-------------|-------|---|
| $Z_0$   | $aa \dashv$ | $q_0$ |   |
| $Z_0A$  | $a \dashv$  | $q_0$ |   |
| $Z_0AA$ | $\dashv$    | $q_0$ | rifiuto: nessuna mossa<br>è definita per $(q_0, \dashv, A)$ |

| Pila   | $x$         | Stato | Commento                             |
|--------|-------------|-------|--------------------------------------|
| $Z_0$  | $aa \dashv$ | $q_0$ |                                      |
| $Z_0A$ | $a \dashv$  | $q_0$ |                                      |
| $Z_0$  | $\dashv$    | $q_1$ |                                      |
| $Z_0$  | $\dashv$    | $q_2$ | riconoscimento nello<br>stato finale |

La stringa  $aa$  è accettata perché esiste un calcolo che la scandisce per intero e giunge allo stato finale. Similmente la stringa vuota è riconosciuta con la mossa da  $q_0$  a  $q_2$ .

### Varietà di automi a pila

L'automa a pila sopra definito si differenzia da quello precedentemente (p. 148) costruito in modo diretto dalla grammatica, in due aspetti: possiede gli stati interni e usa una diversa condizione per accettare le stringhe.

#### Modalità di accettazione

Già si sono incontrate due varianti sul riconoscimento al termine del calcolo: la macchina entra in uno stato finale, o la pila è vuota. Il primo caso, riconoscimento *a stato finale*, prescinde dal contenuto della pila. Il secondo caso, riconoscimento *a pila vuota*, prescinde invece dallo stato in cui si trova l'automa.

È anche possibile una modalità di riconoscimento combinata: *a stato finale e a pila vuota*.

*Proprietà 4.5.* Per la famiglia degli automi a pila indeterministici dotati d'uno o più stati interni, le modalità di accettazione

1. a pila vuota
2. a stato finale
3. combinata (stato finale e pila vuota)

hanno la stessa capacità di riconoscimento del linguaggio.

Infatti se l'automa riconosce a stato finale, esso può essere facilmente modificato in modo da riconoscere a pila vuota: quando raggiunge uno stato finale, esso entra in un nuovo stato, nel quale poi resta, svuotando la pila con mosse spontanee, fino a quando il simbolo di fondo pila affiora.

Viceversa, se l'automa riconosce a pila vuota, le modifiche, per ottenere il riconoscimento a stato finale, sono le seguenti:

- si aggiunge uno stato nuovo finale  $f$  e la mossa che in esso entra quando la pila si vuota;
- quando, per effetto d'una mossa che porta l'automa originale nello stato  $q$ , la pila cessa di essere vuota, il nuovo automa va dallo stato  $f$  allo stato  $q$ .

Analoghe considerazioni valgono per il terzo caso.

#### *Funzionamento senza cicli spontanei e in linea*

Un automa potrebbe eseguire un numero illimitato di mosse senza leggere un carattere d'ingresso; in tal caso esso entra in un *ciclo* detto *spontaneo*, composto esclusivamente di mosse spontanee. Tale comportamento potrebbe da un lato impedire all'automa di leggere per intero la stringa sorgente; dall'altro lato l'automa, a lettura completata, potrebbe aver bisogno d'un numero illimitato di mosse spontanee prima di decidere se accettare o meno. Ma questi comportamenti poco graditi si possono eliminare dagli automi a pila senza perdita di generalità.

Il seguente ragionamento<sup>2</sup> mostra che si può sempre costruire un automa equivalente privo di cicli spontanei. Tale automa può essere programmato in modo da leggere l'intera stringa sorgente, quale che sia, e di fermarsi subito al termine della lettura.

Per primo, si trasforma l'automa dato in modo che, in ogni configurazione, se  $b$  è il carattere corrente, sia definita una mossa che legge  $b$  o una mossa spontanea o entrambe.

Si osservi che, se l'automa ha un ciclo spontaneo e dunque un calcolo ciclico

---

<sup>2</sup>Per un approfondimento di questo e del successivo punto si può vedere ad es. [32, 24].

ripetibile all'infinito, tale calcolo necessariamente passa per una configurazione con uno stato  $p$  e una pila  $\gamma A$ , tali che l'automa può eseguire un numero illimitato di mosse spontanee senza consumare la cima  $A$  della pila. Secondo, si modifica la macchina aggiungendo due nuovi stati; se durante il ciclo spontaneo l'automa non può mai raggiungere una configurazione finale, si aggiunge un nuovo stato d'errore  $p_E$  e la mossa

$$p \xrightarrow{\frac{\epsilon, A}{A}} p_E$$

Altrimenti, si crea anche un nuovo stato finale  $p_F$  e le mosse

$$p \xrightarrow{\frac{\epsilon, A}{A}} p_F, \quad p_F \xrightarrow{\frac{\epsilon, A}{A}} p_E$$

Infine lo stato d'errore  $p_E$  può essere programmato in modo da consumare quanto resta da leggere della stringa sorgente.

Si dice che un automa funziona *in linea* (on line) se esso, alla lettura dell'ultimo carattere della stringa, può subito decidere se accettarla, senza dover fare ulteriori mosse.

Un automa a pila può sempre essere convertito in un automa equivalente del tipo in linea. Grazie al risultato precedente si può supporre che l'automa sia privo di cicli spontanei. Al termine della lettura della stringa, esso nello stato  $p$  potrebbe aver bisogno d'un numero finito di mosse spontanee che esaminano una parte finita ("tappo") della pila, di lunghezza massima  $k$ , prima accettare (o di rifiutare). Nel primo caso il tappo si dirà di accettazione, nel secondo di rifiuto. Si modifica l'automa in modo che esso durante il calcolo memorizzi il tappo anche nello stato. Allora esso, invece di entrare nello stato  $p$ , entrerà in uno stato che rappresenta la combinazione di  $p$  e del tappo. Se il tappo era di accettazione, l'automa accetta altrimenti rifiuta.

## 4.2 Linguaggi liberi e automi a pila: una sola famiglia

Si dimostra ora che il linguaggio accettato da un automa a pila dotato di stati è libero e di conseguenza, poiché si sa (p. 150) che ogni linguaggio libero è riconosciuto da un automa a pila, vale l'enunciato seguente.

→ **Proprietà 4.6.** La famiglia *LIB* dei linguaggi liberi coincide con quella dei linguaggi riconosciuti da automi a pila.

*Dimostrazione.* Dato il linguaggio  $L = L(M)$ , riconosciuto da un automa a pila  $M$  dotato d'uno o più stati, con modalità di accettazione a pila vuota, si costruirà ora la grammatica  $G$  che lo genera. Essa non è pulita e dovrà essere rifinita eliminando le regole inutili. La seguente tabella mostra i componenti dell'automa (dato) e della grammatica (da costruire):

---

*automa*

---

Alfabeto terminale:  $\Sigma = \{a, \dots\}$ Stati:  $Q = \{q_0, q_1, q_i, \dots\}$ Stato iniziale:  $q_0$ Simboli di pila:  $\Gamma = \{A, B, B_i, \dots\}$ Simbolo iniziale:  $Z_0$ Mossa:  $q \xrightarrow{\overline{B_n B_{n-1} \dots B_1} \atop a, A} q_1$ 

---

*grammatica*

---

Nonterminali:  $\langle q_i, A, q_j \rangle$ dove  $q_i, q_j \in Q$  e  $A \in \Gamma$ Assioma  $S$ Regole iniziali:  $\forall q \in Q : S \rightarrow \langle q_0, Z_0, q \rangle$ Regole:  $\langle q, A, q_m \rangle \rightarrow a \langle q_1, B_1, q_2 \rangle \langle q_2, B_2, q_3 \rangle \dots \langle q_{m-1}, B_{m-1}, q_m \rangle$ per ogni  $a \in \Sigma \cup \{\varepsilon\}$ ,per ogni stato  $q, q_1, q_2, \dots, q_m \in Q$ ,e per ogni simbolo di pila  $A, B_1, B_2, \dots, B_m \in \Gamma$ tali che esiste la mossa  $q \xrightarrow{\overline{B_m B_{m-1} \dots B_1} \atop a, A} q_1$ .Nel caso particolare  $m = 0$ : la regola è  $\langle q, A, q_1 \rangle \rightarrow a$ 

Le regole della grammatica sono costruite in modo che, a ogni calcolo valido dell'automa, corrisponda una derivazione sinistra. Come nella vecchia corrispondenza tra grammatiche e automi (p. 148), un nonterminale è associato a un simbolo della pila; in più un nonterminale è ora contraddistinto da due stati, che hanno questo significato. Dal nonterminale  $\langle q, A, r \rangle$  si deriva la stringa  $z$ , se, e solo se, l'automa, partendo nello stato  $q$  con  $A$  in cima alla pila, esegue un calcolo che legge la stringa  $z$ , raggiunge lo stato  $r$  e cancella  $A$  dalla pila.

Omettendo la dimostrazione<sup>3</sup> della correttezza della costruzione, si passa a un esempio.

*Esempio 4.7.* Grammatica equivalente all'automa a pila (Harrison).  
Il linguaggio

$$L = \{a^n b^m a^n \mid m, n \geq 1\} \cup \{a^n b^m c^m \mid m, n \geq 1\}$$

è l'unione di due linguaggi fatti di tre parti concatenate. Le prime due parti sono le stesse, mentre la terza parte usa la lettera  $a$  nel primo linguaggio e la lettera  $c$  nel secondo. Nel primo linguaggio il primo e il terzo gruppo sono

---

<sup>3</sup>Vedasi ad es. [24, 27, 28, 43].

eguali; nel secondo, il secondo e terzo gruppo hanno eguale lunghezza. Il linguaggio è accettato a pila vuota da un automa a tre stati, che opera, con i seguenti passi:

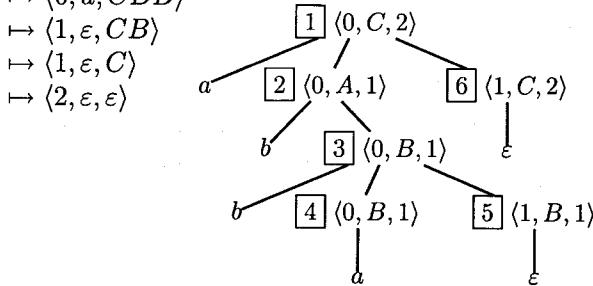
1. il simbolo iniziale in pila è  $C$  e lo stato iniziale è 0;
2. impila le lettere  $a$  (come  $A$ ) e poi le  $b$  (come  $B$ ), via via che sono lette;
3. leggendo la prima lettera della terza parte, va nello stato 1 se è  $a$  o nello stato 2 se è  $c$ ;
4. nello stato 1 cancella tutte le  $B$  dalla pila, e poi pareggia le rimanenti  $a$  dell'ingresso con le  $A$  presenti nella pila;
5. nello stato 2, pareggia le rimanenti  $c$  in ingresso con le  $B$  presenti nella pila, poi cancella le  $A$  dalla pila;

Nella prima colonna della prossima tabella sono mostrate le mosse dell'automa, con accanto le regole costruite per la grammatica. Le regole inutili sono state omesse.

| <i>mossa</i>   | <i>regole</i>  |
|--|--|
| $0 \xrightarrow{a,C} 0$<br>$\xrightarrow{CA}$              | $\langle 0, C, 2 \rangle \rightarrow a \langle 0, A, 1 \rangle \langle 1, C, 2 \rangle \mid a \langle 0, A, 2 \rangle \langle 2, C, 2 \rangle$                                       |
| $0 \xrightarrow{a,A} 0$<br>$\xrightarrow{AA}$              | $\langle 0, A, 1 \rangle \rightarrow a \langle 0, A, 1 \rangle \langle 1, A, 1 \rangle, \quad \langle 0, A, 2 \rangle \rightarrow a \langle 0, A, 2 \rangle \langle 2, A, 2 \rangle$ |
| $0 \xrightarrow{b,A} 0$<br>$\xrightarrow{B}$               | $\langle 0, A, 1 \rangle \rightarrow b \langle 0, B, 1 \rangle, \quad \langle 0, A, 2 \rangle \rightarrow b \langle 0, B, 2 \rangle$   |
| $0 \xrightarrow{b,B} 0$<br>$\xrightarrow{BB}$              | $\langle 0, B, 1 \rangle \rightarrow b \langle 0, B, 1 \rangle \langle 1, B, 1 \rangle, \quad \langle 0, B, 2 \rangle \rightarrow b \langle 0, B, 2 \rangle \langle 2, B, 2 \rangle$ |
| $0 \xrightarrow{a,B} 1$<br>$\xrightarrow{\epsilon}$        | $\langle 0, B, 1 \rangle \rightarrow a$  |
| $1 \xrightarrow{\epsilon,B} 1$<br>$\xrightarrow{\epsilon}$ | $\langle 1, B, 1 \rangle \rightarrow \epsilon$   |
| $1 \xrightarrow{a,A} 1$<br>$\xrightarrow{\epsilon}$        | $\langle 1, A, 1 \rangle \rightarrow a$  |
| $1 \xrightarrow{\epsilon,C} 2$<br>$\xrightarrow{\epsilon}$ | $\langle 1, C, 2 \rangle \rightarrow \epsilon$   |
| $0 \xrightarrow{c,B} 2$<br>$\xrightarrow{\epsilon}$        | $\langle 0, B, 2 \rangle \rightarrow c$  |
| $2 \xrightarrow{c,B} 2$<br>$\xrightarrow{\epsilon}$        | $\langle 2, B, 2 \rangle \rightarrow c$  |
| $2 \xrightarrow{\epsilon,A} 2$<br>$\xrightarrow{\epsilon}$ | $\langle 2, A, 2 \rangle \rightarrow \epsilon$   |
| $2 \xrightarrow{\epsilon,C} 2$<br>$\xrightarrow{\epsilon}$ | $\langle 2, C, 2 \rangle \rightarrow \epsilon$   |

L'assioma è  $\langle 0, C, 2 \rangle$ .

Segue un calcolo dell'automa e il corrispondente albero di derivazione, con i passi numerati:

$$\begin{aligned}
 \langle 0, abba, C \rangle &\mapsto \langle 0, bba, CA \rangle \\
 &\mapsto \langle 0, ba, CB \rangle \\
 &\mapsto \langle 0, a, CBB \rangle \\
 &\mapsto \langle 1, \epsilon, CB \rangle \\
 &\mapsto \langle 1, \epsilon, C \rangle \\
 &\mapsto \langle 2, \epsilon, \epsilon \rangle
 \end{aligned}$$


La corrispondenza tra i due modelli si comprende confrontando il calcolo

$$\langle 0, abba, C \rangle \xrightarrow{3} \langle 0, a, CBB \rangle$$

con la derivazione sinistra

$$\langle 0, C, 2 \rangle \xrightarrow{3} abb \langle 0, B, 1 \rangle \langle 1B, 1 \rangle \langle 1, C, 2 \rangle$$

grazie alle seguenti osservazioni:

- il prefisso della stringa, letto dall'automa, e quello generato dalla derivazione sono identici: *abb*
- il contenuto della pila *CBB* è la stringa riflessa di quella ottenuta concatenando i simboli centrali delle terne (nonterminali) presenti nella stringa derivata;
- due terne (nonterminali) adiacenti nella stringa derivata sono “incatenate” dall’egualanza degli stati contraddistinti dallo stesso tipo di freccia:

$$\langle 0, B, 1 \rangle \langle 1B, 1 \rangle \langle 1, C, 2 \rangle$$

Grazie a queste corrispondenze, che valgono in ogni passo, i due modelli definiscono lo stesso linguaggio.

#### 4.2.1 Intersezione di linguaggi liberi e regolari

Come illustrazione dei concetti precedenti, l'affermazione (cap. 2, p. 78), che l'intersezione d'un linguaggio libero con uno regolare è un linguaggio libero, è ora facile da giustificare. Data la grammatica  $G$  e l'automa finito  $A$ , si mostra come ottenere l'automa a pila  $M$  che riconosce  $L(G) \cap L(A)$ .

Prima si costruisce, con il metodo di p. 148, l'automa  $N$ , avente un solo stato, che a pila vuota riconosce il linguaggio  $L(G)$ .

Poi si costruisce la macchina  $M$  prodotto cartesiano delle due macchine  $N$  e  $A$ , applicando sostanzialmente la stessa costruzione per gli automi finiti (p. 138). L'unica modifica riguarda la pila: la macchina prodotto  $M$  esegue sulla

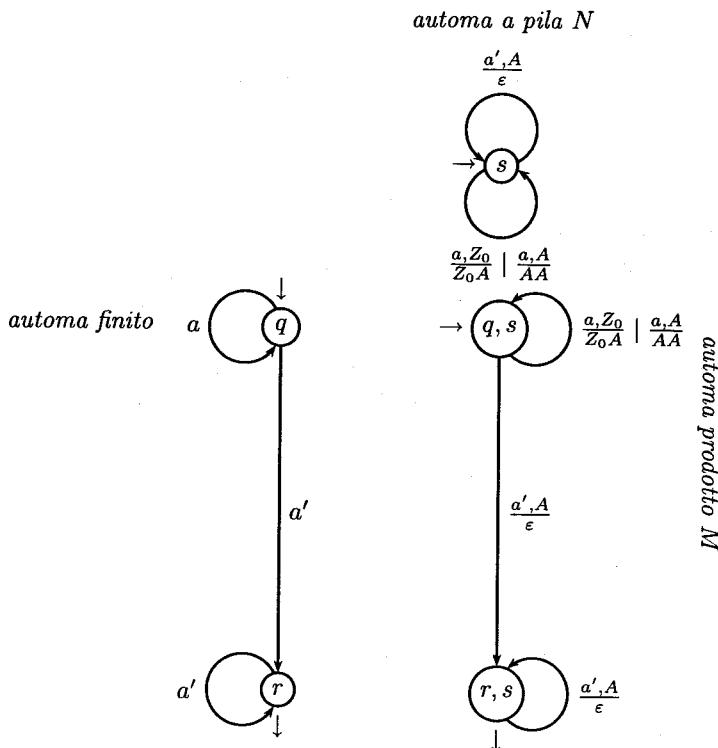
pila le stesse operazioni dell'automa componente  $N$ . La macchina ottenuta ha dunque come insieme degli stati il prodotto cartesiano degli stati delle macchine componenti. Essa riconosce mediante stato finale e pila vuota; sono finali gli stati che contengono uno stato finale dell'automa finito  $A$ .

Circa il determinismo, la macchina è deterministica se entrambe le macchine componenti lo sono.

È facile comprendere che l'automa a pila riconosce a stato finale una stringa se, e solo se, entrambe le macchine componenti la accettano, ossia se essa appartiene all'intersezione dei due linguaggi.

*Esempio 4.8.* Si vuole (come nell'es. 2.82 di p. 80) l'intersezione del linguaggio di Dyck di alfabeto  $\Sigma = \{a, a'\}$  con il linguaggio regolare  $a^*a'^+$ . Il risultato è il linguaggio  $\{a^n a'^n \mid n \geq 1\}$ .

Il linguaggio di Dyck è facilmente riconosciuto a pila vuota da un automa (deterministico) con un solo stato. I riconoscitori dei due linguaggi e l'automa del prodotto cartesiano sono:



Ad es. l'arco da  $\{q, s\}$  a  $\{r, s\}$ , associato alla lettura di  $a'$ , opera sulla pila come l'automa  $N$  cancellando una  $A$  e va dallo stato  $q$  allo stato  $r$  come l'automa finito.

Poiché l'automa a pila componenti accetta a pila vuota, e quello finito riconosce nello stato finale  $r$ , la macchina costruita riconosce a pila vuota nello stato finale  $(r, s)$ .

#### 4.2.2 Automi a pila e linguaggi deterministici

È importante approfondire lo studio dei riconoscitori deterministici e dei loro linguaggi, perché sono i più usati nei compilatori a ragione della loro efficienza. Nel comportamento d'un automa a pila (definito come a p. 151), si possono incontrare tre forme di indeterminismo.

- 1. Incertezza tra più mosse di lettura, se per stato  $q$ , carattere  $a$  e simbolo  $A$  di pila, la funzione di transizione ha più valori:  $|\delta(q, a, A)| > 1$ .
- 2. Incertezza tra una mossa spontanea e una mossa di lettura, ossia sono definite entrambe le mosse  $\delta(q, \varepsilon, A)$  e  $\delta(q, a, A)$ .
- 3. Incertezza tra più mosse spontanee; ossia per qualche stato  $q$  e simbolo  $A$  di pila, la funzione  $\delta(q, \varepsilon, A)$  ha più valori:  $|\delta(q, \varepsilon, A)| > 1$ .

Se, nella funzione di transizione, nessuna delle forme 1, 2 e 3 è presente, l'automa a pila è *deterministico*. Il linguaggio riconosciuto da un automa deterministico è detto (libero) *deterministico*, e la famiglia di tali linguaggi è chiamata *DET*.

*Esempio 4.9.* Forme di indeterminismo.

Il riconoscitore (p. 148) del linguaggio

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

ha un indeterminismo della forma 1

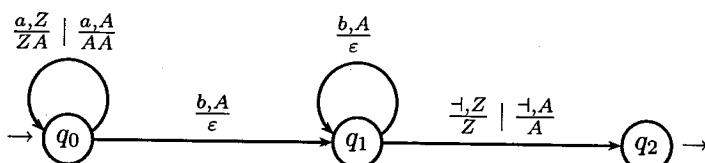
$$\delta(q_0, a, A) = \{(q_0, b), (q_0, bA)\}$$

e anche della forma 2

$$\delta(q_0, \varepsilon, S) = \{(q_0, A)\} \quad \delta(q_0, a, S) = \{(q_0, S)\}$$

(L'unico stato è indicato come  $q_0$ ).

Lo stesso linguaggio è riconosciuto deterministicamente dall'automa  $M_2 = (\{q_0, q_1, q_2\}, \{a, b\}, \{A, Z\}; \delta, q_0, Z, \{q_2\})$ :



Intuitivamente  $M$  impila le  $a$ , codificate come  $A$ ; leggendo la prima  $b$  cancella una  $A$  e va nello stato  $q_1$ . Poi, per ogni  $b$  letta, disimpila una  $A$ . Se vi fossero più  $b$  che  $a$ , esso cadrebbe in errore. Leggendo il terminatore, esso passa nello stato  $q_2$  finale, quale che sia il simbolo in cima.

### Proprietà di chiusura dei linguaggi deterministici

Essendo i linguaggi deterministici una sottoclasse dei linguaggi libri, essi godono di proprietà specifiche. Si aggiorna qui il quadro delle proprietà, delle famiglie di linguaggi, continuando quello presentato a p. 78. Si indica con  $L$ ,  $D$  e  $R$  un linguaggio appartenente rispettivamente alla famiglia  $LIB$ ,  $DET$  e  $REG$ .



| Operazione     | Proprietà                                   | (Proprietà nota)          |
|----------------|---|---------------------------|
| Riflessione    | $D^R \notin DET$                            | $D^R \in LIB$             |
| Stella         | $D^* \notin DET$                            | $D^* \in LIB$             |
| Complemento    | $\neg D \in DET$                            | $\neg L \notin LIB$       |
| Unione         | $D_1 \cup D_2 \notin DET, D \cup R \in DET$ | $D_1 \cup D_2 \in LIB$    |
| Concatenamento | $D_1.D_2 \notin DET, D.R \in DET$           | $D_1.D_2 \in LIB$         |
| Intersezione   | $D \cap R \in DET$                          | $D_1 \cap D_2 \notin LIB$ |

Seguono alcune giustificazioni ed esempi.<sup>4</sup>

Riflessione: il linguaggio

$$L_1 = \{a^n b^n e\} \cup \{a^n b^{2n} d\}, n \geq 1$$

soddisfa l'eguaglianza  $|x|_a = |x|_b$  quando la frase termina con  $e$ , e l'eguaglianza  $2|x|_a = |x|_b$  se la frase termina con  $d$ . Esso non è deterministico, ma il linguaggio speculare sì. Infatti la lettura del primo carattere permette all'automa di scegliere quale eguaglianza applicare in seguito.

Complemento: il complemento d'un linguaggio deterministico è tale; la dimostrazione (analogia a quella dei linguaggi regolari di p. 136) costruisce il riconoscitore del complemento scambiando gli stati finali e non, e creando lo stato pozzo.<sup>5</sup> Se dunque un linguaggio libero ha come complemento un linguaggio non libero, esso non può essere deterministico.

<sup>4</sup> Al solito un enunciato come ad es.  $D^R \notin DET$  va inteso nel senso che esiste un linguaggio  $D$  tale che  $D^R$  non è deterministico.

<sup>5</sup> Si veda ad es. [24, 27]. Esiste anche un metodo [26] per trasformare la grammatica d'un linguaggio deterministico in quella del suo complemento.

Unione: l'es. 4.12 di p. 164 mostrerà che l'unione di linguaggi deterministici (tali sono ovviamente  $L'$  e  $L''$ ) non è in generale deterministica.

Per l'identità di De Morgan si ha  $D \cup R = \neg(\neg D \cap \neg R)$ , che è un linguaggio deterministico, per via dei punti seguenti: il complemento d'un linguaggio deterministico (regolare) è deterministico (regolare); l'intersezione d'un linguaggio deterministico e d'uno regolare è deterministica (punto successivo).

Intersezione  $D \cap R$ : l'automa prodotto cartesiano di un automa a pila deterministico e d'un automa finito deterministico è deterministico.

Per mostrare che l'intersezione di due linguaggi deterministici può uscire da  $DET$ , basta ricordare il linguaggio, non libero, dei tre esponenti (p. 76); esso è l'intersezione di due linguaggi facilmente deterministici:

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

Concatenamento e stella: concatenando due linguaggi deterministici, può accadere che non si riesca a individuare deterministicamente la fine delle stringhe del primo linguaggio, quindi il punto a partire dal quale occorre riconoscere le stringhe del secondo linguaggio. Esempio: dati i linguaggi

$$L_0 = \{a^i b^i a^j \mid i, j \geq 1\} \quad L_1 = \{a^i b^j a^j \mid i, j \geq 1\} \quad R = \{c, c^2\}$$

il linguaggio  $L = cL_0 \cup L_1$  è deterministico, ma il concatenamento  $RL$  no.<sup>6</sup> Infatti la presenza d'un prefisso  $cc$  può essere interpretata in due modi, provenienti da  $c.cL_0$  oppure da  $c^2.L_1$ .

Analoga situazione si incontra prendendo la stella d'un linguaggio deterministico.

Concatenamento con linguaggio regolare: il riconoscitore di  $D.R$  può essere costruito componendo *in cascata* l'automa a pila deterministico e l'automa finito deterministico. Più precisamente, quando l'automa a pila entra nello stato finale che riconosce una frase di  $D$ , la nuova macchina passa nello stato iniziale dell'automa che riconosce  $R$  e lo simula, da quel punto in avanti.

Si nota dalla tabella che le operazioni di base della teoria dei linguaggi non preservano il determinismo. Questa può creare qualche difficoltà per il progettista d'un linguaggio artificiale: basti pensare che, unendo due linguaggi esistenti deterministici, il risultato può essere indeterministico (o addirittura ambiguo). Lo stesso dicasi per il concatenamento e la stella. Il progettista dovrà assicurarsi che la proprietà di determinismo sia mantenuta, dopo ogni modifica del linguaggio da progettare.

Infine conviene menzionare che, tra  $DET$  e  $LIB$ , vi è una sostanziale differenza rispetto alle proprietà decidibili: dati due automi a pila deterministici, esiste un algoritmo<sup>7</sup> per decidere se essi sono equivalenti, ossia se riconoscono lo stesso linguaggio, mentre per automi a pila generici ciò non è decidibile.

<sup>6</sup>La dimostrazione si trova in [24].

<sup>7</sup>Vedasi [44].

## Linguaggi non deterministici

Diversamente dal caso dei linguaggi regolari, vi sono linguaggi liberi che non possono essere riconosciuti da un automa deterministico.

*Proprietà 4.10.* La famiglia  $DET$  dei linguaggi deterministici è contenuta strettamente in quella  $LIB$  dei linguaggi liberi.

L'enunciato segue da due fatti noti: l'inclusione  $DET \subseteq LIB$ , essendo l'automa deterministico a pila un caso particolare dell'automa a pila generale; e la disegualanza  $DET \neq LIB$ , poiché certe proprietà di chiusura valgono per una famiglia ma non per l'altra. Ciò basterebbe, ma è interessante mostrare direttamente che certi linguaggi liberi non sono deterministici, proprio allo scopo di riconoscere tali paradigmi per evitarli nell'uso pratico.

### Lemma del doppio servizio

Uno strumento concettuale per dimostrare che un certo linguaggio libero non è deterministico si basa sull'analisi delle frasi che sono un prefisso di altre frasi.

Sia  $D$  un linguaggio deterministico,  $x \in D$  una sua frase, e vi sia un'altra frase  $y \in D$  tale da essere un prefisso di  $x$ , ovvero  $x = yz$ , dove tutte le stringhe possono essere vuote.

Si definisce ora un nuovo linguaggio, detto il linguaggio del *doppio servizio* di  $D$ , ottenuto inserendo un nuovo terminale diesis  $\sharp$ , tra le stringhe  $y$  e  $z$ :

$$ds(D) = \{y\sharp z \mid y \in D \wedge z \in \Sigma^* \wedge yz \in D\}$$

Ad es. per il linguaggio  $F = \{a, ab, bb\}$  si ha

$$ds(F) = \{a\sharp b, a\sharp, ab\sharp, bb\sharp\}$$

dove appaiono le frasi originali terminate da  $\sharp$  e anche le frasi segmentate dal diesis nel prefisso (appartenente al linguaggio) e nel suffisso.

*Lemma 4.11.* Se  $D$  è un linguaggio deterministico, allora anche il linguaggio  $ds(D)$  del doppio servizio è deterministico.

Per dimostrare l'enunciato, si trasforma il riconoscitore deterministico  $M$  di  $D$  in un automa deterministico  $M'$ , che riconosce il linguaggio del doppio servizio, come ora spiegato. Conviene supporre che l'automa  $M$  funzioni *in linea* (p. 154); ossia che, via via che scandisce la stringa d'ingresso, esso possa decidere se la stringa letta è una frase valida.

Si consideri il calcolo con cui  $M$  accetta la stringa  $y$ . Se la stringa  $y$  è seguita dal diesis, la nuova macchina  $M'$  lo legge e accetta se non vi sono altri caratteri (poiché  $y\sharp \in ds(D)$  se  $y \in D$ ). Altrimenti  $M'$  prosegue deterministicamente il calcolo scandendo la stringa  $z$ , nello stesso modo con cui  $M$  avrebbe riconosciuto la stringa  $yz$ .

Le ragioni del nome del linguaggio sono ora chiare: l'automa svolge contemporaneamente due servizi, in quanto riconosce un prefisso e una stringa più lunga.

Per applicare il lemma a casi concreti, basta osservare che, se il doppio servizio d'un linguaggio libero  $L$  non è libero,  $L$  non è deterministico.

*Esempio 4.12.* Unione non deterministica di linguaggi deterministici.

### Il linguaggio

$$L = \{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\} = L' \cup L''$$

unione di due linguaggi deterministici, non è deterministico.

Intuitivamente l'automa dovrebbe leggere e impilare le  $a$  e, se la stringa (ad es.  $aabb$ ) appartiene al primo insieme, disimpilare una  $a$  alla lettura d'una  $b$ ; ma se la stringa appartiene al secondo insieme (ad es.  $aabbba$ ), sono due le  $b$  da leggere per disimpilare una  $a$ . Non potendo l'automa sapere quale sia la strada giusta (per saperlo dovrebbe contare le  $b$  esaminando una parte illimitata della stringa), è obbligato a tentare entrambe le vie.

Più rigorosamente, se per assurdo  $L$  fosse deterministico, anche il linguaggio  $ds(L)$  lo sarebbe, e, per la proprietà di chiusura di  $DET$  (p. 161) rispetto all'intersezione con  $REG$ , anche il linguaggio

$$L_R = ds(L) \cap (a^+ b^+ \# b^+)$$

dovrebbe essere deterministico. Ma  $L_R$  non è neppure un linguaggio libero, quindi tanto meno un linguaggio deterministico, perché le sue frasi hanno la forma  $a^i b^i \# b^i$ ,  $i \geq 1$ , con tre esponenti eguali (p. 76) e si sa che tale linguaggio non è libero.

*Esempio 4.13.* Palindromi.

Analoghe considerazioni mostrano che non è deterministico il linguaggio dei palindromi,  $L$  definito dalla grammatica:

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \epsilon$$

Anche questa dimostrazione interseca il linguaggio del doppio servizio con un linguaggio regolare, allo scopo di ottenere un linguaggio che fuoriesca dalla famiglia  $LIB$ . Si guardi il linguaggio

$$L_R = ds(L) \cap (a^* ba^* \# ba^*)$$

Una stringa della forma  $a^i b a^j \# b a^k$  appartiene a  $L_R$  se, e solo se,  $j = i \wedge k = i$ . Ma tale linguaggio è di nuovo quello dei tre esponenti, che non è libero.

### Determinismo e inambiguità del linguaggio

Se un linguaggio è accettato da un automa deterministico, ogni frase è riconosciuta con uno e un solo calcolo, e si può dimostrare che il linguaggio può

essere generato da una grammatica inambigua.

Si riguardi il procedimento di p. 155, per costruire la grammatica equivalente all'automa, che ne simula i calcoli mediante le proprie derivazioni. La grammatica genera una frase con una certa derivazione sinistra, se, e soltanto se, l'automa ha un corrispondente calcolo che riconosce la stessa frase. In altre parole, due diverse derivazioni sinistre corrispondono necessariamente a calcoli diversi, che accettano stringhe diverse. Di conseguenza vale il seguente enunciato.

→ **Proprietà 4.14.** Sia  $M$  un automa a pila deterministico; allora la corrispondente grammatica di  $L(M)$ , costruita con il procedimento di p. 155, non è ambigua.

Naturalmente non è detto che ogni grammatica del linguaggio sia inambigua. D'altra parte, si ha come corollario che un linguaggio libero inerentemente ambiguo non è deterministico. Infatti, se esso per assurdo fosse deterministico, la proprietà precedente permetterebbe di costruire una grammatica inambigua, mentre per la definizione stessa (p. 56), ogni grammatica del linguaggio è ambigua. Per un linguaggio inerentemente ambiguo non esiste dunque un riconoscitore a pila deterministico.

**Esempio 4.15.** Linguaggio inerentemente ambiguo e indeterminismo.

Il linguaggio inerentemente ambiguo dell'es. 2.56 (p. 56) può essere scritto come l'unione dei linguaggi

$$L_A = \{a^i b^i c^* \mid i \geq 0\} \cup \{a^* b^i c^i \mid i \geq 0\} = L_1 \cup L_2$$

entrambi deterministici (un'altra prova della non chiusura di  $DET$  rispetto all'unione).

Intuitivamente, per riconoscere il linguaggio, un automa deve attuare due metodi necessariamente diversi per le stringhe del primo e del secondo insieme. Nel primo caso deve registrare nella pila le  $a$  lette e depennarle, via via che si leggono le  $b$ ; e similmente nel secondo caso, con rispettivamente i simboli  $b$  e  $c$ . Di conseguenza, le frasi, che appartengono a entrambi gli insiemi, sono accettate con due calcoli diversi, e l'automa è indeterministico.

Anche per gli automi, si può definire il concetto di ambiguità. Un *automa* è detto *ambiguo* se esiste una frase del linguaggio che è accettata con due calcoli distinti.

Si noti che il concetto di determinismo è più restrittivo di quello di inambiguità: la famiglia degli automi a pila deterministici è strettamente contenuta in quella degli automi a pila inambigui.

Per chiarire l'affermazione conviene riprendere due esempi.

**Esempio 4.16.** Il linguaggio  $L_A$  dell'es. precedente è ambiguo oltre che non deterministico, perché certe frasi sono necessariamente riconosciute da due calcoli diversi.

D'altra parte, il linguaggio dell'es. 4.12

$$L_I = \{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\} = L' \cup L''$$

pur non essendo deterministico (come prima dimostrato), è inambiguo. Infatti i due linguaggi  $L', L''$  da unire sono disgiunti. I due modi di funzionamento corrispondenti (descritti a p. 164) determinano due calcoli diversi, uno solo dei quali può terminare con successo.

### Sottoclassi degli automi deterministici a pila

La famiglia  $DET$  per definizione è quella dei linguaggi riconosciuti dal modello più generale di automa a pila deterministico, quello che possiede più stati e riconosce a stato finale. Diverse limitazioni sugli stati interni e sui modi di riconoscimento, che sono senza effetto nel caso indeterministico, causano una restrizione dei linguaggi riconoscibili da un automa deterministico. Le principali restrizioni sono qui brevemente ricordate.<sup>8</sup>

- Automa con un solo stato: il riconoscimento a pila vuota risulta meno potente di quello a stato finale.
- Limitazioni sul numero degli stati interni: causano una restrizione nella famiglia dei linguaggi riconoscibili; simile effetto hanno le limitazioni sul numero delle configurazioni finali dell'automa.
- Funzionamento in tempo reale, cioè senza mosse spontanee: restringe la classe riconoscibile.

### Semplici sottofamiglie deterministiche

In molti ambiti applicativi dell'informatica i linguaggi sono progettati in modo da essere analizzabili deterministicamente: l'esemplificazione comprende quasi tutti i linguaggi programmativi e i linguaggi XML della Rete. Vi sono più modi per assicurare a priori che un linguaggio sia deterministico, imponendo restrizioni sul linguaggio stesso o sulla sua grammatica. Si ottengono così più sottoclassi di  $DET$ , tra le quali spiccano per semplicità le due prossime. Altri casi più importanti per le applicazioni corrispondono ai linguaggi  $LL(k)$  e  $LR(k)$  accettati dagli algoritmi veloci di analisi sintattica studiati più avanti.

#### *Linguaggi deterministici semplici*

Una grammatica è detta *deterministica semplice* se soddisfa le seguenti condizioni:

1. ogni parte destra d'una regola inizia con un carattere terminale (pertanto non vi sono regole nulle);
2. per ogni nonterminale  $A$  non esistono due alternative che iniziano con lo stesso carattere terminale:

$$\not\exists A \rightarrow a\alpha \mid a\beta, \text{ con } a \in \Sigma, \alpha, \beta \in (\Sigma \cup V)^*, \alpha \neq \beta$$

---

<sup>8</sup>Per approfondimenti si veda [24, 45].

Un esempio è la grammatica  $S \rightarrow aSb \mid c$ .

Costruendo, mediante la corrispondenza (p. 148) tra le regole e le mosse, l'automa, esso risulta chiaramente deterministico e consuma un carattere a ogni mossa, ossia funziona in tempo reale.

Ma questa restrizione rappresenta un ostacolo eccessivo per un uso pratico della grammatica.

### *Linguaggi a parentesi*

I linguaggi a parentesi, introdotti nel cap. 2 a p. 43, sono deterministici, come è facile vedere. Infatti ogni frase generata da una grammatica parentesizzata, ha una struttura a parentesi che marca l'inizio e la fine della parte destra d'una regola sintattica. Si suppone che la grammatica sia parentesizzata con segni distinti (p. 45) o che, se essa usa un solo tipo di parentesi, non vi siano regole con la stessa parte destra, al fine di garantire l'assenza di ambiguità. Un semplicissimo algoritmo di riconoscimento è il seguente. Si scandisce la stringa fino a trovare una coppia di parentesi priva di parentesi al proprio interno. Si ricerca la stringa compresa tra le parentesi (estremi inclusi) tra le parti destre delle regole. Se non la si trova il riconoscimento fallisce. Altrimenti si riduce tale stringa al nonterminale corrispondente (parte sinistra della regola). Poi si riprende la ricerca d'una coppia di parentesi allo stesso modo. Si riconosce la stringa sorgente se all'ultimo passo essa si riduce all'assioma.

Non è difficile programmare l'algoritmo sotto forma d'un automa a pila deterministico.

I documenti della Rete nel formato XML sono punteggiati da marche di apertura e di chiusura, che delimitano le varie parti del documento e permettono un loro efficiente riconoscimento. La famiglia dei linguaggi XML è dunque anch'essa deterministica. Essa è simile a quella dei linguaggi definiti dalle grammatiche parentesizzate, ma se ne distacca nel permettere l'uso degli operatori delle espressioni regolari all'interno delle regole grammaticali.<sup>9</sup>

---

<sup>9</sup>Le grammatiche XML, pur generando linguaggi liberi, si scostano da quelle BNF in vari aspetti. Per un confronto iniziale si veda [9].

### 4.3 Analisi sintattica

Il resto del capitolo espone gli algoritmi di parsificazione più diffusi, iniziando dai veloci metodi deterministici discendenti e ascendenti, anche per grammatiche estese con espressioni regolari, e terminando con un algoritmo generale di complessità maggiore, ma sempre polinomiale.

Data una grammatica  $G$ , l'analizzatore sintattico o parsificatore legge la stringa *sorgente*, e se appartiene al linguaggio  $L(G)$ , ne produce una derivazione o un albero sintattico; altrimenti si ferma indicando la configurazione in cui l'errore è comparso (diagnosi); eventualmente prosegue l'analisi, saltando le sottostringhe contaminate dall'errore, ossia ripara l'effetto dell'errore.

Trascurando il trattamento degli errori, un analizzatore è dunque un riconoscitore, arricchito con la capacità di produrre la derivazione della stringa. A tale fine, basta che l'automa, quando compie la mossa corrispondente al riconoscimento d'una regola grammaticale, salvi in una struttura dati l'etichetta che la identifica. La struttura, al termine dell'analisi, rappresenterà l'albero sintattico.

Se la stringa sorgente è ambigua, il risultato è un insieme di alberi (o di derivazioni).

#### 4.3.1 Analisi discendente e ascendente

Si sa che uno stesso albero corrisponde a più derivazioni: quella sinistra, quella destra (e tutte le altre di minore interesse). A seconda che si consideri una derivazione destra o sinistra, e dell'ordine in cui essa è ricostruita, si ottengono due classi importanti di analizzatori.

**Analisi discendente:** costruisce la derivazione sinistra, procedendo dall'assio-  
ma, ossia dalla radice dell'albero, verso le foglie; i passi dell'algoritmo sono  
in corrispondenza con le derivazioni immediate.

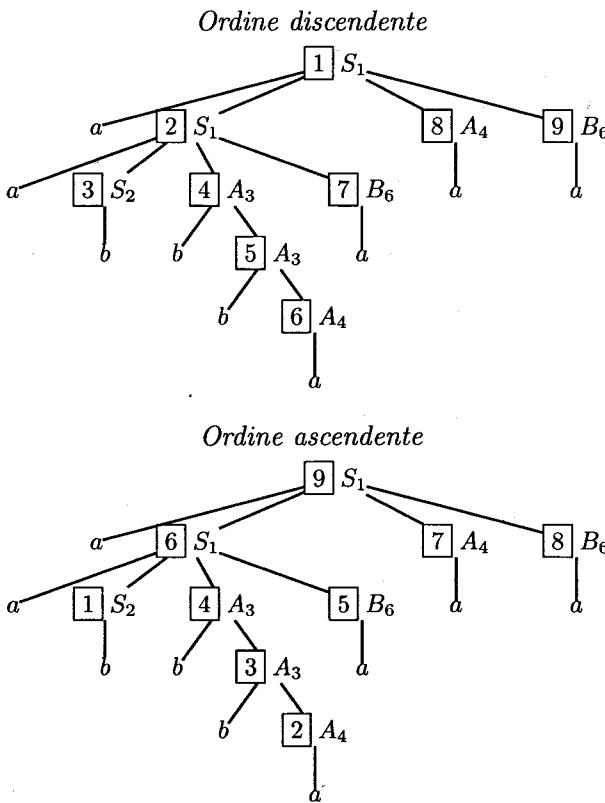
**Analisi ascendente:** costruisce la derivazione destra della stringa, ma nell'or-  
dine riflesso, cioè, come si vedrà, dalle foglie alla radice dell'albero; i passi  
dell'algoritmo sono in corrispondenza con le operazioni di riduzione.

*Esempio 4.17.* Ordini di visita dell'albero.

Per la grammatica

- |   |  |
|---|--|
| 1. $S \rightarrow aSAB$<br>3. $A \rightarrow bA$<br>5. $B \rightarrow cB$ | 2. $S \rightarrow b$<br>4. $A \rightarrow a$<br>6. $B \rightarrow a$ |
|---|--|

e la frase  $a^2b^3a^4$ , gli ordinamenti corrispondenti alla visita discendente e ascendente sono indicati nelle seguenti figure dai numeri inquadrati, mentre i pedici denotano la regola applicata.



Un analizzatore discendente sviluppa le parti sinistre delle regole nelle corrispondenti parti destre. Se queste contengono dei simboli terminali, essi devono combaciare con quelli presenti nel testo sorgente. Il processo termina quando tutti i simboli nonterminali sono stati trasformati in simboli terminali, che combaciano con il testo (o in stringhe vuote).

Nell'analisi ascendente si riducono in un nonterminale le parti destre delle regole via via incontrate, prima nella stringa sorgente, poi nelle forme di frase da essa ottenute mediante le riduzioni. Il processo termina quando l'intera stringa si è ridotta all'assioma.

In entrambi i casi il processo si interrompe incontrando un errore.

In linea di principio, l'analisi potrebbe invertire l'ordine di scansione, leggendo la stringa dal fondo all'inizio. Ma la totalità dei linguaggi artificiali è progettata per essere elaborata da sinistra a destra (come del resto avviene nella lingue umane dove l'ordine naturale di lettura corrisponde a quello di emissione della frase da parte del parlatore). Per inciso, il cambio di direzione nella scansione del testo può invalidare la proprietà di determinismo d'un linguaggio, perché la famiglia dei linguaggi deterministici non è chiusa rispetto alla riflessione speculare.

### 4.3.2 La grammatica come rete di automi finiti

Conviene rappresentare la grammatica mediante una rete di automi finiti, allo scopo di facilitare e unificare l'esposizione degli algoritmi di parsificazione.

Data una grammatica  $G$ , ogni nonterminale è la parte sinistra di una o più alternative. Se la grammatica è nella forma detta BNF estesa, nella parte destra vi sono anche gli operatori delle espressioni regolari. In tale caso si può supporre che ogni nonterminale abbia una sola regola  $A \rightarrow \alpha$ , dove  $\alpha$  è una e.r. di alfabeto terminale e non. La parte destra  $\alpha$  definisce dunque un linguaggio regolare, di cui è facile costruire l'automa finito riconoscitore,  $M_A$ .

Nel caso semplice in cui  $\alpha$  contenga soltanto simboli terminali, l'automa  $M_A$  riconosce il linguaggio  $L_A(G)$  generato dalla grammatica partendo dal nonterminale  $A$ . Ma in generale in  $\alpha$  compaiono anche simboli nonterminali. Una transizione dell'automa etichettata con un nonterminale  $B$  è da pensare come l'invocazione d'un altro automa, quello associato alla regola  $B \rightarrow \beta$ . Ma  $B$  può anche coincidere con  $A$ , nel qual caso l'invocazione è ricorsiva.

Per evitare confusioni si diranno macchine "prova" gli automi finiti, riservando il termine "automa" a quello a pila che riconosce il linguaggio  $L(G)$ .

**Definizione 4.18. Rete ricorsiva di macchine finite**

*Le macchine che qui interessano sono deterministiche.*

NR

- Siano  $\Sigma$  l'alfabeto terminale,  $V = \{S, A, B, \dots\}$  i nonterminali,  $S$  l'assio-  
ma della grammatica libera estesa  $G$ .
- Per ogni nonterminale  $A$  vi è una sola regola  $A \rightarrow \alpha$ , la cui parte destra  $\alpha$  è una e.r. di alfabeto  $\Sigma \cup V$ .
- Siano  $S \rightarrow \sigma, A \rightarrow \alpha, B \rightarrow \beta, \dots$  le regole della grammatica.  
*Si indicano con  $R_S, R_A, R_B, \dots$  i linguaggi regolari, di alfabeto  $\Sigma \cup V$ , definiti dalle e.r.  $\sigma, \alpha, \beta, \dots$*
- $M_S, M_A, M_B, \dots$  sono i nomi delle macchine finite deterministiche che accettano i linguaggi regolari  $R_S, R_A, \dots$  L'insieme delle macchine, ossia la rete, è indicata da  $\mathcal{M}$ .
- Per chiarezza, conviene supporre che i nomi degli stati interni delle macchine siano tutti distinti. L'insieme degli stati della macchina  $M_A$  è  $Q_A$ , lo stato iniziale è  $q_{A,0}$  e gli stati finali sono  $F_A$ . La funzione di transizione di ogni macchina può essere indicata con lo stesso nome  $\delta$  senza rischio di confusione, visto che gli insiemi degli stati sono disgiunti.
- Per uno stato  $q$  della macchina  $M_A$ , si indica con  $R(M_A, q)$  (o con  $R(q)$  se la macchina è sottintesa) il linguaggio regolare, di alfabeto  $\Sigma \cup V$ , accettato dalla macchina partendo dallo stato  $q$ . Nel caso  $q$  sia lo stato iniziale, risulta  $R(M_A, q_{A,0}) \equiv R_A$ .

Una regola  $A \rightarrow \alpha$  è perfettamente rappresentata dal grafo della macchina  $M_A$ , essendovi corrispondenza biunivoca tra le stringhe definite da  $\alpha$  e i cammini del grafo conducenti dallo stato iniziale a uno stato finale.

Poiché l'insieme delle macchine  $\mathcal{M} = \{M_S, M_A, \dots\}$  è una pura variante notazionale, i concetti studiati per le grammatiche continuano a valere, in particolare quello di derivazione. Così il linguaggio terminale (di alfabeto  $\Sigma$ )  $L(\mathcal{M})$  definito (o riconosciuto) dalla rete di macchine coincide con quello generato dalla grammatica,  $L(G)$ .

Inoltre servirà il linguaggio terminale definito da una macchina generica  $M_A$ , partendo da uno stato di essa anche diverso da quello iniziale. Per uno stato  $q$  della macchina  $M_A$  si pone

$$L(M_A, q) = \{y \in \Sigma^* \mid \eta \in R(M_A, q) \wedge \eta \xrightarrow{*} y\}$$

Se la macchina è sottintesa basta scrivere  $L(q)$ . La formula considera ogni stringa  $\eta$ , contenente terminali o nonterminali, accettata dalla macchina  $M_A$  partendo dallo stato  $q$ . Le derivazioni che originano da  $\eta$  producono le stringhe terminali appartenenti al linguaggio  $L(q)$ .

In particolare per quanto detto risulta:

$$L(M_A, q_{A,0}) \equiv L(q_{A,0}) \equiv L_A(G)$$

Per l'assioma si ha:

$$L(M_S, q_{S,0}) \equiv L(q_{S,0}) \equiv L(\mathcal{M}) \equiv L(G)$$

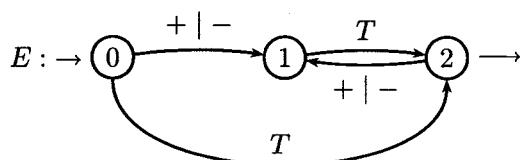
*Esempio 4.19.* Rete di macchine per le espressioni aritmetiche.

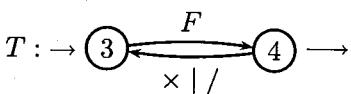
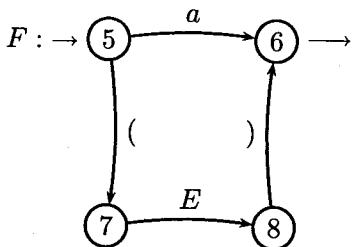
La grammatica contiene tre nonterminali ( $E$  è l'assioma), dunque tre regole:

$$E \rightarrow [+ | -]T \quad ((+ | -)T)^* \quad T \rightarrow F \quad ((\times | /)F)^* \quad F \rightarrow (a \mid (' E'))$$

La prossima figura mostra le macchine della rete  $\mathcal{M}$ .

$M_E$  :



$M_T$ : $M_F$ :

Risulta:

$$R(M_E, 0) = R(M_E) = [+ | -]T ((+ | -)T)^*$$

$$R(M_T, 4) = R(4) = ((\times | /)F)^+$$

$$L(M_E, 0) = L(0) = L_E(G) = L(G) = L(\mathcal{M}) = \{a, a + a, a \times (a - a), \dots\}$$

$$L(M_F, 5) = L(5) = L_F(G) = \{a, (a), (\neg a + a), \dots\}$$

$$L(M_F, 8) = L(8) = \{'\}'\}$$

### Diagrammi sintattici

Si apre un intermezzo sull'impiego delle reti di macchine nella documentazione dei linguaggi tecnici. Come in altri settori della tecnica, anche qui certi schemi grafici sono impiegati per documentare il progetto, in sostituzione o accanto alle definizioni testuali (espressioni regolari o regole grammaticali).

I *diagrammi sintattici* delle grammatiche libere, anche estese con espressioni regolari, sono una documentazione grafica molto comune nei manuali dei linguaggi tecnici; nonché, sotto il nome di reti di transizioni, nella linguistica computazionale, per il trattamento automatico del linguaggio naturale.

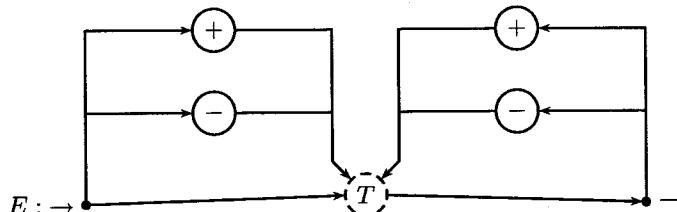
La diffusione dei diagrammi sintattici deriva dalla loro facile leggibilità e dal fatto che forniscono allo stesso tempo la definizione strutturale del linguaggio e gli schemi di flusso delle procedure del parsificatore, come si vedrà nel seguente.

I diagrammi sintattici differiscono dalle reti di macchine soltanto nelle convenzioni grafiche. Primariamente il grafo dell'automa finito è trasformato nel grafo duale, in cui archi e nodi sono scambiati. I nodi del grafo sono i simboli, terminali e non, della parte destra della regola grammaticale  $A \rightarrow \alpha$ . Gli archi non sono etichettati; ossia gli stati non sono contraddistinti dal nome, ma solo dalla posizione nel grafo. Il diagramma ha convenzionalmente un solo punto di entrata, evidenziato dalla freccia etichettata con il nome del nonterminale  $A$ , e un solo punto di uscita, corrispondente allo stato finale.

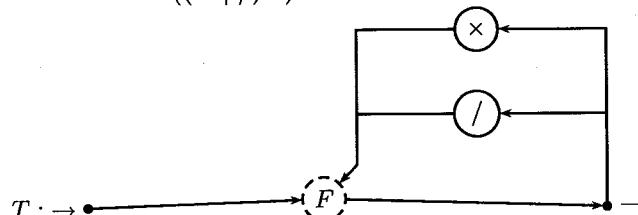
Un esempio è sufficiente per presentare i diagrammi sintattici.

*Esempio 4.20.* Diagrammi sintattici di espressioni aritmetiche (es. 4.19). Le macchine della rete di p. 171 si trasformano nei grafi tipici dei diagrammi sintattici: i nodi del grafo sono i simboli della grammatica, tratteggiati i nonterminali, a tratto continuo i terminali;<sup>10</sup> gli stati dell'automa non sono disegnati come nodi né portano etichetta.

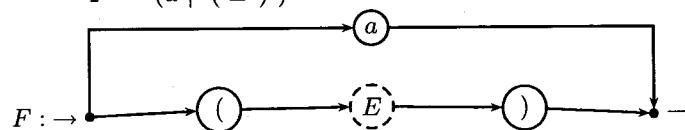
$$E \rightarrow [+ | -]T ((+ | -)T)^*$$



$$T \rightarrow F((\times | /)F)^*$$



$$F \rightarrow (a | ' (E')')$$



Attraversando un grafo dall'entrata all'uscita, si ottengono le possibili parti destre della regola sintattica, ad es. nel primo diagramma:

<sup>10</sup>Anche altri stili grafici possono essere impiegati per distinguere visivamente le due classi di simboli.

$$T, \quad +T, \quad T + T, \quad \dots$$

Data una grammatica libera estesa, per ottenere i diagrammi sintattici che la rappresentano, basta costruire, con uno dei metodi noti, i riconoscitori finiti delle e.r. delle parti destre delle regole, e aggiustarli alla convenzione grafica adottata dai diagrammi sintattici.

### 4.3.3 Procedura ricorsiva indeterministica di riconoscimento

Si mostra che è immediato interpretare una rete di macchine ricorsive come lo schema di flusso dell'algoritmo indeterministico di riconoscimento delle frasi del linguaggio.

Si può pensare che ciascuna macchina sia un sottoprogramma e che il salto da una macchina all'altra sia l'invocazione d'un sottoprogramma, al cui termine avviene il ritorno alla macchina chiamante. Poiché le macchine della rete possono invocarsi ricorsivamente, questo tipo di algoritmo è detto a *discesa ricorsiva*.

È essenziale dire che, affinché la ricorsione termini, la grammatica non deve essere ricorsiva a sinistra. In caso contrario, se vi fosse la regola s-ricorsiva  $A \rightarrow A\gamma$ , il sottoprogramma associato alla macchina  $M_A$  invocherebbe se stesso senza fine.

L'algoritmo realizza un automa a pila, in generale non deterministico, con un solo stato interno, che accetta a pila vuota.

Si descrive dapprima a parole il suo funzionamento. Alla partenza, la macchina attiva è quella dell'assioma e il carattere corrente il primo carattere del testo. Se la stringa è valida, esisterà un calcolo che, dopo aver eventualmente invocato altre macchine, condurrà tale macchina nello stato finale.

L'automa a ogni passo può proseguire il calcolo all'interno della *macchina attiva*, proprio come farebbe un riconoscitore finito, leggendo il prossimo carattere terminale, e cambiando stato. Il cambio di stato è registrato scrivendo nella pila il nuovo stato al posto del vecchio, sicché la cima contiene sempre lo stato corrente della macchina attiva.

La macchina, se dallo stato corrente esce un arco non terminale  $A$ , può saltare spontaneamente allo stato iniziale della macchina chiamata  $M_A$  (la stessa macchina attiva se la regola è ricorsiva), depositando sulla pila tale stato sopra quello della macchina sospesa.

Quando un calcolo raggiunge lo stato finale d'una macchina, si dice che essa è terminata. L'automa deve ritornare alla macchina sospesa, anche detta chiamante. L'operazione di ritorno comprende due passi: prima cancella lo stato della macchina attiva, e così ripristina lo stato dell'ultima macchina sospesa, che torna a essere attiva; poi esegue la mossa etichettata con il nome della macchina terminata.

*Algoritmo 4.21.* Riconoscitore a discesa ricorsiva non deterministico.

L'automa a pila è così definito:

*begin*

1. la stringa sorgente è  $x$  e  $cc$  il carattere terminale corrente;
2. i simboli della pila sono l'unione (disgiunta)  $Q = Q_A \cup Q_B \cup \dots$  degli stati di tutte le macchine;
3. l'automa a pila ha un solo stato interno che si lascia sottinteso;
4. inizialmente la pila contiene lo stato iniziale  $q_{S,0}$ ;
5. transizioni; sia  $s \in Q_A$  il simbolo in cima, cioè lo stato della macchina attiva  $A$ ;
  - a) (mossa di scansione)
 

se la mossa  $s \xrightarrow{cc} s'$  è definita (nella macchina attiva  $M_A$ ), consuma il carattere corrente e scrivi nella pila  $s'$  al posto di  $s$ ;
  - b) (mossa di chiamata)
 

se la mossa  $s \xrightarrow{A} s'$  è definita per un nonterminale (generico)  $A$ , effettua una mossa spontanea che depone sulla pila lo stato iniziale  $q_{A,0}$  della macchina  $M_A$ , che così diviene quella attiva;
  - c) (mossa di ritorno)
 

se  $s$  è uno stato finale d'una macchina generica  $M_A$ , cancellalo dalla pila, effettua dallo stato  $r$  emerso sulla cima la mossa  $r \xrightarrow{A} s'$  e scrivi nella pila  $s'$  al posto di  $r$ ;
  - d) (mossa di riconoscimento)
 

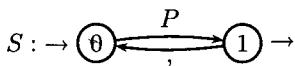
se  $s$  è uno stato finale di  $M_S$  (la macchina dell'assioma) e  $cc = \perp$ , accetta e termina;
  - e) in ogni altro caso rifiuta la stringa e termina.

*end*

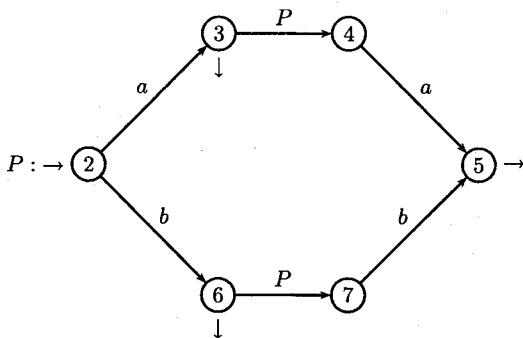
*Esempio 4.22.* Lista di palindromi dispari.

La rete ha due macchine:

$$S \rightarrow P(, P)^*$$



$$P \rightarrow aPa \mid bPb \mid a \mid b$$



Partendo dallo stato 0 si ha il linguaggio  $L(0) \equiv L(G)$  delle liste di palindromi di lunghezza dispari. Dallo stato 2 si ha invece il linguaggio  $L(2)$  dei palindromi dispari.

L'analisi della stringa  $a, b \dashv$  inizia con lo stato 0 in cima alla pila. La macchina  $M_S$  pone sulla pila 2, ossia chiama la macchina  $M_P$ , la quale, leggendo  $a$ , scrive sulla pila 3 al posto di 2. Poi  $M_P$  può terminare, oppure chiamare ricorsivamente se stessa, scelta quest'ultima che fallirebbe. Se  $M_P$  decide di terminare, toglie 3 dalla pila, e il calcolo riprende in 0. La macchina  $M_S$  impila 1 al posto di 0, poi, leggendo la virgola, impila 0 al posto di 1, e di nuovo chiama  $M_P$ , e così via. Il calcolo termina nello stato 1, leggendo il carattere terminatore.

La prossima sezione svilupperà, sotto opportune condizioni, una versione deterministica dell'algoritmo.

#### 4.4 Analisi sintattica discendente deterministica

Molte delle grammatiche dei linguaggi tecnici sono state progettate apposta per permettere un veloce riconoscimento deterministico delle frasi. La pros-

sima strada, detta  $LL(k)$ <sup>11</sup>, per costruire gli automi a pila deterministici a discesa ricorsiva, è intuitiva e consente grande flessibilità nell'implementazione dei traduttori guidati dalla sintassi.

Si vedranno due varianti implementative: l'automa a pila classico, e il programma a discesa ricorsiva, realizzato mediante la pila delle aree di attivazione dei sottoprogrammi.

Infine seguiranno alcune trasformazioni utili per adattare una grammatica a questo tipo di analisi.

#### 4.4.1 Condizioni per la costruzione del riconoscitore $LL(1)$

Sia data una grammatica  $G$ , sotto forma di rete ricorsiva di macchine, ossia ogni regola  $A \rightarrow \alpha$  è specificata dalla macchina finita deterministica  $M_A$ , che riconosce il linguaggio regolare definito dalla e.r.  $\alpha$ .

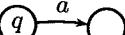
Si ricorda che un linguaggio è detto *annullabile* se contiene la stringa vuota. Riprendendo concetti simili a quelli introdotti per gli automi a stati finiti (algoritmi GMY 3.8.2 e BS 3.8.3 p. 129 e seguenti) si danno le seguenti definizioni:

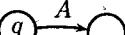
*Insieme degli inizi d'uno stato  $q$ :* per un generico stato  $q$  si prendono gli inizi delle stringhe riconosciute partendo da tale stato  $Ini(q) = Ini(L(q))$ .

*Insieme dei seguiti:* L'insieme dei seguiti  $Seg(A)$  d'un nonterminale  $A$  contiene i caratteri terminali che possono seguire  $A$  in qualche derivazione. Inoltre il terminatore  $\dashv$  appartiene ai seguiti dell'assioma.

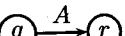
Il calcolo degli inizi e dei seguiti è presentato dai seguenti algoritmi sotto forma di clausole logiche. Siano  $a$  un terminale,  $A, B$  dei nonterminali e  $q, r$  degli stati.

*Algoritmo 4.23.* Insieme degli inizi del linguaggio  $L(q)$ .

1.  $a \in Ini(q)$  se  $\exists$  arco 

2.  $a \in Ini(q)$  se  $\exists$  arco 

$\wedge a \in Ini(q_{A,0})$ , dove  $q_{A,0}$  è lo stato iniziale della macchina  $M_A$ .

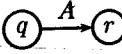
3.  $a \in Ini(q)$  se  $\exists$  arco   
 $\wedge L(q_{A,0})$  è annullabile  $\wedge a \in Ini(r)$

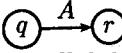
I casi 1. e 2. si spiegano da soli. Nel caso 3. poiché  $\varepsilon \in L(q_{A,0})$ , dallo stato  $q$  si può andare in  $r$  senza leggere alcun carattere; allora il primo carattere incontrato sarà quello che si legge a partire da  $r$ .

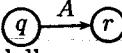
<sup>11</sup>La sigla  $LL(1)$  ha un significato storico. La prima elle indica che l'analisi esamina il testo da sinistra (left) a destra, la seconda elle indica che la derivazione è sinistra (leftmost) e il numero  $k$  precisa la lunghezza in caratteri della prospezione.

*Algoritmo 4.24.* Insieme dei seguiti d'un nonterminale  $A$ .

→ 1.  $\dashv \in Seg(S)$

2.  $a \in Seg(A)$  se  $\exists$  arco   $\wedge a \in Ini(r)$

3.  $a \in Seg(A)$  se  $\exists$  arco  nella macchina  $M_B, B \neq A$   
 $\wedge$  il linguaggio  $L(r)$  è annullabile  
 $\wedge a \in Seg(B)$

4.  $a \in Seg(A)$  se  $\exists$  arco  →  
 $\wedge r$  è uno stato finale della macchina  $M_B$ , con  $B \neq A$   
 $\wedge a \in Seg(B)$

Il caso 1. dice che l'assioma, poiché deriva una frase completa, è seguito dal terminatore.

Il caso 2. aggiunge ai seguiti di  $A$  i caratteri iniziali del linguaggio riconosciuto partendo dallo stato destinazione d'una freccia etichettata  $A$ .

Nel caso 4., letta una stringa del linguaggio  $L_A(G)$ , l'automa si trova nello stato finale della macchina  $M_B$ , quindi ha terminato la lettura d'una stringa del linguaggio  $L_B(G)$ . Perciò il carattere seguente appartiene ai seguiti di  $B$  e anche a quelli di  $A$ .

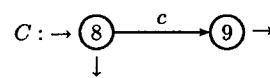
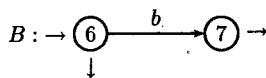
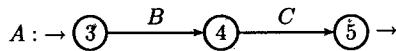
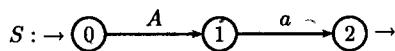
Il caso 3. generalizza il caso 4., nel senso che lo stato  $r$  non è direttamente finale, ma è collegato a uno stato finale della macchina  $M_B$ , tramite un cammino che può leggere la stringa vuota.

Per eseguire il calcolo degli inizi e dei seguiti, si inizializzano tutti gli insiemi degli inizi e dei seguiti a vuoto. Poi si applicano le clausole degli algoritmi ripetutamente e in qualsiasi ordine, finché gli insiemi degli inizi e dei seguiti non crescono più (raggiungimento d'un punto fisso).

*Esempio 4.25.* Inizi e seguiti.

Queste situazioni sono illustrate per la grammatica.

$$S \rightarrow Aa \quad A \rightarrow BC \quad B \rightarrow b \mid \epsilon \quad C \rightarrow c \mid \epsilon$$



I linguaggi generati da  $A, B, C$  sono annullabili. Per l'insieme degli inizi si ha:

$$\begin{aligned} Ini(0) &= Ini(3) \cup Ini(1) \\ &= Ini(6) \cup Ini(4) \cup Ini(1) \\ &= Ini(6) \cup Ini(8) \cup Ini(5) \cup Ini(1) \\ &= \{b\} \cup \{c\} \cup \emptyset \cup \{a\} \end{aligned}$$

Gli insiemi dei seguiti sono:

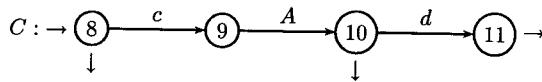
$$Seg(S) = \{\dashv\}$$

$$\begin{aligned} Seg(A) &= Ini(1) \\ &= \{a\} \end{aligned}$$

$$\begin{aligned} Seg(B) &= Ini(4) \cup Seg(A) \\ &= Ini(8) \cup Seg(A) \\ &= \{c\} \cup \{a\} \end{aligned}$$

$$\begin{aligned} Seg(C) &= Seg(A) \\ &= \{a\} \end{aligned}$$

Se nella rete vi sono più comparsate d'un nonterminale, i seguiti di ogni comparsa vanno uniti. Per illustrarlo, si modifichi la macchina  $M_C$  della rete:



Si ricalcolano i seguiti di  $A$ :

$$\begin{aligned} Seg(A) &= Ini(1) \cup Ini(10) \cup Seg(C) \\ &= \{a\} \cup \{d\} \cup Seg(A) \\ &= \{a\} \cup \{d\} \end{aligned}$$

Si noti che all'ultimo passaggio il termine  $Seg(A)$ , identico alla parte sinistra, è eliminato.

### Insieme guida

L'insieme guida prossimamente definito serve da selezionatore per la scelta della mossa negli stati dove si presenta un bivio. L'idea è di calcolare per ogni ramo d'un bivio l'insieme dei primi caratteri che potranno essere incontrati prendendo quella strada. Se tali insiemi guida sono disgiunti per gli archi del bivio, la scelta della mossa è univoca: quella il cui insieme guida contiene il carattere corrente.

**Definizione 4.26.** Insieme guida (*o di prospezione*).

L'insieme guida  $Gui(q \rightarrow \dots) \subseteq \Sigma \cup \{\dashv\}$  è definito per ogni freccia (di mossa o di accettazione) della rete. A seconda del tipo di freccia e della sua etichetta, la definizione dell'insieme si divide nei seguenti casi.

1. Per un arco  $q \xrightarrow{b} r$ , con etichetta terminale  $b \in \Sigma$ , si ha:

$$Gui(q \xrightarrow{b} r) = \{b\}$$

2. Per un arco  $q \xrightarrow{A} r$  della macchina  $M_B$ , etichettato dal nonterminale  $A$ , si ha:

a)  $Gui(q \xrightarrow{A} r) = Ini(L(q_{A,0}) L(r))$ , se il concatenamento  $L(q_{A,0}) L(r)$  non è annullabile;

b)  $Gui(q \xrightarrow{A} r) = Ini(L(q_{A,0}) L(r)) \cup Seg(B)$ , altrimenti.

3. Per la freccia  $q \rightarrow$  d'uno stato finale  $q$  della macchina  $M_B$  si ha:

$$Gui(q \rightarrow) = Seg(B)$$

Il caso 1. è ovvio: il primo carattere è l'etichetta stessa dell'arco.

In 2.a il carattere che fa scegliere la mossa etichettata dal nonterminale  $A$ , è un carattere iniziale del linguaggio definito dal nonterminale; ma se esso è annullabile si prende anche un carattere iniziale del linguaggio riconosciuto a partire dallo stato  $r$ .

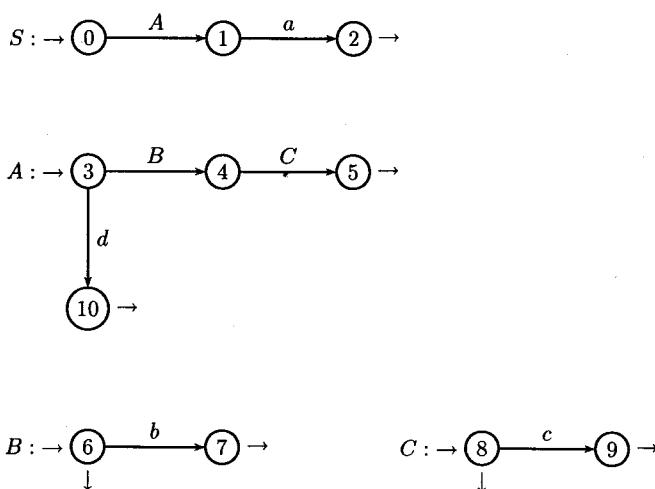
Il caso più complesso è 2.b. Poiché il concatenamento dei linguaggi  $L(q_{A,0})L(r)$  è annullabile, il riconoscitore può raggiungere uno stato finale della macchina  $M_B$  senza incontrare alcun carattere. In tale caso, il primo carattere incontrato sarà uno di quelli che possono seguire il nonterminale  $B$ .

Nel caso 3. il calcolo sulla macchina  $M_B$  può terminare quando il carattere corrente sta nei seguiti di  $B$ .

*Esempio 4.27.* Casi dell'insieme guida.

Queste situazioni sono illustrate dalla seguente grammatica.

$$S \rightarrow Aa \quad A \rightarrow BC \mid d \quad B \rightarrow b \mid \varepsilon \quad C \rightarrow c \mid \varepsilon$$



Lo stato 3 della macchina  $M_A$  è un bivio, dove, essendo il concatenamento

$$L(6)L(8) = L(6)L(4) = \{\varepsilon, b\}\{\varepsilon, c\}$$

annullabile, risulta:

$$\begin{aligned} Gui(3 \xrightarrow{B} 4) &= Ini(L(6)L(8)) \cup Seg(A) \\ &= Ini(\{\varepsilon, b, c, bc\}) \cup \{a\} \\ &= \{b, c, a\} \end{aligned}$$

$$Gui(3 \xrightarrow{d} 10) = \{d\}$$

Nel bivio 6 si ha:

$$\begin{aligned} Gui(6 \rightarrow) &= Seg(B) \\ &= \{a, c\} \\ Gui(6 \xrightarrow{b} 7) &= \{b\} \end{aligned}$$

L'ultimo bivio da considerare è 8, in cui risulta:

$$\begin{aligned} Gui(8 \rightarrow) &= Seg(C) \\ &= Seg(A) \\ &= \{a\} \\ Gui(8 \xrightarrow{c} 9) &= \{c\} \end{aligned}$$

Poiché in ogni bivio gli insiemi guida delle frecce uscenti sono disgiunti, la grammatica soddisfa la condizione  $LL(1)$ .

Si completa l'esposizione con una condizione sufficiente, affinché l'automa a pila dell'Algoritmo 4.21 diventi deterministico.

#### **Definizione 4.28. Condizione $LL(1)$**

*Una macchina  $M_A$  della rete soddisfa la condizione  $LL(1)$  nello stato  $q$  se, per ogni coppia di frecce uscenti dallo stato, gli insiemi guida sono disgiunti.*

*Una regola soddisfa la condizione  $LL(1)$  se ogni stato della macchina corrispondente la soddisfa.*

*Una grammatica gode della proprietà  $LL(1)$  se ogni stato delle macchine della rete soddisfa la condizione  $LL(1)$ .*

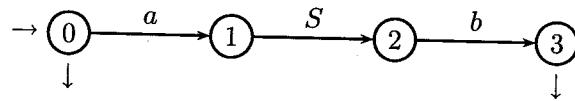
È facile osservare che, se uno stato fosse indeterministico e da esso partissero due frecce con la stessa etichetta, la condizione  $LL(1)$  sarebbe violata. Questa è la ragione che motiva l'utilizzo di macchine deterministiche nell'ambito della costruzione dei parsificatori  $LL(1)$ . Seguono vari esempi di calcolo degli insiemi guida e di verifica della condizione.

*Esempio 4.29.* Grammatiche  $LL(1)$  e non.

#### 1. La grammatica

$$G_1 : \quad S \rightarrow aSb \mid \epsilon$$

genera il linguaggio  $\{a^n b^n \mid n \geq 0\}$ . Disegnata la macchina deterministica corrispondente



La verifica della condizione  $LL(1)$  richiede il calcolo degli insiemi guida, soltanto dove almeno due frecce escono, cioè nel solo stato 0. Gli insiemi

$$Gui(0 \xrightarrow{a} 1) = \{a\}, \quad Gui(0 \rightarrow) = Seg(S) = \{b, \dashv\}$$

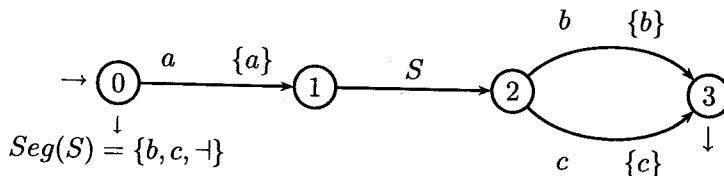
sono disgiunti e la grammatica è  $LL(1)$ .

Si noti che il secondo insieme contiene  $b$  perché  $S$  nello stato 2 è seguito da  $b$ ; e contiene il terminatore perché  $S$  è la macchina assioma.

### 2. La grammatica

$$G_2 : \quad S \rightarrow aSb \mid aSc \mid \epsilon$$

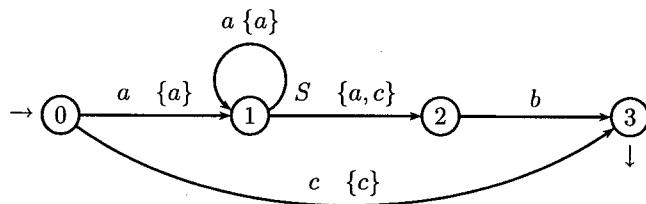
genera il linguaggio  $\{a^n(b \mid c)^n \mid n \geq 0\}$ . La macchina risulta  $LL(1)$  poiché negli stati bivii 0 e 2 gli insiemi guida delle frecce uscenti, tra graffe nel disegno, sono disgiunti.



### 3. La grammatica

$$G_3 : \quad S \rightarrow a^+ Sb \mid c$$

genera il linguaggio  $\{a^* a^n c b^n \mid n \geq 0\}$ . Il grafo della macchina contiene un circuito (autoanello) corrispondente all'operatore croce.

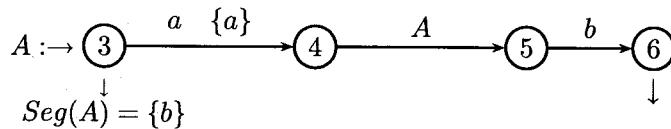
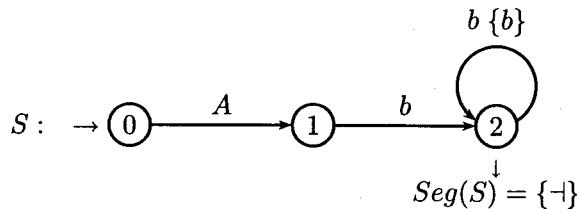


Lo stato 1 viola la condizione  $LL(1)$  poiché  $Ini(S) = \{a, c\}$  contiene  $a$ .  
Nel bivio 0 la condizione è invece verificata.

#### 4. La grammatica

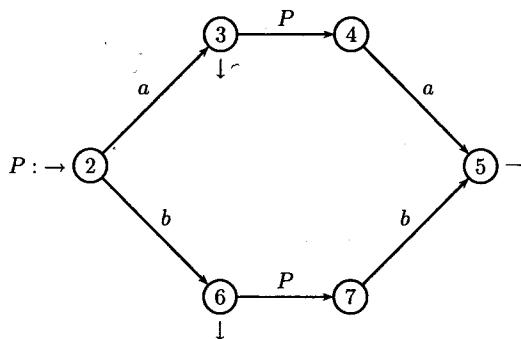
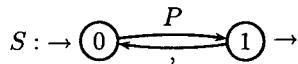
$$G_4 : \quad S \rightarrow Ab^+ \quad A \rightarrow aAb \mid \varepsilon$$

genera il linguaggio  $\{a^n b^n b^+ \mid n \geq 0\}$ .



Le macchine soddisfano la condizione nei loro bivii, 2 e 3.

#### 5. La grammatica delle liste di palindromi (es. 4.22 p. 175)



non è  $LL(1)$  nello stato 3, dove la freccia d'uscita ha l'insieme  $Gui(3 \rightarrow) = Seg(P) = \{\cdot\} \cup \{a\} \cup \{b\}$ , calcolato osservando le tre comparse di  $P$  nella rete. Ma l'insieme  $Gui(3 \xrightarrow{P} 4) = Ini(L_P(G)) = Ini(2) = \{a, b\}$  è sovrapposto. Lo stesso conflitto avviene nello stato 6.

Invece il bivio 1 della prima macchina soddisfa la condizione, poiché la virgola non appartiene all'insieme  $Gui(1 \rightarrow) = Seg(S) = \{\cdot\}$ .

#### Calcolo semplificato degli insiemi guida

Il calcolo degli insiemi guida, se la grammatica non è estesa con espressioni regolari, può essere direttamente condotto sulle regole stesse, senza la necessità di disegnare le macchine corrispondenti.

La definizione si semplifica nel seguente modo. Sia  $A \rightarrow \alpha$  una regola della grammatica  $G$  di assioma  $S$ .

$$\begin{cases} Gui(A \rightarrow \alpha) = Ini(\alpha), & \text{se } \alpha \text{ non è annullabile;} \\ Gui(A \rightarrow \alpha) = Ini(\alpha) \cup Seg(A), & \text{altrimenti.} \end{cases}$$

I termini che compaiono possono essere definiti, invece che sulle macchine, direttamente sulla grammatica:

$$Ini(\alpha) = \{a \in \Sigma \mid \alpha \xrightarrow{*} ay \wedge y \in \Sigma^*\}$$

$$Seg(A) = \{a \in \Sigma \cup \{\cdot\} \mid S \dashv \xrightarrow{*} yAaz \wedge y, z \in \Sigma^*\}$$

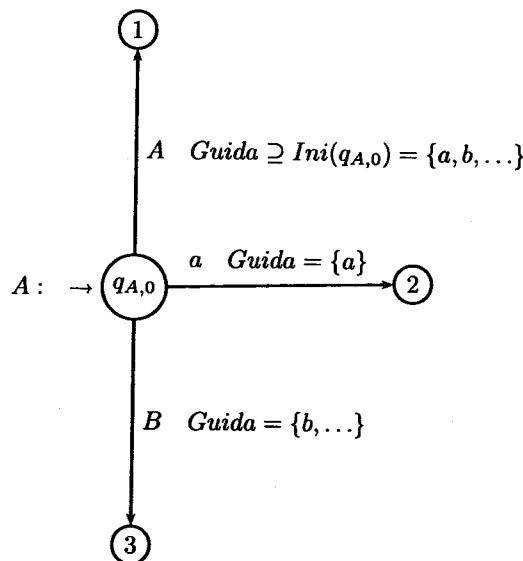
La condizione  $LL(1)$  dunque impone che gli insiemi guida siano disgiunti, per ogni coppia di regole alternative  $A \rightarrow \alpha, A \rightarrow \alpha'$  della grammatica.

*Ricorsione a sinistra*

Dalla definizione segue facilmente la seguente proprietà, che esclude dall'analisi discendente le grammatiche aventi regole ricorsive a sinistra.

*Proprietà 4.30.* Una regola ricorsiva a sinistra viola la condizione  $LL(1)$ .

*Dimostrazione.* Per semplicità si esamina una regola con ricorsione soltanto immediata,  $A \rightarrow A \dots | a \dots | B \dots$  sotto schematizzata, ma lo stesso ragionamento vale nel caso d'una derivazione ricorsiva di lunghezza maggiore di uno.



Il calcolo dell'insieme guida del primo arco (dall'alto) porta a rientrare ricorsivamente nella macchina stessa, così ottenendo come caratteri iniziali quelli stessi che stanno negli insiemi guida del secondo e terzo arco. Di conseguenza l'insieme guida dell'arco ricorsivo a sinistra contiene l'unione degli altri insiemi guida.

Di conseguenza, le ricorsioni sinistre presenti in una grammatica dovranno essere trasformate in destre, se si vuole applicare l'analisi  $LL(1)$ .

*Riconoscitore  $LL(1)$* 

Il punto d'arrivo del percorso concettuale precedente è il prossimo enunciato

*Proprietà 4.31.* Se la grammatica gode della proprietà  $LL(1)$ , l'algoritmo 4.21 (p. 175) di riconoscimento delle stringhe diviene deterministico.

Si guardi la riformulazione dell'algoritmo, sotto la condizione  $LL(1)$ .

*Algoritmo 4.32.* Riconoscitore a discesa ricorsiva deterministico

L'automa a pila è così definito:

*begin*

1. la stringa sorgente è  $x$  e  $cc$  il carattere terminale corrente;
2. i simboli della pila sono l'unione (disgiunta)  $Q = Q_A \cup Q_B \cup \dots$  degli stati di tutte le macchine;
3. l'automa a pila ha un solo stato interno, che si lascia sottinteso;
4. inizialmente la pila contiene lo stato iniziale  $q_{S,0}$ ;
5. transizioni; sia  $s \in Q_A$  il simbolo in cima, cioè lo stato della macchina attiva  $M_A$ ;
  - a) (mossa di scansione)
 

se la mossa  $s \xrightarrow{cc} s'$  è definita (nella macchina attiva  $M_A$ ), consuma il carattere corrente e scrivi nella pila  $s'$  al posto di  $s$ ;
  - b) (mossa di chiamata)
 

se la mossa  $s \xrightarrow{A} s'$  è definita per un nonterminale  $A$ , e  $cc \in Gui(s \xrightarrow{A} s')$  effettua la mossa spontanea che depone sulla pila lo stato iniziale  $q_{A,0}$  della macchina  $M_A$ , che così diviene quella attiva;
  - c) (mossa di ritorno)
 

se  $s$  è uno stato finale d'una macchina generica  $M_A$  e  $cc \in Gui(s \rightarrow)$  cancellalo, effettua dallo stato  $r$  emerso sulla cima la mossa  $r \xrightarrow{A} s'$  spontanea e scrivi nella pila  $s'$  al posto di  $r$ ;
  - d) (mossa di riconoscimento)
 

se  $s$  è uno stato finale di  $M_S$  (macchina dell'assioma) e  $cc = -l$ , accetta e termina;
  - e) in ogni altro caso rifiuta la stringa e termina.

*end*

Poiché gli insiemi guida delle frecce uscenti dallo stato  $s$  sono per l'ipotesi  $LL(1)$  disgiunti, al passo 5. le condizioni a), b), c) e d) sono mutuamente esclusive e la scelta dell'automa è deterministica.

La complessità di calcolo è lineare nella lunghezza  $n$  della stringa sorgente. Infatti o l'automa legge e consuma un carattere sorgente, o effettua una mossa spontanea nei casi b) e c). Ma il numero di mosse spontanee, comprese tra due letture successive, è limitato superiormente da una costante, grazie al ragionamento seguente. Una mossa spontanea invoca una macchina, che a sua volta può immediatamente (ossia senza leggere un carattere) invocare un'altra macchina, e così via. Poiché la grammatica non può contenere regole (anche mediamente) ricorsive a sinistra, la lunghezza della catena delle invocazioni è limitata superiormente dal numero dei nonterminali della grammatica. In definitiva il numero di mosse spontanee tra due letture non può superare una costante, indipendente dalla lunghezza della stringa sorgente, quindi la complessità dell'algoritmo è  $O(n)$ .  
Si noti poi che il numero di passi diviene esattamente eguale alla lunghezza

della stringa quando la grammatica  $LL(1)$  è nella forma normale di Greibach, che per questa ragione è anche detta in tempo reale.

### *Implementazione del parsificatore a procedure ricorsive*

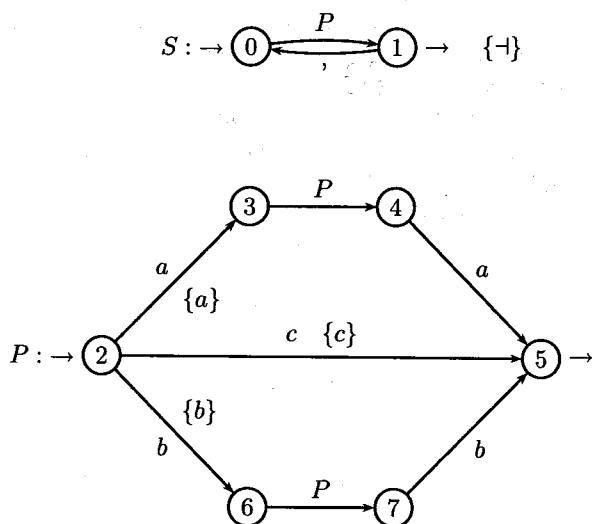
Si illustra con un esempio la realizzazione del parsificatore come insieme di procedure corrispondenti alle macchine della grammatica.

*Esempio 4.33.* Procedure ricorsive per lista di palindromi con centro.

La seguente grammatica

$$S \rightarrow P(, P)^* \quad P \rightarrow aPa \mid bPb \mid c$$

e rete di macchine ha la proprietà  $LL(1)$ , come testimoniato dagli insiemi guida riportati nel disegno:



Il programma consiste di due procedure, in corrispondenza con i simboli non-terminali della grammatica.

In sostanza ogni procedura ha lo schema a blocchi che corrisponde alla macchina associata a quel nonterminale.

La procedura, esaminato il carattere corrente, decide, in base alla sua appartenenza a un insieme guida, quale freccia del grafo seguire.

Se la freccia ha etichetta terminale, il prossimo carattere corrente è calcolato dalla funzione Prossimo (cioè dall'analizzatore lessicale o scansore). Se la

```

procedure S
begin
  1. call P;
  2. if cc='-' then accetta e termina;
  3. else if cc='.' then cc := Prossimo; go to 1;
  4. else Errore;
end

```

```

procedure P
begin
  1. if cc='a' then
    begin
      cc:=Prossimo; call P; if cc='a' then
      cc:=Prossimo else Errore
    end
  2. else if cc='b' then
    begin
      cc:=Prossimo; call P; if cc='b' then
      cc:=Prossimo else Errore
    end
  3. else if cc='c' then
    begin
      cc:=Prossimo
    end
  4. else Errore
end

```

freccia è nonterminale, la corrispondente procedura è invocata. Infine se il carattere non sta in nessun insieme guida, si ha un errore, e la stringa sorgente è rifiutata.

Inizialmente si lancia la procedura dell'assioma *S*, sul primo carattere della stringa.

Vari miglioramenti di stile e di efficienza si possono immaginare nella programmazione delle procedure. Ad es. nella prima regola l'iterazione mediante la stella, potrebbe essere codificata direttamente in un ciclo iterativo *while ... do*.

#### *Trattamento degli errori*

In caso di errore, questo semplice parsificatore termina subito, senza fornire spiegazioni, mentre nella pratica è di solito richiesto un messaggio diagnostico. Non è difficile generare automaticamente una semplice diagnostica, confrontando i caratteri attesi in un certo stato (quelli appartenenti agli insiemi guida delle frecce uscenti) con il carattere corrente, e scrivendo: "i caratteri attesi sono ..., invece di ...".

Inoltre, al fine di anticipare la scoperta d'un errore, il parsificatore può eseguire il test dell'insieme guida non soltanto nei bivii, ma anche in quegli stati

da cui esce una sola freccia, se essa ha etichetta nonterminale. Se infatti il carattere corrente non sta nell'insieme guida, la stringa è errata. Si noti che l'errore sarebbe comunque rilevato più avanti.

Infine un buon analizzatore deve essere capace di proseguire il calcolo dopo il primo errore, al fine di esaminare l'intera stringa sorgente con una sola compilazione, anche se essa contiene degli errori.<sup>12</sup>

#### 4.4.2 Come ottenere grammatiche $LL(1)$

Posti di fronte a una grammatica, per evitare l'inutile calcolo degli insiemi guida, conviene verificare che essa non sia ambigua, e che non presenti ricorsioni sinistre.

In caso contrario, occorre rimuovere l'ambiguità (p. 49), e poi trasformare le ricorsioni sinistre in destre (p. 64).

Dopo tali trasformazioni si calcolano gli insiemi guida, e se la grammatica non è  $LL(1)$ , si analizzano meglio le cause della violazione. A tale scopo, è utile un modo più sintetico di definire la condizione  $LL(1)$ . Per maggior generalità, si dà la definizione con valore parametrico della lunghezza di prospezione (anticipando p. 194).

*Proprietà 4.34.* Un nonterminale  $A$  della grammatica viola la condizione  $LL(k)$ ,  $k \geq 1$ , se esistono due derivazioni

$$S \xrightarrow{*} uAv \Rightarrow u\alpha v \xrightarrow{*} uzv \quad S \xrightarrow{*} uAv \Rightarrow u\alpha'v' \xrightarrow{*} uz'v'$$

dove  $A$  è nonterminale,  $\alpha, \alpha'$  sono stringhe di terminali o nonterminali, e le stringhe  $u, v, z, z', v'$  sono terminali anche vuote, con la seguente condizione: detto  $m = \min(k, |zv|, |z'v'|)$  il minimo tra  $k$  e le lunghezze delle stringhe  $zv$  e  $z'v'$ , i prefissi di lunghezza  $m$  di  $zv$  e di  $z'v'$  coincidono.

Nell'enunciato, la precisazione relativa alla lunghezza minima ha soltanto lo scopo di impedire che il prefisso sia più lungo della stringa da cui è estratto. Si prenda  $k = 1$ . Per convincersi che la formulazione equivale a quella precedente 4.28, si osservi che:

1. se esistessero due tali derivazioni, l'esame dei prossimi  $k = 1$  caratteri, ossia degli insiemi guida delle regole non basterebbe per la scelta tra le derivazioni  $A \rightarrow \alpha$  e  $A \rightarrow \alpha'$ ;
2. se gli insiemi guida, nello stato della macchina  $M_A$  in cui si presenta il bivio tra i rami  $\alpha$  e  $\alpha'$ , fossero disgiunti, allora i prefissi considerati nell'enunciato, non coinciderebbero.

---

<sup>12</sup>Per i metodi di trattamento degli errori si può vedere ad es. [23].

### Fattorizzazione sinistra

Il prossimo esempio mostra una violazione della condizione e il modo di correggerla.

*Esempio 4.35.* Fattorizzazione sinistra.

La grammatica

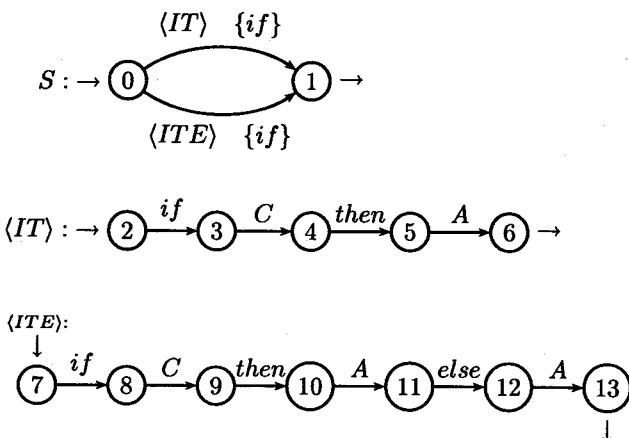
$$\begin{aligned} S &\rightarrow \langle IT \rangle \mid \langle ITE \rangle \\ \langle IT \rangle &\rightarrow if\ C\ then\ A & \langle ITE \rangle &\rightarrow if\ C\ then\ A\ else\ A \\ C &\rightarrow \dots & A &\rightarrow \dots \end{aligned}$$

schematizza le frasi condizionali con e senza il ramo *else*; i nonterminali *C* e *A*, lasciati incompleti, stanno per condizione booleana e istruzione d'assegnamento.

Per applicare la condizione precedente, si esaminino le derivazioni:

$$\begin{aligned} S &\Rightarrow \underbrace{\langle IT \rangle}_{A} \Rightarrow \underbrace{if\ C\ then\ A}_{\alpha} \stackrel{+}{\Rightarrow} \underbrace{if\ x \geq 3\ then\ x = 5}_{z} \\ S &\Rightarrow \underbrace{\langle ITE \rangle}_{A} \Rightarrow \underbrace{if\ C\ then\ A\ else\ A}_{\alpha'} \stackrel{+}{\Rightarrow} \underbrace{if\ x \geq 3\ then\ x = 5\ else\ x = 7}_{z'} \end{aligned}$$

Poiché i prefissi di lunghezza 1 (ossia gli inizi) delle stringhe *z* e *z'* sono eguali a *if*, il nonterminale *S* viola la condizione *LL(1)*. Alla stessa conclusione si giunge esaminando gli insiemi guida della prima macchina della rete:



Evidentemente gli insiemi guida degli archi uscenti dallo stato 0 coincidono. Anche aumentando la lunghezza, i prefissi di  $z$  e di  $z'$  restano eguali:

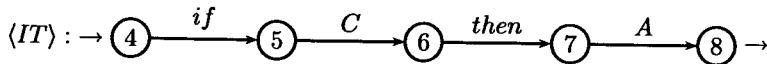
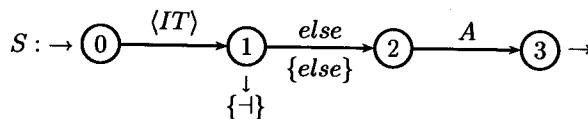
$$\text{if } x, \quad \text{if } x \geq, \quad \text{if } x \geq 3, \quad \text{ecc.}$$

Per differenziare i prefissi dell'esempio, si deve prendere il valore  $k = 11$ . Ma è evidente che, per ogni valore di  $k$  fissato, si può scegliere una stringa (condizione booleana) derivante da  $C$  e una stringa (assegnamento) derivante da  $A$ , tali da rendere uguali i prefissi di lunghezza  $k$ . Ciò significa che con questa grammatica, non si può costruire un parsificatore deterministico discendente, per quanto grande si prenda la lunghezza di prospezione.

La causa del problema è che il condizionale semplice è un prefisso del condizionale doppio. Per rimuoverla, basta ritardare la scelta tra i due costrutti condizionali, fino al momento in cui la presenza o l'assenza di *else* palesa quale sia il costrutto.

Ecco la grammatica dopo la trasformazione:

$$S \rightarrow \langle IT \rangle (\varepsilon \mid \text{else } A) \quad \langle IT \rangle \rightarrow \text{if } C \text{ then } A$$



Ora l'unico punto di scelta è nello stato 1, dove gli insiemi guida sono disgiunti.

La modifica illustrata è nota come *fattorizzazione sinistra* e spesso riesce a ottenere una grammatica  $LL(1)$  equivalente.

La trasformazione consiste nel mettere in evidenza il più lungo prefisso comune ai linguaggi definiti dagli archi uscenti dallo stesso stato. In sostanza si modifica il grafo (o la grammatica) in modo da raccogliere i prefissi comuni in un cammino comune, che sarà posto a monte dei cammini che differenziano i suffissi dei due casi. L'effetto è di spostare più a valle il bivio.

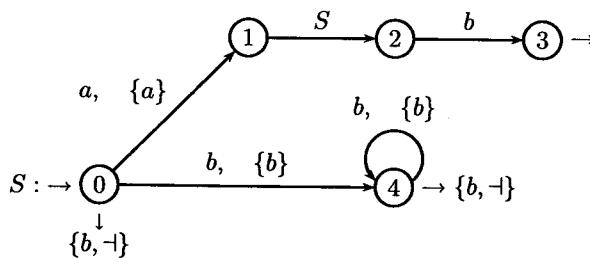
Si noti la somiglianza concettuale con il metodo di determinizzazione degli automi finiti.

### ~~Altre trasformazioni~~

Esistono altri tipi di infrazioni, alle quali si rimedia con trasformazioni di tipo diverso, come ad es. la seguente.

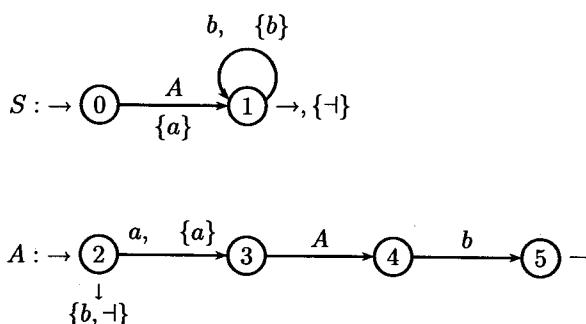
*Esempio 4.36.* Estrazione d'un costrutto da una ricorsione.  
Il linguaggio  $\{a^n b^* b^n \mid n \geq 0\}$  è definito dalla grammatica:

$$S \rightarrow aSb \mid b^*$$



Gli insiemi guida sono sovrapposti negli stati 0 e 4. Spostando il termine  $b^*$  all'esterno della ricorsione, si ottiene la grammatica equivalente  $LL(1)$ :

$$S \rightarrow Ab^* \quad A \rightarrow aAb \mid \epsilon$$



Infine si ricorda che, se la grammatica data genera un linguaggio regolare, è facile ottenere una grammatica equivalente  $LL(1)$ . Basta infatti costruire l'automa finito deterministico che riconosce il linguaggio: esso presenta soltanto etichette terminali e soddisfa per definizione la condizione  $LL(1)$ .

#### 4.4.3 Allungamento della prospezione

Per ottenere un parsificatore deterministico, quando la condizione  $LL(1)$  cade, invece di modificare la grammatica, conviene spesso adottare una tattica alternativa, che consiste nell'esaminare non solo il carattere corrente, ma anche quello o quelli successivi. L'algoritmo, in altre parole, compie una prospezione di lunghezza  $k \geq 1$  sul testo. Se, così facendo, la scelta della mossa da compiere per uscire dallo stato diventa unica, si dice che esso soddisfa la condizione  $LL(k)$ .

Una grammatica si dice  $LL(k)$  se esiste un intero  $k$  tale che, per ogni macchina della rete e per ogni stato, la scelta tra le frecce uscenti dallo stato possa essere fatta utilizzando una prospezione di lunghezza  $\leq k$ .  
È sufficiente un esempio per illustrare il calcolo degli insiemi guida di lunghezza  $k = 2$ .

*Esempio 4.37.* Conflitto tra etichette e variabili.

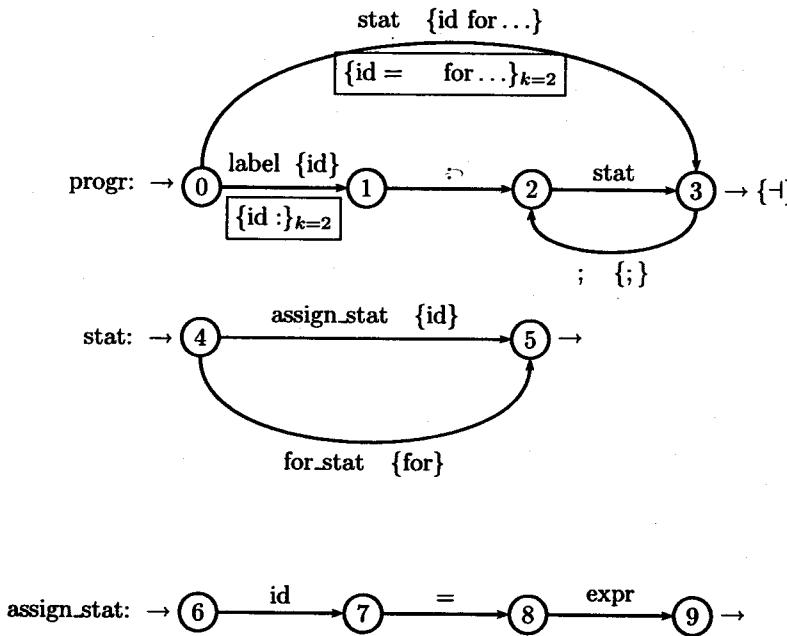
Si definisce una parte d'un linguaggio di programmazione. Esso è una lista di istruzioni (assegnamenti, frasi *for*, frasi *if*, ecc.), con o senza etichetta. Le etichette e i nomi di variabili sono degli *identificatori*.

La grammatica estesa è:

```

    progr → [label :]stat(; stat)*
    stat → assign_stat | for_stat | ...
    assign_stat → id = expr
    for_stat → ...
    label → id
    expr → ...
  
```

Le macchine della rete sono mostrate con gli insiemi guida di lunghezza 1 e di lunghezza 2 (inquadriati):



Lo stato 0 non è  $LL(1)$  perché entrambi gli insiemi guida (tra graffe non inquadrati) contengono l'identificatore. Ma si osserva che, se l'identificatore è un'etichetta, è sempre seguito da due punti; mentre, nel caso dell'arco stat, l'identificatore è la parte sinistra d'un assegnamento, seguito dunque dal segno di eguale.<sup>13</sup> Poiché, la prospezione del secondo carattere determina la scelta della mossa nello stato 0, esso soddisfa la condizione  $LL(2)$ . Le due frecce uscenti dallo stato 0 sono contraddistinte dagli insiemi guida di lunghezza due, inquadrati in figura. Poiché gli insiemi sono disgiunti, lo stato 0 soddisfa la condizione  $LL(2)$ .

Si noti che nello stato 3 è sufficiente la lunghezza  $k = 1$ , e sarebbe sciocco uniformare la prospezione al massimo dei valori necessari nella grammatica.

Gli elementi d'un insieme guida  $LL(k)$  sono formalmente delle stringhe di lunghezza  $k$ .<sup>14</sup> In pratica i parsificatori che operano con il metodo  $LL(k)$  usano lunghezze diverse di prospezione, con un criterio di economia: in ogni stato si usa la minima lunghezza  $m \leq k$ , necessaria per risolvere la scelta tra

<sup>13</sup>Nel caso dell'istruzione *for* o *if* si suppone per semplicità che i lessemi *for* o *if* siano parole chiave riservate, vietate come identificatori di variabili.

<sup>14</sup>Possono avere lunghezza inferiore quando terminano con il terminatore  $\dashv$ .

le frecce uscenti dallo stato; se il  $m$ -esimo carattere di prospezione è sufficiente a risolvere l'incertezza, è inutile esaminare il successivo.

Sono anche stati sviluppati parsificatori<sup>15</sup> che, oltre a eseguire una prospezione di lunghezza variabile, negli stati dove una prospezione di lunghezza finita non basta, fanno uso d'un a condizione semantica per effettuare la scelta: l'idea sarà presentata nel prossimo capitolo.

In conclusione, il vantaggio di usare una prospezione più lunga d'uno è di evitare la modifica delle regole grammaticali, al contrario delle trasformazioni, come la fattorizzazione sinistra, che producono sgradite deformazioni della grammatica di riferimento del linguaggio.

### Limiti della famiglia $LL(k)$

La *famiglia  $LL(k)$*  contiene tutti e soli i linguaggi che possono essere definiti da una grammatica  $LL(k)$  per un valore finito di  $k \geq 1$ .

La prima limitazione è che non tutti i linguaggi deterministici sono generabili da grammatiche  $LL(k)$ .

*Proprietà 4.38.* La famiglia dei linguaggi  $LL(k)$  è strettamente contenuta nella famiglia *DET* dei linguaggi deterministici.

Un caso semplice è il seguente.

*Esempio 4.39.* Linguaggio deterministico ma non  $LL(k)$ .

È facile costruire l'automa a pila deterministico del linguaggio  $L_1 = \{a^*a^n b^n \mid n \geq 0\}$ . Si impila un simbolo  $A$  per ogni lettera  $a$  letta; al primo  $b$  letto si cambia stato, e si disimpila una  $A$ ; poi, per ogni successiva  $b$ , si disimpila una  $A$ ; si riconosce se al termine della stringa la pila contiene zero o più  $A$ .

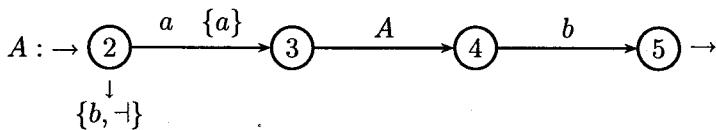
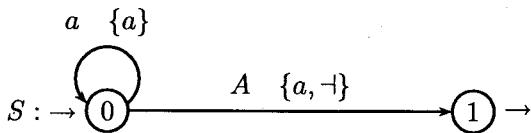
Una grammatica del linguaggio è  $G_1$ :

$$S \rightarrow a^* A \quad A \rightarrow aAb \mid \epsilon$$

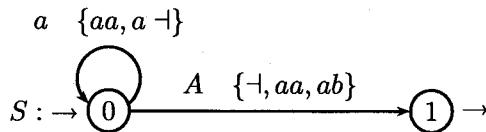
Le macchine della rete, con gli insiemi guida, sono:

---

<sup>15</sup>Come ANTLR [40].



Gli insiemi guida per  $k = 1$  nello stato 0 sono sovrapposti. Ma anche con  $k = 2$  gli insiemi guida, sotto mostrati, contengono una stringa comune,  $aa$ :



In generale, per quanto grande si prenda  $k$ , si trova che entrambi gli archi uscenti dallo stato 0 sono compatibili con la prospezione  $a^k$ . Dunque  $G_1$  non è una grammatica  $LL(k)$ , per nessun valore finito di  $k$ .

Sorge la domanda se, per lo stesso linguaggio, esista una grammatica che gode della proprietà  $LL(k)$ ; la risposta è negativa.<sup>16</sup>

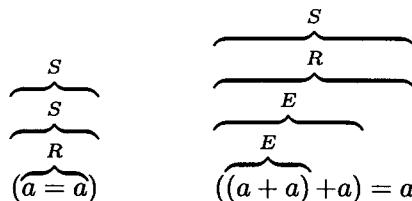
Questo esempio pur semplice evidenzia una limitazione della famiglia dei linguaggi  $LL(k)$ . Il prossimo mostra che talvolta la decisione di usare il metodo  $LL(k)$  impone al progettista del linguaggio alcune cautele nella scelta dei simboli terminali, al fine di evitare sovrapposizioni negli insiemi guida.

*Esempio 4.40.* Relazioni e espressioni.

La grammatica  $G$ :

$$S \rightarrow R \mid (S) \quad R \rightarrow E = E \quad E \rightarrow a \mid (E + E)$$

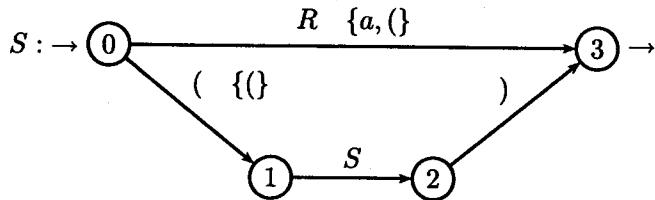
definisce delle relazioni di eguaglianza tra espressioni aritmetiche additive, quali



<sup>16</sup> La dimostrazione usa il lemma di pompaggio dei linguaggi  $LL(k)$  [6].

Mentre una relazione come  $a = a$  può essere racchiusa tra parentesi, un'espressione additiva (non atomica) deve esserlo.

La prima regola viola la condizione  $LL(1)$ :



Infatti una stringa iniziante con il carattere '(' può essere una relazione  $R$  parentesizzata oppure un'espressione  $E$ .

Lo stesso ragionamento mostra che non esiste un valore finito di  $k$  per cui la  $G$  risulti  $LL(k)$ . Infatti, per quanto grande si prenda  $k$ , si possono trovare due stringhe simili alle precedenti tali che: entrambe iniziano con  $k + 1$  parentesi aperte, la prima di esse è generata mediante la regola  $S \rightarrow R$ , mentre la seconda è generata mediante la regola  $S \rightarrow (S)$ .

Poiché, comunque sia fatta la grammatica che genera questo linguaggio, essa necessariamente presenta una scelta tra le due strutture, si potrebbe dimostrare una proprietà più forte: che il linguaggio non è  $LL(k)$ .

D'altra parte il linguaggio  $L(G)$  è deterministico, come si potrebbe vedere in seguito applicando il metodo  $LR(1)$  per costruire il parsificatore.

In questo caso, per rendere  $LL(1)$  il linguaggio, è necessario modificarlo. Una possibilità è di differenziare le parentesi, usando ad esempio le parentesi quadre per le espressioni aritmetiche, e le tonde per le relazioni.

## 4.5 Analisi sintattica ascendente deterministica

Il metodo  $LL(1)$  non è applicabile se da uno stato d'una macchina escono due frecce con insiemi guida sovrapposti. Poiché per ipotesi ogni macchina è deterministica, almeno una delle uscite è un arco di etichetta nonterminale  $A$  o la freccia d'uno stato finale. Nel primo caso, la caduta della condizione  $LL(1)$  significa che in quello stato, se il prossimo carattere sta in entrambi gli insiemi guida, non si sa se invocare la macchina  $A$  o compiere l'altra mossa. Una tattica efficace è quella di rinviare la decisione, se gli elementi disponibili non permettono di prenderla, e di proseguire il calcolo, tenendo aperte entrambe le strade, fino al momento in cui emergeranno nuove evidenze sufficienti a decidere. Tuttavia nell'intervallo, intercorrente tra la comparsa dell'incertezza e la sua risoluzione, il calcolo deve preservare le informazioni intermedie, che saranno necessarie per eseguire la decisione nel secondo momento.

Questa è l'idea alla base degli algoritmi di analisi sintattica ascendente, storicamente detti  $LR(k)$ <sup>17</sup>. Essi costruiscono un automa à pila dotato di stati interni, che applica una prospezione di lunghezza  $k$ . Diversamente dal caso  $LL(k)$ , ora ha senso ridurre a zero la lunghezza della prospezione, poiché nei casi più semplici la scelta può essere determinata dal solo stato in cui si trova l'automa. Per gradualità, l'esposizione inizia da questo caso, considerando prima le grammatiche BNF non estese.

### 4.5.1 Analisi $LR(0)$

Conviene introdurre l'idea partendo proprio da un esempio di grammatica non accettabile con il metodo  $LL(1)$ .

*Esempio 4.41.* Esempio introduttivo  $LR(0)$ .

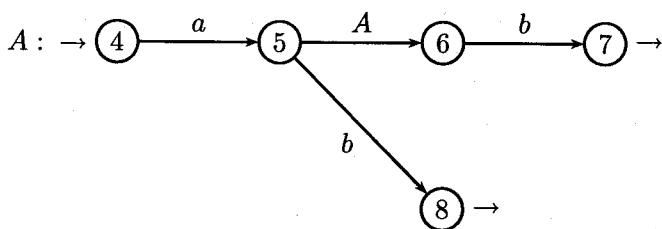
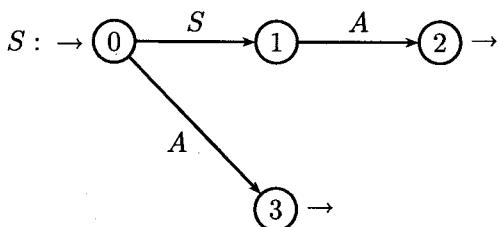
Una lista non vuota di costrutti  $a^n b^n, n \geq 1$  è definita dalla grammatica:

$$S \rightarrow SA \mid A \quad A \rightarrow aAb \mid ab$$

rappresentata dalla rete:

---

<sup>17</sup>Nella sigla dell'inventore Donald Knuth, la s indica in inglese che la scansione del testo è da sinistra a destra, la r indica che la derivazione calcolata è destra, e il parametro è la lunghezza di prospezione in caratteri.



Si osservi che ogni macchina è stata disegnata con l'avvertenza di tenere distinti gli stati finali associati alle diverse alternative grammaticali.

Il metodo  $LL(k)$  fallisce nello stato 0, poiché tanto  $S$  quanto  $A$  iniziano con  $a$ , essendo ricorsiva a sinistra la grammatica.

Si descrive il funzionamento dell'automa a pila deterministico costruito con il metodo  $LR(0)$ .

Inizialmente l'automa si trova nello stato di incertezza o *macrostato*  $I_0 = \{0, 4\}$ , poiché nello stato iniziale 0 dell'assioma è anche possibile invocare la macchina  $M_A$  (per brevità scritta semplicemente  $A$ ), avente stato iniziale 4. Si può quindi pensare che vi siano due automi a pila deterministici contemporaneamente attivi, l'automa  $S$  e l'automa  $A$ , che evolveranno simultaneamente. La situazione ricorda il prodotto cartesiano di due macchine finite, con la differenza che ora gli automi devono anche aggiornare la loro pila. Ciò che caratterizza il metodo  $LR(k)$  è che vi è una sola pila per tutti gli automi contemporaneamente attivi.

Il simbolo iniziale della pila è  $I_0$ . Data ad es. la stringa  $ab$ , leggendo e consumando la prima  $a$ , il parsificatore dal macrostato  $I_0 = \{0, 4\}$  va, con una mossa detta di *spostamento*, nel macrostato successivo  $I_2$  così calcolato. Si prende l'unione degli stati prossimi di 0 e di 4

$$\delta(0, a) \cup \delta(4, a) = \emptyset \cup \{5\} = \{5\}$$

Poiché da 5 si può invocare  $A$ , si aggiunge lo stato iniziale 4, completando così il macrostato  $I_2 = \{4, 5\}$ .

La mossa di spostamento si conclude impilando  $I_2$  sopra  $I_0$ .

Esaminando il prossimo carattere  $b$ , si trova che, tra gli stati 4 e 5 di  $I_2$ , soltanto 5 permette la mossa che porta nello stato 8; l'automa impila il macrostato  $I_3 = \{8\}$ .

Essendo lo stato 8 finale della macchina  $A$  è giunto il momento di scegliere la regola grammaticale da applicare,  $A \rightarrow ab$ , riducendo la stringa  $ab$  nel non-terminale  $A$ .

Tale operazione, detta di *riduzione*, cancella dalla pila  $I_2 I_3$ , ossia un numero di simboli pari alla lunghezza  $|ab| = 2$  della parte destra della regola riconosciuta nello stato 8.

La pila contiene ora  $I_0 = \{0, 4\}$ . Poiché il simbolo  $A$ , parte sinistra della regola riconosciuta, sta su un arco uscente dallo stato 0 (componente di  $I_0$ ), l'automa esegue uno spostamento e registra sulla pila il macrostato di arrivo della transizione  $\delta(0, A) = 3$ , denotato  $I_4 = \{3\}$ . Essendo 3 uno stato finale della macchina assioma, la stringa, di cui è stata completata la lettura, è accettata.

#### 4.5.2 Grammatiche $LR(0)$

I concetti precedenti si formalizzano attraverso una serie di definizioni che condurranno alla famiglia delle grammatiche  $LR(0)$ .

È data la grammatica  $G = (V, \Sigma, P, S)$ , con regole che per semplicità sono per ora del tipo non esteso. Ogni nonterminale  $A \in V$  è la parte sinistra d'una o più regole alternative  $A \rightarrow \beta \mid \gamma \mid \dots$ , che sono anche rappresentate da una macchina detta  $M_A$  (o  $A$  quando non vi sia rischio di confusione). Anche se non necessario per il metodo  $LR(k)$ , conviene mantenere l'ipotesi che la macchina sia deterministica. Lo stato iniziale di  $M_A$  è  $q_{A,0}$  e gli stati finali sono  $F_A$ .

Inoltre si suppone che ogni alternativa  $A \rightarrow \beta$  abbia uno stato finale distinto,  $q_{A,\beta} \in F_A$ ; ciò consentirà al parsificatore di selezionare l'alternativa da applicare nella mossa di riduzione.

Per facilitare l'esposizione, conviene aggiungere alla grammatica la regola iniziale  $S_0 \rightarrow S \dashv$ , che dice che ogni frase è seguita dal terminatore. Il simbolo  $S_0$  diviene così l'assioma, mentre  $S$  è degradato al ruolo d'un nonterminale qualsiasi.

Gli stati iniziale e finale della macchina dell'assioma sono rispettivamente  $q_{iniziale}$  e  $q_{terminale}$ .

La seguente funzione calcola gli stati raggiungibili da uno stato mediante una catena di zero o più mosse spontanee.

##### Definizione 4.42. Chiusura $LR(0)$

Sia  $q \in Q$  uno stato della rete. La chiusura di  $q$  è:

```

 $C := \{q\};$ 
repeat
 $C := C \cup \{q_{A,0} \mid \text{per qualche } p \in C \text{ e } A \in V, \exists \text{ la mossa } \delta(p, A)\};$ 
until nessun nuovo elemento è stato aggiunto a  $C$ ;
 $\text{chius}(q) := C$ 

```

Per un insieme  $R \subseteq Q$  di stati la chiusura è definita da:

$$\text{chius}(R) := \{r \mid \text{per qualche } q \in R, r \in \text{chius}(q)\}$$

La funzione serve nella costruzione della seguente macchina astratta.

**Definizione 4.43.** Macchina pilota dell'analisi LR(0). <sup>18</sup>  
Si definisce un automa finito

$$N = (R, \Sigma \cup V, \vartheta, I_0, R)$$

che piloterà il parsificatore: l'insieme degli stati, o meglio macrostati, è  $R \subseteq$  parti finite di  $Q$ ; non si usano stati finali, ma per convenzione si prende l'intero insieme  $R$  come finale;

l'alfabeto d'ingresso è l'unione di quello terminale e nonterminale della grammatica.

I macrostati,  $R = \{I_0, \dots\}$  e la funzione di transizione  $\vartheta$  sono calcolati, partendo dal macrostato iniziale, per mezzo della procedura:

```

 $I_0 = \text{chius}(q_{ini});$ 
 $R := \{I_0\};$ 
repeat per ogni  $I_j \in R$  e per ogni  $X \in \Sigma \cup V$ 
 $\vartheta(I_j, X) := \text{chius}(\{r \mid r \in \delta(q, X), \text{ per qualche } q \in I_j\});$ 
if  $\vartheta(I_j, X) \notin R$  then  $R := R \cup \vartheta(I_j, X);$ 
until nessun nuovo macrostato è stato creato dall'iterazione;

```

La funzione  $\vartheta$  produce un macrostato, da aggiungere all'insieme  $R$ , se non già presente. Il calcolo della funzione e dei macrostati prosegue finché nessun nuovo macrostato è prodotto dal passo.

Un macrostato ha dunque come componenti uno o più stati, che possono essere finali o non, in qualche macchina della rete.

In un macrostato, uno stato è detto *di spostamento* se da esso, nella macchina della rete, esce un arco verso un altro stato.

Uno stato finale, appartenente a un macrostato, è detto *stato di riduzione*.

Le *riduzioni* associate a un macrostato sono le regole sintattiche il cui stato finale compare nel macrostato:

$$\text{riduz}(I_j) = \{A \rightarrow \alpha \mid q_{A,\alpha} \in I_j\}$$

---

<sup>18</sup>Anche detta *riconoscitore dei prefissi ascendenti* per le ragioni poi esposte.

Si noti che uno stato finale può essere allo stesso tempo di riduzione e di spostamento: ciò avviene quando da esso esce anche un arco etichettato. Riunendo i concetti, un macrostato può essere classificato come:

*riduzione*: se contiene soltanto stati di riduzione;

*spostamento*: se contiene soltanto stati di spostamento;

*misto*: se contiene stati di spostamento e stati di riduzione.

La condizione più semplice di determinismo è la seguente.

**Definizione 4.44.** Condizione  $LR(0)$  e famiglia  $LR(0)$ .

Una grammatica soddisfa la condizione  $LR(0)$  se ogni macrostato della macchina pilota verifica entrambe le condizioni:

1. non è misto;

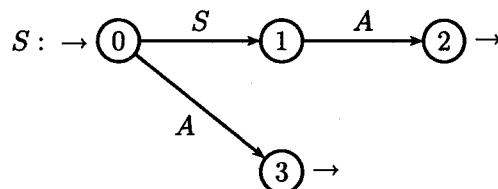
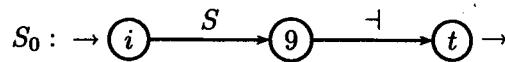
2. se è un macrostato di riduzione, contiene un solo stato.

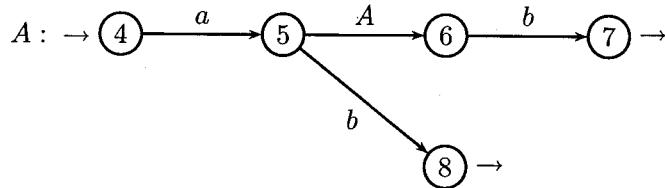
La famiglia dei linguaggi  $LR(0)$  è l'insieme dei linguaggi che possono essere generati da grammatiche della famiglia  $LR(0)$ .

Si osservi che la condizione 1. vieta la compresenza d'un o stato finale e non finale nello stesso macrostato. La condizione 2. vieta la compresenza di due o più stati finali.

**Esempio 4.45.** Costruzione della macchina pilota per la grammatica dell'es. 4.41, riprodotta con l'aggiunta del nuovo assioma:

$$S_0 \rightarrow S \dashv \quad S \rightarrow SA \mid A \quad A \rightarrow aAb \mid ab$$

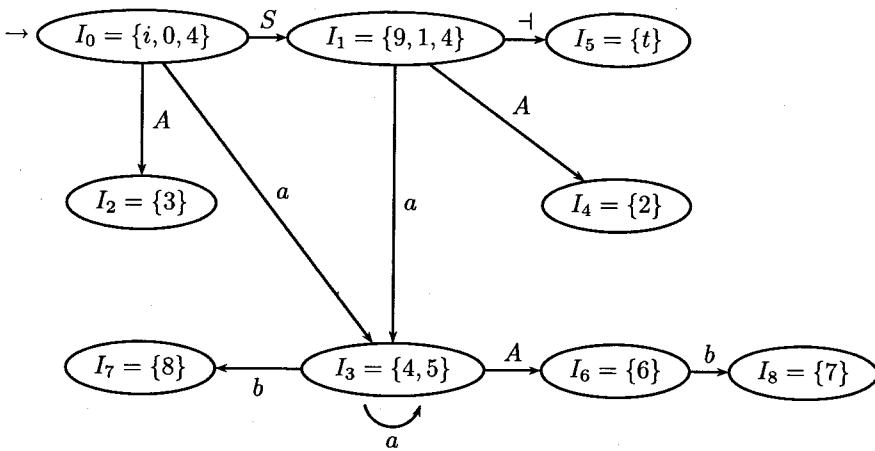




Sono riportati i passi di calcolo dei macrostati e della funzione di transizione del pilota.

|    | Insieme dei macrostati                           | Macrostate creato  | Mossa creata |
|----|--|--|--------------|
| 0  | $\emptyset$                                      | $\text{chius}(i) = \{i, 0, 4\} = I_0$  |              |
| 1  | $I_0 = \{i, 0, 4\}$                              | $\text{chius}(\delta(i, S) \cup \delta(0, S) \cup \delta(4, S)) = \vartheta(I_0, S) = I_1$<br>$\text{chius}(\{9, 1\}) = \{9, 1, 4\} = I_1$ |              |
| 2  | $I_0 = \{i, 0, 4\}, I_1$                         | $\text{chius}(\delta(i, A) \cup \delta(0, A) \cup \delta(4, A)) = \vartheta(I_0, A) = I_2$<br>$\text{chius}(3) = \{3\} = I_2$              |              |
| 3  | $I_0 = \{i, 0, 4\}, I_1, I_2$                    | $\text{chius}(\delta(i, a) \cup \delta(0, a) \cup \delta(4, a)) = \vartheta(I_0, a) = I_3$<br>$\text{chius}(5) = \{4, 5\} = I_3$           |              |
| 4  | $I_0, I_1 = \{9, 1, 4\}, I_2, I_3$               | $\text{chius}(\delta(9, A) \cup \delta(1, A) \cup \delta(4, A)) = \vartheta(I_1, A) = I_4$<br>$\text{chius}(2) = \{2\} = I_4$              |              |
| 5  | $I_0, I_1 = \{9, 1, 4\}, I_2, I_3, I_4$          | $\text{chius}(\delta(9, a) \cup \delta(1, a) \cup \delta(4, a)) = \vartheta(I_1, a) = I_3$<br>$\text{chius}(5) = \{4, 5\} = I_3$           |              |
| 6  | $I_0, I_1 = \{9, 1, 4\}, I_2, I_3, I_4$          | $\text{chius}(\delta(9, \neg) \cup \delta(1, \neg) \cup \delta(4, \neg)) = \vartheta(I_1, \neg) = I_5$<br>$\text{chius}(t) = \{t\} = I_5$  |              |
| 7  | $I_0, I_1, I_2, I_3 = \{4, 5\}, I_4, I_5$        | $\text{chius}(\delta(4, A) \cup \delta(5, A)) = \vartheta(I_3, A) = I_6$<br>$\text{chius}(6) = \{6\} = I_6$                                |              |
| 8  | $I_0, I_1, I_2, I_3 = \{4, 5\}, I_4, I_5, I_6$   | $\text{chius}(\delta(4, a) \cup \delta(5, a)) = \text{chius}(5) = \vartheta(I_3, a) = I_3$<br>$\{4, 5\} \equiv I_3$                        |              |
| 9  | $I_0, I_1, I_2, I_3 = \{4, 5\}, I_4, I_5, I_6$   | $\text{chius}(\delta(4, b) \cup \delta(5, b)) = \text{chius}(8) = \vartheta(I_3, b) = I_7$<br>$\{8\} = I_7$                                |              |
| 10 | $I_0, I_1, I_2, I_3, I_4, I_5, I_6 = \{6\}, I_7$ | $\text{chius}(\delta(6, b)) = \text{chius}(7) = \{7\} = \vartheta(I_6, b) = I_8$<br>$I_8$  |              |
| 11 | $I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8$    |  | alt          |

Segue il grafo dell'automa pilota.



I macrostati  $I_2, I_4, I_5, I_7, I_8$  sono di riduzione e contengono un solo stato finale. Gli altri macrostati sono di spostamento, poiché contengono soltanto stati non finali. Non vi sono macrostati misti, né due o più riduzioni nello stesso macrostato di riduzione. Pertanto la condizione è verificata e la grammatica ha la proprietà  $LR(0)$ .

### Proprietà della macchina pilota

Osservando l'esempio è facile riscontrare le particolari proprietà del grafo d'una macchina pilota  $LR(0)$ :

1. tutte le frecce entranti in un macrostato portano la stessa etichetta (vedasi  $I_3$ );
2. ogni macrostato di riduzione è privo di successori;
3. ogni macrostato di spostamento ha almeno un successore.

Si può chiedere quale sia il linguaggio riconosciuto dalla macchina pilota, supponendo che ogni macrostato sia finale. Nell'esempio le stringhe riconosciute sono

$S, A, a, S \dashv, SA, Sa, aA, aa, ab, SaA, Saa, aAb, aaa, aab, aaAb, \dots$

Esse hanno la caratteristica di essere dei prefissi delle forme di frasi generate dalla grammatica  $G$ , mediante derivazioni destre.

Ad es. la derivazione destra

$$S_0 \Rightarrow S \dashv \Rightarrow A \dashv \Rightarrow aAb \dashv \Rightarrow aaAbb \dashv \Rightarrow aaabbb \dashv$$

presenta i prefissi accettati dal pilota:

$$S, S \dashv, A, A \dashv, a, aA, aAb, aa, aaA, aaAb, aaab$$

Ma non tutti i prefissi delle forme destre sono accettati, ad es.  $aaAbb$  e  $aaabb$  non lo sono.

La spiegazione di questa proprietà si trova nel ragionamento che ha portato alla costruzione del pilota, con la tattica di condurre in contemporanea più calcoli, fino al momento in cui uno di essi raggiunge uno stato finale d'una macchina, ossia un macrostato di riduzione.

Più precisamente, si osservino nel grafo del pilota i cammini che risalgono a ritroso da un macrostato di riduzione.

Sia  $A \rightarrow \beta$  la riduzione associata al macrostato di riduzione  $I$ ; allora esiste nel pilota un cammino retroverso, etichettato  $\beta^R$ , che da  $I$  conduce a un macrostato contenente come componente lo stato iniziale della macchina  $A$ . Così dal macrostato  $I_8 = \{7\}$  i tre passi a ritroso  $b, A, a$  risalgono al macrostato  $I_0$ , che contiene lo stato iniziale 4 della macchina  $A$ ; oppure al macrostato  $I_3$ , che pure contiene 4.

In che modo differiscono i prefissi riconosciuti e non riconosciuti dal pilota? Un prefisso non riconosciuto come  $aaabb$  contiene una sottostringa *interna* che è la parte destra d'una regola:

$$\begin{array}{c} aa \quad \underbrace{ab} \quad b \\ A \rightarrow ab \end{array}$$

Invece in un prefisso riconosciuto, come  $aaab$ , tale sottostringa può solo occupare la posizione finale (suffisso):

$$\begin{array}{c} aa \quad \underbrace{ab} \\ A \rightarrow ab \end{array}$$

#### 4.5.3 Analizzatore a spostamento e riduzione

Se una grammatica è  $LR(0)$ , dalla sua macchina pilota si ottiene subito l'automa a pila deterministico che riconosce le frasi del linguaggio e ne costruisce gli alberi sintattici. Esso opera nel seguente modo. I simboli della pila sono i macrostati, ai quali si possono affiancare per migliore leggibilità anche i simboli della grammatica. Esaminando il carattere terminale corrente, l'automa, avendo in cima alla pila il macrostato  $I$ , esegue la mossa prescritta dall'automa pilota e impila il prossimo macrostato; tale operazione è detta di *spostamento*. Avendo in cima alla pila un macrostato di riduzione, cui è per la seconda condizione  $LR(0)$  associata una sola alternativa sintattica, l'automa esegue una serie di operazioni note come *riduzione*. Esse simulano l'omonimo passo nella derivazione d'una frase, prima cancellando, poi inserendo simboli sulla pila.

Al termine della scansione, l'automa accetta la stringa sorgente se la pila è vuota (o se contiene il solo macrostato iniziale).

*Algoritmo 4.46.* Analizzatore a spostamento e riduzione (senza prospezione). Sia  $G$  una grammatica  $LR(0)$  e  $N = (R, \Sigma \cup V, \vartheta, I_0, R)$  la sua macchina pilota. Si costruisce l'automa a pila  $A$  come segue.

Alfabeto di pila:  $R \cup \Sigma \cup V$ ;

Insieme degli stati di  $A$ : irrilevante perché contiene un solo stato;

Configurazione iniziale: la pila contiene il macrostato iniziale  $I_0$ ;

Mosse dell'automa: vi sono due tipi di mosse, quelle di spostamento e quelle di riduzione. Sia  $I$  il macrostato in cima alla pila e  $a$  il carattere corrente:

Mossa di *spostamento*: se per il macrostato corrente  $I$  è definita nel pilota la mossa  $\vartheta(I, a) = I'$ , l'automa legge  $a \in \Sigma$ , avanzando la testina, e impila la stringa  $aI'$ ;

Mossa di *riduzione*: se il macrostato corrente  $I$ , qui denotato  $I_n$ , contiene lo stato di riduzione  $q$ , con

$$\text{riduz}(q) = \{B \rightarrow X_1 X_2 \dots X_n\}$$

dove  $n \geq 0$  è la lunghezza della parte destra, si compie l'azione seguente.

In cima si troverà necessariamente (per il modo in cui il pilota è stato costruito) una stringa  $\beta'$

$$\dots I' \overbrace{X_1 I_1 X_2 I_2 \dots X_n I_n}^{\beta'}$$

contenente  $n$  macrostati inseriti da mosse precedenti.

La mossa di riduzione è una mossa spontanea (senza avanzamento della testina) che, prima cancella dalla pila la stringa  $\beta'$  (ossia i primi  $2n$  simboli dalla cima), poi inserisce la stringa  $BI''$ , dove  $I''$  è il macrostato raggiunto leggendo (per così dire) il nonterminale  $B$ , ossia è  $I'' = \vartheta(I', B)$ .

Configurazione di riconoscimento: pila =  $I_0$ , ingresso = -

Se l'automa a pila raggiunge una configurazione in cui la mossa non è possibile, esso rifiuta la stringa.

Si noti che, grazie all'ipotesi  $LR(0)$ , il macrostato in cima alla pila seleziona il tipo di mossa, accettazione, spostamento o riduzione, e nell'ultimo caso designa univocamente la riduzione da eseguire. Di conseguenza il funzionamento è deterministico.

A meglio vedere, le informazioni messe in pila sono ridondanti: infatti lo spostamento inserisce nella pila un carattere terminale  $a$  e il macrostato di arrivo  $I'$ , ma dal secondo si deduce il primo, perché nel grafo della macchina pilota tutti gli archi entranti in un macrostato portano la stessa etichetta. L'inserimento del carattere terminale ha soltanto scopo esplicativo e permette di seguire più facilmente le simulazioni dell'automa a pila.

Una questione più sottile riguarda gli stati dell'automa a pila. È davvero corretta la precedente affermazione che l'automa non fa uso degli stati? A meglio vedere, la mossa di riduzione è un'operazione più complessa delle istruzioni elementari consentite a un automa a pila. Certamente la riduzione può essere decomposta in una serie di istruzioni elementari, ma facendo ricorso a una memoria finita per contare quanti simboli vanno rimossi dalla pila, e per memorizzare il simbolo nonterminale  $B$  della parte sinistra della riduzione. In conclusione, l'automa a pila  $LR(0)$  fa un uso nascosto di stati interni nelle mosse di riduzione.

Passando al metodo generale  $LR(k)$ , gli stati interni avranno un ulteriore motivo: per memorizzare i  $k$  caratteri esaminati dalla prospezione (esattamente come nel caso  $LL(k)$ ).

In definitiva, tutti i parsificatori deterministici, fanno uso d'una memoria finita oltre che della pila illimitata (anche se la presentazione degli algoritmi non ne parla per non cadere in una descrizione di livello troppo basso). Il contrario sarebbe sorprendente, infatti si sa che la famiglia  $DET$  dei linguaggi deterministici (p. 160) è caratterizzata dall'avere come riconoscitore un automa a pila deterministico dotato di stati interni.

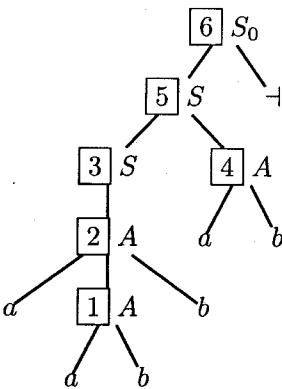
### Derivazione destra inversa

Si mostra ora come l'automa a pila parsifica una stringa e in qual ordine ricostruisce l'albero sintattico.

*Esempio 4.47.* Traccia dell'analisi (es. 4.45).

| Pila  | $x$    |        |        |              |          |          | Commento |                                       |
|-------|--------|--------|--------|--------------|----------|----------|----------|---------------------------------------|
| $I_0$ | $a$    | $a$    | $b$    | $b$          | $a$      | $b$      | $\dashv$ | sposta                                |
| $I_0$ | $aI_3$ | $a$    | $b$    | $b$          | $a$      | $b$      | $\dashv$ | sposta                                |
| $I_0$ | $aI_3$ | $aI_3$ | $b$    | $b$          | $a$      | $b$      | $\dashv$ | sposta                                |
| $I_0$ | $aI_3$ | $aI_3$ | $bI_7$ | $b$          | $a$      | $b$      | $\dashv$ | riduci con $A \rightarrow ab$         |
| $I_0$ | $aI_3$ | $AI_6$ | $b$    | $a$          | $b$      | $\dashv$ |          | sposta                                |
| $I_0$ | $aI_3$ | $AI_6$ | $bI_8$ | $a$          | $b$      | $\dashv$ |          | riduci con $A \rightarrow aAb$        |
| $I_0$ | $AI_2$ |        |        | $a$          | $b$      | $\dashv$ |          | riduci con $S \rightarrow A$          |
| $I_0$ | $SI_1$ |        |        | $a$          | $b$      | $\dashv$ |          | sposta                                |
| $I_0$ | $SI_1$ |        | $aI_3$ | $b$          | $\dashv$ |          |          | sposta                                |
| $I_0$ |        | $SI_1$ | $aI_3$ | $bI_7$       | $\vdash$ |          |          | riduci con $A \rightarrow ab$         |
| $I_0$ |        | $SI_1$ |        | $AI_4$       | $\vdash$ |          |          | riduci con $S \rightarrow SA$         |
| $I_0$ |        | $SI_1$ |        |              | $\vdash$ |          |          | sposta                                |
| $I_0$ |        |        | $SI_1$ | $\dashv I_5$ |          |          |          | riduci con $S_0 \rightarrow S \dashv$ |
| $I_0$ |        |        |        |              |          |          |          | accetta                               |

### L'albero sintattico



è costruito dalle mosse di riduzione nell'ordine dei numeri crescenti. Le regole applicate sono

$$A \rightarrow ab, A \rightarrow aAb, S \rightarrow A, A \rightarrow ab, S \rightarrow SA, S_0 \rightarrow S \dashv$$

e corrispondono alla catena di riduzioni:

$$aabbab \dashv \Rightarrow aAbab \dashv \Rightarrow Aab \dashv \Rightarrow Sab \dashv \Rightarrow SA \dashv \Rightarrow S \dashv \Rightarrow S_0$$

L'osservazione dell'ordine di costruzione dell'albero evidenzia che questo è un parsificatore ascendente. Invertendo specularmente l'ordine della riduzione, si ottiene 654321 e la corrispondente derivazione  $S_0 \stackrel{+}{\Rightarrow} aabbab \dashv$ , è una derivazione destra, come già osservato (p. 168).

#### 4.5.4 Analisi sintattica con prospettiva $LR(k)$

Il prossimo algoritmo di analisi è il più potente tra quelli deterministici. Esso combina efficacemente due accorgimenti: l'uso della prospettiva, come nei metodi  $LL(k)$ , e la tattica di rinvio delle scelte, come nel caso  $LR(0)$ . Dopo un esame dei limiti del metodo  $LR(0)$ , si espone il procedimento  $LR(k)$  di calcolo della prospettiva per la macchina pilota, fissando a uno la lunghezza della prospettiva. Tale scelta da un lato semplifica l'esposizione, dall'altro, diversamente dal caso  $LL(k)$ , non restringe la famiglia dei linguaggi riconoscibili.

#### Limiti del metodo $LR(0)$

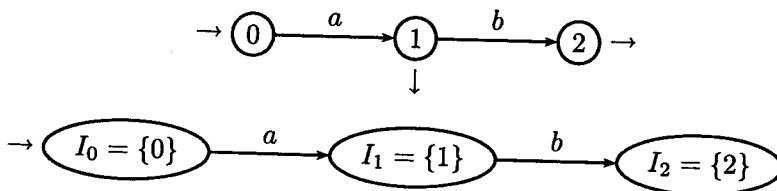
Avviene di rado che un parsificatore si possa costruire con il metodo  $LR(0)$  e ben pochi linguaggi tecnici soddisfano tale condizione.

La limitazione più seria dei linguaggi  $LR(0)$  è la seguente: se una stringa appartiene al linguaggio, nessun prefisso di essa può appartenervi, ossia i linguaggi  $LR(0)$  sono necessariamente *privi di prefissi*.

*Esempio 4.48.* Un linguaggio finito non  $LR(0)$ .

Il linguaggio finito  $\{a, ab\}$  non è  $LR(0)$ , perché una frase è prefisso d'un'altra. Si osservi la grammatica, la macchina corrispondente e la macchina pilota:

$$S \rightarrow a \mid ab$$



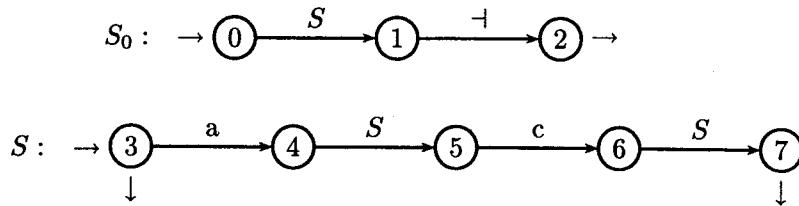
Il macrostato  $I_1$ , contenendo lo stato finale 1, è misto: di riduzione con la regola  $S \rightarrow a$  e di spostamento per l'arco uscente verso 2. Pertanto la condizione  $LR(0)$  cade. Intuitivamente, ciò significa che l'analizzatore non può decidere se applicare la riduzione  $a \Rightarrow A$  o continuare a spostare; per deciderlo dovrebbe esaminare il prossimo carattere ( $\dashv$  per la riduzione e  $b$  per lo spostamento), come farebbe un analizzatore  $LL(1)$ .

Un secondo esempio contenente dei prefissi è il classico linguaggio delle liste con separatore  $e(se)^*$ , dove s'incontrano le stringhe  $e$ ,  $ese$ , una prefisso dell'altra. Vi sono dunque linguaggi regolari che escono dalla famiglia  $LR(0)$ . Ma la limitazione relativa ai prefissi è ancora più seria, perché si applica alle stringhe generate, non soltanto dall'assioma, ma da qualsiasi nonterminale. Di conseguenza, le liste con separatori non possono comparire neanche come costrutto in un linguaggio tecnico, il che preclude ad es. le liste dei parametri d'una procedura, e tanti casi simili.

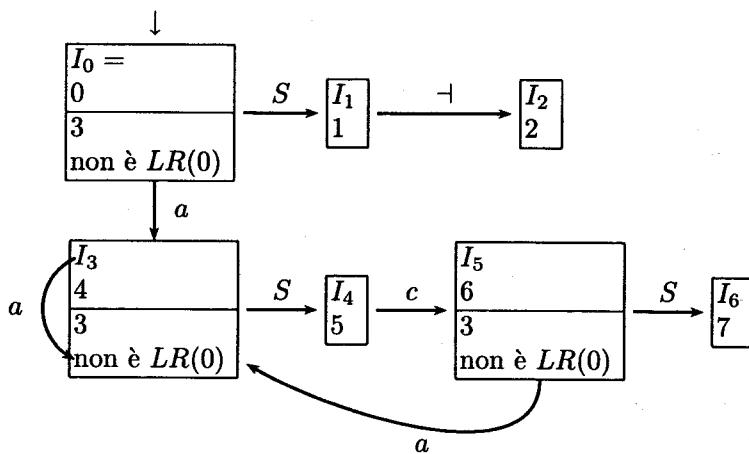
Proseguendo la rassegna delle limitazioni, è semplice vedere che la presenza di regole vuote nella grammatica viola la condizione  $LR(0)$ . Infatti una regola del tipo  $A \rightarrow \epsilon \mid \beta$  fa sì che lo stato iniziale  $q_{A,0}$  della macchina  $M_A$  sia anche finale. Ma da tale stato esce anche l'arco associato alla parte destra (non vuota)  $\beta$ . Di conseguenza, il macrostato contenente lo stato  $q_{A,0}$  viola la condizione  $LR(0)$ .

*Esempio 4.49.* La grammatica di Dyck non è  $LR(0)$ .  
La nota grammatica

$$S_0 \rightarrow S \dashv \quad S \rightarrow aScS \mid \epsilon$$



porta alla macchina pilota  $LR(0)$  seguente:



Nella figura ogni macrostato è suddiviso dalla riga orizzontale nei sottoinsiemi calcolati dalla funzione di chiusura.

Tutti i macrostati contenenti lo stato 3 sono misti, per la presenza della riduzione  $S \rightarrow \epsilon$  e dello spostamento  $3 \xrightarrow{a} 4$ ; essi sono dunque inadeguati per  $LR(0)$ .

#### Aggiunta della prospezione alla macchina pilota

Per potenziare il metodo  $LR(0)$  e trasformarlo nel metodo pratico  $LR(1)$ , si aggiungerà a ogni macrostato l'informazione sulla prospezione.

Poiché un macrostato può contenere più stati, è necessario calcolare l'insieme dei caratteri terminali di prospezione per ciascuno stato e per ogni freccia uscente da esso.

Intuitivamente, un macrostato risulta adeguato per l'analisi  $LR(1)$  se la prossima mossa dell'automa a pila è univocamente determinata dal macrostato posto in cima alla pila e dall'insieme di prospezione cui appartiene il carattere corrente.

**Definizione 4.50.** *Componenti della macchina pilota  $LR(1)$ .*

*Un macrostato  $I$  della macchina pilota  $LR(1)$  è un insieme di coppie, dette candidate, della forma*

$$\langle q, a \rangle \in Q \times (\Sigma \cup \{\vdash\})$$

*dove  $Q$  è l'insieme degli stati della rete di macchine.*

*Per brevità, più candidate aventi in comune lo stesso stato saranno solitamente raggruppate:*

$$\langle q, \{a_1, a_2, \dots, a_k\} \rangle \equiv \{\langle q, a_1 \rangle, \langle q, a_2 \rangle, \dots, \langle q, a_k \rangle\}$$

*L'insieme  $\{\widehat{a_1}, \widehat{a_2}, \dots, \widehat{a_k}\}$  è detto insieme di prospezione dello stato  $q$ .*

La funzione di chiusura sotto descritta arricchisce quella del caso  $LR(0)$ , con il calcolo degli insiemi di prospezione.

**Algoritmo 4.51.** Chiusura  $LR(1)$ .

Sia  $c = \langle q, \pi \rangle$  una candidata. La chiusura  $LR(1)$  di  $c$ ,  $\text{chius}_1(c)$ , è così calcolata:

$$C := \{\langle q, \pi \rangle\};$$

*repeat*

$$C := C \cup \{\langle q_{A,0}, \rho \rangle\} \text{ dove:}$$

$q_{A,0}$  è lo stato iniziale della macchina  $M_A$  tale che:

esiste una candidata  $\langle p, \pi \rangle \in C \wedge$  da  $p$  esce un arco  $\delta(p, A) = p'$ .

La prospezione  $\rho$  è così calcolata:

$$\begin{cases} \rho = \text{Ini}(L(p')) & \text{se } L(p') \text{ non è annullabile,} \\ \rho = \text{Ini}(L(p')) \cup \pi & \text{altrimenti} \end{cases}$$

*until* nessun nuovo elemento è stato aggiunto a  $C$ ;  
 $\text{chius}_1(q) := C$

Commento: se nell'insieme corrente  $C$  vi è uno stato  $p$  con prospezione  $\pi$ , e nella grammatica esiste l'arco  $p \xrightarrow{A} p'$ , allora si aggiunge all'insieme  $C$  lo stato iniziale  $q_{A,0}$  (come nel caso  $LR(0)$ ). Per calcolare l'insieme di prospezione  $\rho$  di  $q_{A,0}$ , si raccolgono i caratteri che possono seguire la stringa che deriva da  $A$ . Tali caratteri possono provenire da due casi: gli inizi del linguaggio  $L(p')$  riconosciuto partendo dallo stato  $p'$ ; oppure, se lo stato  $p'$  è finale o più in

generale, se  $L(p')$  contiene la stringa vuota, anche i caratteri dell'insieme  $\pi$  confluiscono nell'insieme  $\rho$ .

Per un insieme  $I$  di candidate, la chiusura è l'unione delle chiusure delle candidate appartenenti all'insieme  $I$ .

La macchina pilota è costruita in modo simile al caso  $LR(0)$ , ma ora i componenti dei macrostati sono le candidate e non i semplici stati.

*Algoritmo 4.52.* Macchina pilota dell'analisi  $LR(1)$

Si definisce un automa finito

$$N = (R, \Sigma \cup V, \vartheta, I_0, R)$$

che piloterà il parsificatore, così caratterizzato:

- l'insieme dei macrostati è  $R$ ;
- l'alfabeto è l'unione di quello terminale  $\Sigma$  e nonterminale  $V$  della grammatica;
- il macrostato  $I_0$  ha come contenuto iniziale la coppia  $\langle q_{ini}, \dashv \rangle$  (stato iniziale della rete, terminatore);
- i macrostati,  $R = \{I_0, \dots\}$  e la funzione di transizione  $\vartheta$  sono calcolati, partendo dal macrostato iniziale, per mezzo della procedura seguente:

$$I_0 = \text{chius}_1(\langle q_{ini}, \dashv \rangle);$$

$$R := \{I_0\};$$

repeat per ogni macrostato  $I_j \in R$  e per ogni simbolo  $X \in \Sigma \cup V$

$$\vartheta(I_j, X) := \text{chius}_1(\{\langle r, \pi \rangle\}) \text{ dove}$$

l'insieme delle candidate  $\langle r, \pi \rangle$  è così definito:

esiste nel macrostato  $I_j$  una candidata  $\langle q, \rho \rangle$  per la quale

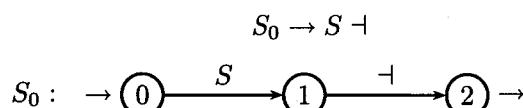
esiste nella rete l'arco  $\delta(q, X) = r$ ;

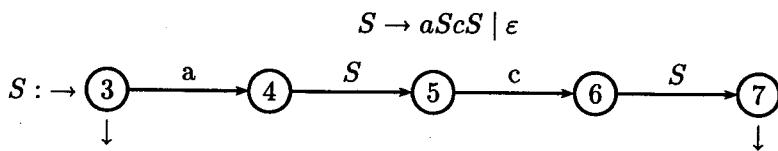
l'insieme di prospezione  $\pi$  è copiato da  $\rho$ , ossia  $\pi := \rho$ ;

if  $\vartheta(I_j, X) \notin R$  then  $R := R \cup \vartheta(I_j, X)$ ;

until nessun nuovo macrostato è stato prodotto dall'iterazione.

*Esempio 4.53.* Linguaggio di Dyck (es. 4.49) come  $LR(1)$ .



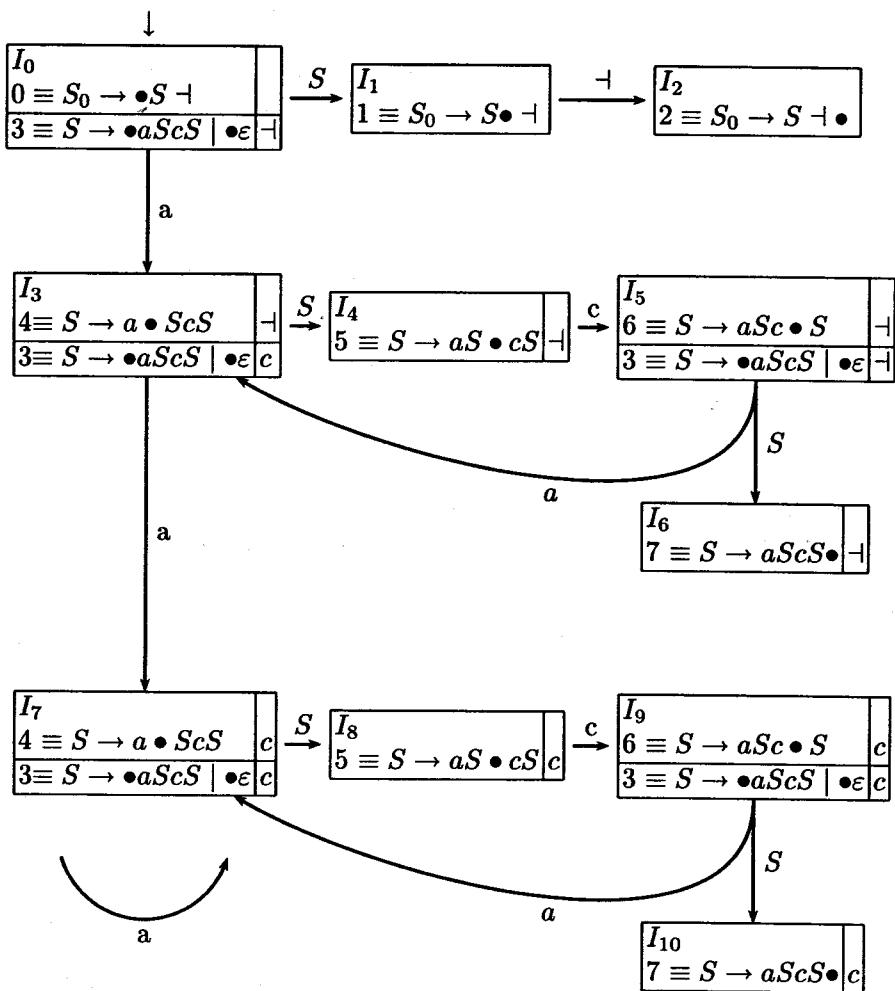


La macchina pilota  $LR(1)$  è sotto mostrata. La colonna di destra d'un macrostato contiene le prospezioni.

Per facilitare la correlazione visiva tra i macrostati e la grammatica, si possono riportare nel disegno del pilota alcune o tutte le regole, marcando il punto corrispondente allo stato.

Una regola grammaticale, come  $S \rightarrow aScS$ , è detta *marcata* se la parte destra contiene una marca (rappresentata dal tondino). Ad es. la regola marcata  $S \rightarrow a \bullet ScS$  denota lo stato 4 della macchina  $M_S$ .

La compresenza di stati e regole marcate è ridondante, perché la marca è soltanto un altro modo di individuare lo stato d'una macchina della rete.



Ciascun macrostato contiene il proprio nome e l'insieme delle candidate. Ad es. le due candidate di  $I_3$  sono  $\langle 4, \{\dashv\} \rangle$  e  $\langle 3, \{c\} \rangle$ .

Per agevolare la lettura, un tratto orizzontale separa le candidate calcolate dall'invocazione della funzione chiusura, ma l'ordine di scrittura delle candidate è irrilevante, poiché un macrostato è un insieme.

Un caso particolare sono le candidate (nei macrostati  $I_0, I_1, I_2$ ) della regola convenzionale  $S_0 \rightarrow S \dashv$ , in quanto essi non hanno prospezione.

Per illustrare il calcolo delle prospezioni, si consideri  $I_3$ . La coppia  $\langle 4, \dashv \rangle$  ha la stessa prospezione dello stato 3 di  $I_0$  da cui nasce grazie all'arco  $3 \xrightarrow{a} 4$ .

Applicando la chiusura alla coppia  $\langle 4, \dashv \rangle$  si ottiene lo stato 3 e la prospezione  $\{c\}$ , poiché  $Ini(L(5)) = \{c\}$  e  $L(5)$  non è annullabile.

Si noti che lo stato 3 è commentato da due regole marcate,  $S \rightarrow \bullet aScS$  e  $S \rightarrow \bullet \varepsilon$ ; la seconda, si può scrivere come  $S \rightarrow \varepsilon \bullet$ .

Si osserva la crescita del numero dei macrostati rispetto al pilota  $LR(0)$ , causata dalla presenza delle prospezioni. Infatti due macrostati, come  $I_3, I_7$ , che differiscono soltanto nelle prospezioni, sono fusi insieme nella macchina  $LR(0)$ .

### Condizione $LR(1)$

Una regola marcata è detta *completata* se la marca sta in fondo alla parte destra. Es.:

$$S \rightarrow \varepsilon \bullet \quad S \rightarrow aScS \bullet \quad S_0 \rightarrow S \dashv \bullet$$

Si riprende la classificazione dei macrostati del caso  $k = 0$ .

Una *candidata* è detta di *riduzione* se lo stato è finale in una macchina della rete, nel qual caso ad esso è associata una regola marcata completata.

Una *candidata* è detto di *spostamento*, se lo stato è l'origine di una freccia etichettata con un simbolo, terminale o non; ossia se la regola marcata contiene un simbolo a destra del pallino.

Ma la drastica condizione  $LR(0)$ , che proibiva ai macrostati di contenere riduzioni e spostamenti o riduzioni multiple, è sostituita da un controllo più fine sugli insiemi di prospezione.

**Definizione 4.54.** Condizione  $LR(1)$  Una grammatica soddisfa la condizione  $LR(1)$  se, per ogni macrostato della sua macchina pilota, valgono entrambe le condizioni:

1. assenza di conflitto riduzione-spostamento: ogni candidata di riduzione ha un insieme di prospezione disgiunto dall'insieme dei simboli terminali di spostamento (ossia dall'insieme delle etichette terminali delle frecce uscenti dal macrostato);
2. assenza di conflitto riduzione-riduzione: se vi sono due candidate di riduzione, i loro insiemi di prospezione sono disgiunti.

Una grammatica gode della proprietà  $LR(1)$  se ogni macrostato della macchina pilota soddisfa la condizione  $LR(1)$ .

**Esempio 4.55.** Linguaggio di Dyck (es. 4.53): verifica della condizione  $LR(1)$ . I macrostati  $I_1, I_2, I_4, I_6, I_8, I_{10}$ , aventi una sola candidata, soddisfano banalmente la condizione.

Ciascuno dei macrostati  $I_0, I_3, I_5, I_7, I_9$  possiede la candidata di riduzione  $S \rightarrow \bullet \varepsilon$ , la candidata di spostamento  $S \rightarrow \bullet aScS$  (entrambe associate allo stato 3), e un'altra candidata di spostamento. Segue la verifica della disgiunzione:

| macrostato | prospezione di $S \rightarrow \varepsilon$ | etichette uscenti | condizione                          |
|------------|--|-------------------|-------------------------------------|
| $I_0$      | $\vdash$                                   | $a$               | $\{\vdash\} \cap \{a\} = \emptyset$ |
| $I_3$      | $c$  | $a$               | $\{c\} \cap \{a\} = \emptyset$      |
| $I_5$      | $\vdash$                                   | $a$               | $\{\vdash\} \cap \{a\} = \emptyset$ |
| $I_7$      | $c$  | $a$               | $\{c\} \cap \{a\} = \emptyset$      |
| $I_9$      | $c$  | $a$               | $\{c\} \cap \{a\} = \emptyset$      |

La grammatica risulta  $LR(1)$ .

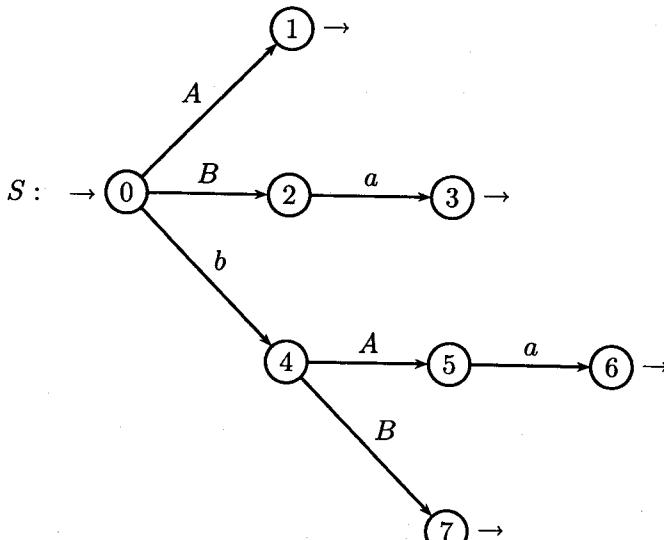
Il prossimo linguaggio (finito) illustra il ramo 2 della condizione.

*Esempio 4.56.* Condizione  $LR(1)$  con due riduzioni nel macrostato.  
La grammatica e la rete sono:

$$S \rightarrow A \mid Ba \mid bAa \mid bB$$

$$A \rightarrow a$$

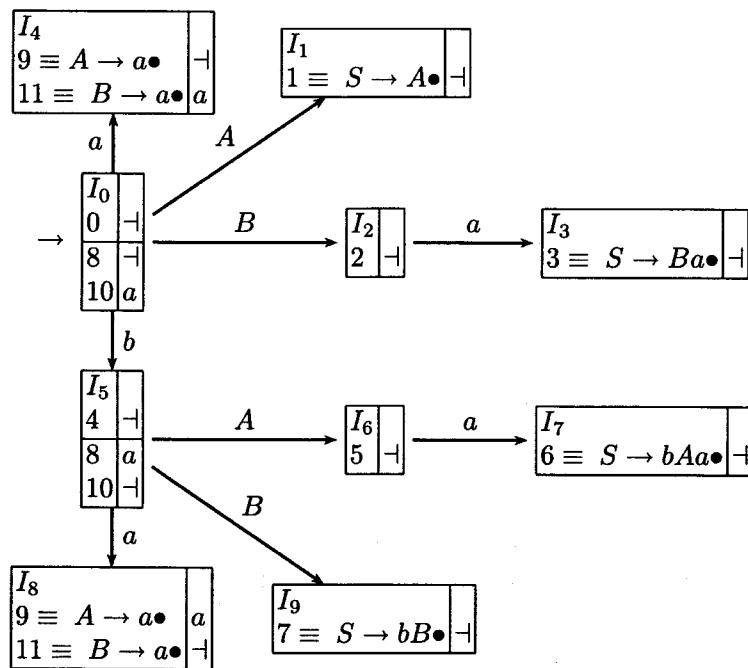
$$B \rightarrow a$$



$$A : \rightarrow 8 \xrightarrow{a} 9 \rightarrow$$

$$B : \rightarrow 10 \xrightarrow{a} 11 \rightarrow$$

Per non complicare il disegno, nell'automa pilota  $LR(1)$  si scrivono, soltanto negli stati di riduzione, le regole marcate:



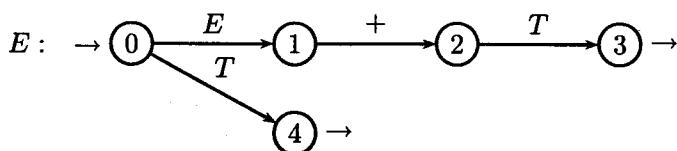
I macrostati che potrebbero violare la condizione sono  $I_4$  e  $I_8$ , mentre gli altri soddisfano già la condizione  $LR(0)$ . Ciascuno dei due macrostati contiene due candidate di riduzione, ma i loro insiemi di prospezione sono disgiunti. Per altro, poiché da nessuno dei due escono archi, la parte 1 della condizione è manifestamente soddisfatta.

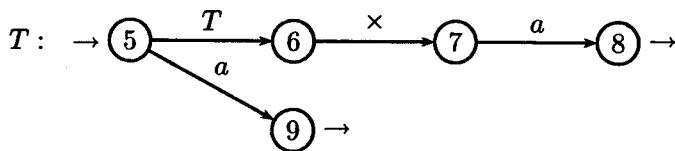
 La grammatica risulta  $LR(1)$ , ma non  $LR(0)$ , perché in  $I_4$  e in  $I_8$  vi sono due riduzioni, tra le quali la scelta, senza prospettiva, sarebbe indeterministica.

Il prossimo esempio chiarisce ulteriormente il ruolo degli insiemi di prospezione.

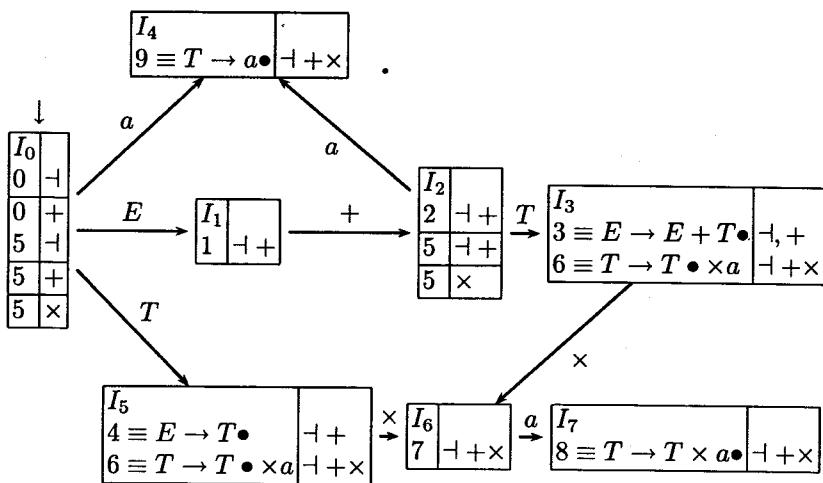
*Esempio 4.57. Espressioni aritmetiche, condizione  $LR(1)$ .*

$$E \rightarrow E + T \mid T \quad T \rightarrow T \times a \mid a$$





Macchina pilota  $LR(1)$ :



La grammatica risulta  $LR(1)$ . Infatti nessun macrostato contiene due stati finali e, nei macrostati misti riduzione/spostamento, la condizione di disgiunzione è soddisfatta, come sotto verificato:

$$I_3 : \times \notin \{\dashv, +\}, \text{ prospezione della riduzione } E \rightarrow E + T$$

$$I_5 : \times \notin \{\dashv, +\}, \text{ prospezione della riduzione } E \rightarrow T$$

Si noti che in  $I_3$  gli insiemi di prospezione della candidata di spostamento  $6 \equiv T \rightarrow T \bullet \times a$  e di quella di riduzione  $3 \equiv E \rightarrow E + T \bullet$  non sono disgiunti:  $\{\dashv, +, \times\} \cap \{\dashv, +\} \neq \emptyset$ , ma ciò non viola la condizione  $LR(1)$ . Infatti l'insieme di prospezione d'una candidata di spostamento non è argomento del predicato di verifica, ma serve soltanto per calcolare gli insiemi di prospezione dei macrostati successivi.

Pertanto, una volta che la macchina pilota è stata costruita, gli insiemi di prospezione delle candidate di spostamento possono essere eliminati, perché non servono al parsificatore.

### Condizione *LALR(1)*

Si considera brevemente una condizione di determinismo intermedia tra quelle *LR(0)* e *LR(1)*, che ha conosciuto ampia diffusione quando la ridotta memoria dell'elaboratore rendeva sconsigliabile l'uso del metodo *LR(1)* a causa del numero elevato di macrostati del pilota.

Riprendendo il caso d'una grammatica *LR(1)* ma non *LR(0)*, si immagini di semplificare il pilota *LR(1)*, fondendo insieme i macrostati che sono indistinguibili nel pilota *LR(0)*, ossia quelli le cui candidate differiscono soltanto nella seconda componente, la prospezione. Al contempo si desidera preservare le informazioni sulla prospezione. Più precisamente, quando si fondono due macrostati, si uniscono gli insiemi di prospezione delle candidate che coincidono nella prima componente (ossia lo stato). Il grafo della macchina pilota<sup>19</sup> *LALR(1)*<sup>20</sup> così ottenuta è isomorfo a quello del pilota *LR(0)*. Esso può però contenere macrostati misti o con riduzioni plurime, purché valga la condizione seguente, identica a quella del caso *LR(1)*.

Una grammatica soddisfa la *condizione LALR(1)* se, per ogni macrostato dell'automa pilota *LALR(1)*, valgono entrambe le condizioni:

1. ogni candidata di riduzione ha un insieme di prospezione disgiunto dall'insieme delle etichette terminali degli archi uscenti dal macrostato;
2. se vi sono due candidate di riduzione, i loro insiemi di prospezione sono disgiunti.

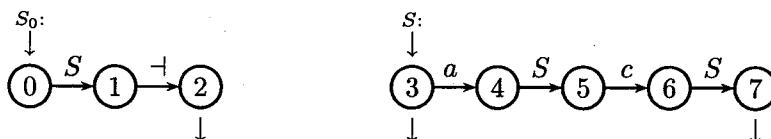
Dalla definizione segue immediatamente che la famiglia delle grammatiche *LALR(1)* è inclusa in quella delle grammatiche *LR(1)* e include la famiglia *LR(0)*.

Anche per i linguaggi valgono le stesse inclusioni strette: esistono linguaggi *LR(1)* ma non *LALR(1)*, e linguaggi *LALR(1)* ma non *LR(0)*.

Alcuni casi sono illustrati dai prossimi esempi.

*Esempio 4.58.* Linguaggio di Dyck 4.53 come *LALR(1)*.

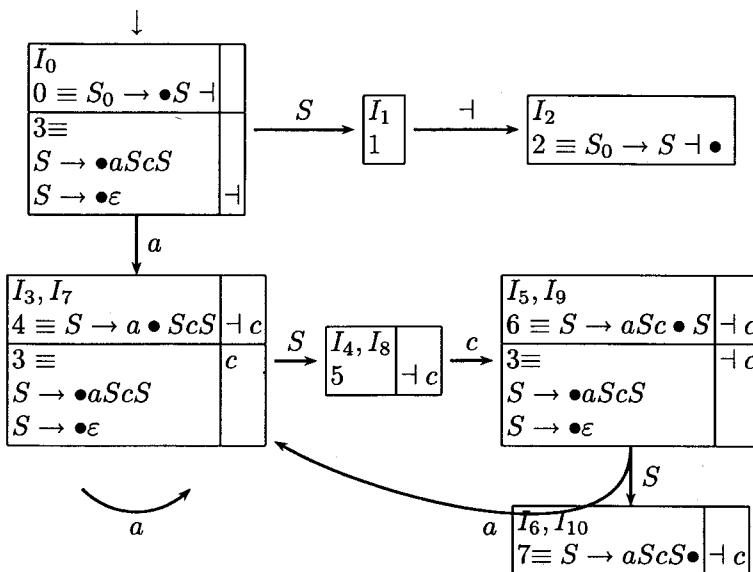
La grammatica è nota:



Fondendo i macrostati del pilota *LR(1)* (di p. 215), non distinguibili nel pilota *LR(0)* (di p. 210), si ottiene il pilota *LALR(1)*:

<sup>19</sup>Per la costruzione del pilota *LALR(1)* esistono algoritmi diretti, che evitano di passare attraverso il pilota *LR(1)*.

<sup>20</sup>Sigla di *Look Ahead LR(1)*.



il cui grafo è isomorfo a quello del pilota  $LR(0)$ .

Nessun macrostato contiene più riduzioni, ma vi sono i macrostati misti  $I_0, [I_3, I_7], [I_5, I_9]$ , che fanno cadere il metodo  $LR(0)$ . Nei macrostati misti, l'insieme di prospezione della riduzione  $S \rightarrow \epsilon$  è disgiunto dalle etichette degli archi uscenti, e la grammatica risulta  $LALR(1)$ .

Questo pilota svolge, con un numero minore di stati, lo stesso compito del pilota  $LR(1)$ .

Il prossimo è un esempio in cui i piloti  $LR(1)$  e  $LALR(1)$  coincidono.

*Esempio 4.59.* Espressioni aritmetiche (es. 4.57) come  $LALR(1)$ .

La macchina pilota  $LR(1)$  di p. 219 è già isomorfa a quella  $LR(0)$ , perché non contiene macrostati identici a meno degli insiemi di prospezione. Ciò significa che la grammatica

$$E \rightarrow E + T \mid T \quad T \rightarrow T \times a \mid a$$

soddisfa la condizione  $LALR(1)$ .

*Esempio 4.60.* Esempio  $LR(1)$  non  $LALR(1)$ .

Nell'es. 4.56 (p. 217), fondendo insieme i macrostati  $I_4$  e  $I_8$ , indistinguibili senza la prospezione, si ottiene il macrostato del pilota  $LALR(1)$ :

|                                     |             |
|-------------------------------------|-------------|
| $I_4, I_8$                          |             |
| $9 \equiv A \rightarrow a \bullet$  | $\dashv, a$ |
| $11 \equiv B \rightarrow a \bullet$ | $\dashv, a$ |

Poiché le due riduzioni hanno insiemi di prospezione sovrapposti (identici), il macrostato non è  $LALR(1)$ .

Molte esperienze di progetto dei compilatori hanno dimostrato che il metodo  $LALR(1)$  è spesso adatto alla costruzione dei parsificatori, di solito al costo di piccole modifiche della grammatica di riferimento del linguaggio.

#### 4.5.5 Algoritmo di parsificazione $LR(1)$

Costruito il pilota  $LR(1)$  o  $LALR(1)$ , il parsificatore agisce come quello  $LR(0)$  (p. 207), con in più l'esame della prospezione per scegliere la mossa nei macrostati misti o aventi riduzioni multiple.

*Algoritmo 4.61.* Costruzione dell'analizzatore a spostamento e riduzione con prospezione.

Sia  $G$  una grammatica  $LR(1)$  (o  $LALR(1)$ ) e

$$N = (R, \Sigma \cup V, \vartheta, I_0, R)$$

la sua macchina pilota. Si costruisce l'automa a pila  $A$  come segue:

*Alfabeto di pila:*  $R \cup \Sigma \cup V$ ;

*Insieme degli stati* di  $A$ : irrilevante perché contiene un solo stato;

*Configurazione iniziale*: la pila contiene  $I_0$ , il macrostato iniziale;

*Mosse dell'automa*: Sia  $I$  il macrostato in cima alla pila e  $a$  il carattere corrente. Due sono i tipi di mosse.

*Mossa di spostamento*: se per il macrostato  $I$  è definita nel pilota la mossa  $\vartheta(I, a) = I'$ , l'automa legge  $a \in \Sigma$ , avanzando la testina, e impila la stringa  $aI'$ ;

*Mossa di riduzione*: se il macrostato corrente, qui denotato  $I_n$ , contiene la candidata di riduzione

$$q, \equiv B \rightarrow X_1 X_2 \dots X_n \bullet, \pi$$

(ossia è  $riduz(q) = \{B \rightarrow X_1 X_2 \dots X_n\}$ ) dove  $n \geq 0$  è la lunghezza della parte destra, e

se il carattere  $a$  appartiene all'insieme di prospezione  $\pi$ , si compiono le operazioni seguenti.

In cima alla pila si trova necessariamente una stringa  $\beta'$

$$I' \overbrace{X_1 I_1 X_2 I_2 \dots X_n I_n}^{\beta'}$$

contenente  $n$  macrostati inseriti da mosse precedenti.

La riduzione è una mossa spontanea che, prima cancella dalla pila la stringa  $\beta'$  (ossia gli ultimi  $2n$  simboli), poi inserisce la stringa  $BI''$ , dove  $I'' = \vartheta(I', B)$  è il macrostato raggiunto "leggendo" il nonterminale  $B$ .

*Configurazione di riconoscimento:* pila =  $I_0$ , ingresso =  $-$

Se nessuna mossa è possibile, la stringa è rifiutata.

Si noti che, grazie all'ipotesi  $LR(1)$ , il macrostato posto in cima e il carattere corrente selezionano univocamente la mossa: spostamento, o riduzione con una ben precisa candidata di riduzione, o rifiuto. Di conseguenza il funzionamento è deterministico.

Segue la traccia dell'analisi sintattica d'una stringa.

*Esempio 4.62.* Es. 4.57: traccia dell'analisi della stringa  $a + a$ .

Per le espressioni aritmetiche con pilota  $LR(1)$  o  $LALR(1)$  (p. 219) si ha:

| Pila  | x                 | Commento   |
|-------|-------------------|--|
| $I_0$ | $a$               | + $\dashv$ sposta  |
| $I_0$ | $aI_4$            | + $\dashv$ riduci con $T \rightarrow a$                            |
| $I_0$ | $TI_5$            | + $\dashv$ $\in$ prospezione di 4:<br>riduci con $E \rightarrow T$ |
| $I_0$ | $EI_1$            | + $\dashv$ sposta  |
| $I_0$ | $EI_1 + I_2 a$    | $\dashv$ sposta  |
| $I_0$ | $EI_1 + I_2 aI_4$ | $\dashv$ riduci con $T \rightarrow a$                              |
| $I_0$ | $EI_1 + I_2 TI_3$ | $\dashv$ riduci con $E \rightarrow E + T$                          |
| $I_0$ | $\dashv$          | accetta  |

#### 4.5.6 Proprietà delle sottofamiglie deterministiche e confronti

Il confronto teorico tra le varie sottofamiglie dei linguaggi deterministicci è troppo articolato per essere qui discusso in modo completo. Ci si limita all'enunciazione delle proprietà più fondamentali, cominciando dalla famiglia  $LR(k)$  e proseguendo poi con le famiglie dei linguaggi regolari e di quelli  $LL(k)$ .

#### Proprietà dei linguaggi e delle grammatiche $LR(k)$

Per completare il quadro teorico, si enunciano, senza dimostrazione, alcune proprietà, in parte già accennate, riguardanti le grammatiche  $LR(k)$  e i loro linguaggi.<sup>21</sup>

<sup>21</sup>Per le dimostrazioni e gli approfondimenti si rimanda a [24, 46, 47].

*Proprietà 4.63.* La famiglia  $DET$  dei linguaggi deterministici (che si ricorda sono quelli accettati a stato finale dagli automi a pila deterministici) coincide con quella dei linguaggi generati dalle grammatiche  $LR(1)$ .

Ovviamente questo non implica che ogni grammatica, il cui linguaggio è deterministico, sia necessariamente  $LR(1)$ : infatti essa potrebbe essere ambigua o richiedere una prospezione di lunghezza  $k > 1$ . Ma esisterà una grammatica equivalente, che gode della proprietà  $LR(1)$ .

Affidandosi all'intuizione del lettore, si immagini di allungare la prospezione a valori  $k > 1$  impiegando un riconoscitore  $LR(k)$ . Evidentemente esso non è altro che un automa a pila deterministico, il cui pilota contiene più macrostati del pilota  $LR(1)$ , nati dalla differenziazione delle prospettive. Dalla precedente proprietà discende la prossima.

*Proprietà 4.64.* La famiglia dei linguaggi generati dalle grammatiche  $LR(k)$ , per ogni  $k > 1$ , coincide con la famiglia dei linguaggi generati dalle grammatiche  $LR(1)$ , dunque con la famiglia  $DET$  dei linguaggi deterministici.

Un linguaggio libero ma indeterministico, come quelli studiati a p. 163, non può quindi avere una grammatica  $LR(k)$ .

Se ogni linguaggio deterministico è generabile da una grammatica  $LR(1)$ , a che serve considerare grammatiche con un parametro  $k$  maggiore? Per rispondere si passa ora al confronto tra le grammatiche, anziché tra i linguaggi. Vi sono grammatiche  $LR(2)$  ma non  $LR(1)$ , e più in generale vale l'enunciato seguente.

*Proprietà 4.65.* Per ogni valore di  $k \geq 1$ , esistono grammatiche che soddisfano la condizione  $LR(k)$ , ma non la condizione  $LR(k-1)$ .

Anche se la proprietà 4.64 assicura che ogni grammatica  $LR(k)$ ,  $k > 1$ , può essere sostituita da una grammatica  $LR(1)$  equivalente, la seconda può risultare meno naturale della prima (si vedranno tra breve degli esempi in 4.5.7).

Per ultimo si cita un risultato teorico negativo.

*Proprietà 4.66.* Data una grammatica, non è decidibile se esiste un intero  $k > 0$ , per cui essa risulti  $LR(k)$ ; di conseguenza non è decidibile se il linguaggio generato da una grammatica libera è deterministico.

Se però il valore di  $k$  è fissato, per es. a 1, si può decidere se la grammatica è  $LR(k)$ , costruendo il pilota e verificando che non presenti macrostati inadeguati.

### Confronti tra $REG$ , $LL(k)$ e $LR(k)$

Può interessare il confronto tra vari modelli teorici di famiglie di linguaggi deterministici. I modelli considerati sono i linguaggi regolari  $REG$ , definiti da espressioni regolari, automi finiti o grammatiche unilineari; i linguaggi  $LR(k)$ , con le loro varianti ( $LR(0)$  e  $LALR(1)$ ); e i linguaggi  $LL(k)$ . Quanto

alle grammatiche, si fa qui l'ipotesi che esse non siano estese con espressioni regolari.

Si inizia con il confronto tra linguaggi regolari e  $LL(1)$ .

*Proprietà 4.67.* Ogni linguaggio regolare è generato da una grammatica  $LL(1)$ .

La dimostrazione è semplice. Si può supporre che il linguaggio regolare sia definito da un automa finito deterministico, dunque la rete contiene una sola macchina, i cui archi portano etichette terminali soltanto. La condizione  $LL(1)$  (p. 182) è soddisfatta per le seguenti ragioni: se due frecce escono da uno stato, esse portano etichette terminali diverse; se uno stato è finale, la freccia di terminazione ha come insieme guida il delimitatore  $\dashv$ , il quale non può etichettare nessun arco della macchina.

Il confronto tra i casi  $LL(k)$  e  $LR(k)$  risulta più articolato, in quanto dipende dal considerare le grammatiche o i linguaggi, nonché dal valore del parametro  $k$ .

Alcuni fatti noti possono essere così riassunti.

- Ogni linguaggio  $LL(k)$  con  $k \geq 1$ , risulta deterministico, cioè  $LR(1)$  per la proprietà 4.63. Infatti il parsificatore d'un linguaggio  $LL(k)$  è un automa deterministico a pila.
- Vi sono linguaggi deterministici che non sono definibili con una grammatica  $LL(k)$ , per nessun valore di  $k$ .

Basta ricordare gli esempi 4.39 e 4.40 (p. 196, 197).

Dai linguaggi si passa ora al confronto tra le grammatiche  $LR$  e  $LL$ , a pari valore della lunghezza di prospezione. La seguente inclusione testimonia la maggiore potenza espressiva delle grammatiche  $LR$  rispetto alle  $LL$ .

*Proprietà 4.68.* Per ogni valore  $k \geq 1$ , se una grammatica soddisfa la condizione  $LL(k)$  essa soddisfa anche la condizione  $LR(k)$ .<sup>22</sup>

Ciò si giustifica, considerando per semplicità il caso  $k = 1$ . Si guardi a p. 213 la grammatica del linguaggio di Dyck, che è evidentemente  $LL(1)$ , e l'automa pilota  $LR(1)$  ivi disegnato. Esso ha la particolarità che ogni macrostato contiene un solo stato nella sezione che sta sopra alla riga orizzontale. Ciò significa che, durante la costruzione del pilota, prima dell'operazione di chiusura  $textchius_1$ , ogni macrostato contiene un solo stato. Questa proprietà scende necessariamente dall'essere verificata la condizione  $LL(1)$ . Essa, unita alla condizione che gli insiemi guida delle alternative sono disgiunti, ha come conseguenza che non vi possono essere conflitti in nessun macrostato.

Se si prendono lunghezze di prospezione diverse, 0 e 1, si trova che le due classi di grammatiche  $LR(0)$  e  $LL(1)$  non sono incluse una nell'altra. Infatti si ricorda che:

---

<sup>22</sup>La dimostrazione si può trovare in [47].

1. una grammatica con regole epsilon non è  $LR(0)$  ma può essere  $LL(1)$ ;
2. una grammatica con ricorsioni a sinistra non è  $LL(1)$  ma può essere  $LR(0)$ .

Per confrontare i linguaggi  $LL(1)$  e  $LR(0)$  si richiamano dei fatti già noti:

1. un linguaggio contenente delle frasi, i cui prefissi appartengono al linguaggio, non è  $LR(0)$  ma può essere  $LL(1)$ ;
2. il linguaggio  $\{a^*a^n b^n \mid n \geq 0\}$  è  $LR(0)$  ma non è  $LL(1)$  (p. 196).

Di conseguenza, le famiglie dei linguaggi  $LL(1)$  e  $LR(0)$  sono distinte e incomparabili.

Il confronto tra le classi  $LL(1)$  e  $LALR(1)$  è un po' sottile:<sup>23</sup> basti dire che quasi tutte le grammatiche  $LL(1)$ , con l'eccezione di certi casi di interesse solo teorico, sono anche  $LALR(1)$ ; in altre parole la famiglia  $LALR(1)$  è in pratica più ampia della famiglia  $LL(1)$ .

#### 4.5.7 Come ottenere grammatiche $LR(1)$

Le grammatiche dei linguaggi tecnici soddisfano molto spesso la condizione  $LR(1)$ , ma talvolta è necessario trasformarle per poter progettare un parsificatore deterministico. Si immagini dunque di trovarsi di fronte a una grammatica non ambigua, ma che viola la condizione  $LR(1)$ . I casi da considerare sono due: la grammatica è  $LR(k)$  ma con prospezione maggiore di uno; la grammatica non è  $LR(k)$ .

#### Grammatica $LR(2)$ con conflitto riduzione-riduzione per $k = 1$

Ponendosi nel primo caso, si supponga che la grammatica sia  $LR(2)$  ma non  $LR(1)$ . La violazione per  $k = 1$  si ha in particolare se in qualche macrostato  $I$  del pilota vi sono due candidate di riduzione

$$A \rightarrow \alpha \bullet, \{a\} \quad B \rightarrow \beta \bullet, \{a\}$$

con lo stesso primo carattere di prospezione  $a$ , ma discriminate dal secondo carattere.

La modifica da farsi è detta scansione anticipata; essa allunga le parti destre delle regole, appendendo ad esse il carattere comune, in modo che nella grammatica modificata le due regole vengano a avere i secondi caratteri negli insiemi di prospezione.

Più precisamente l'intervento introduce due nuovi nonterminali e le regole

$$\langle Aa \rangle \rightarrow \alpha a \quad \langle Ba \rangle \rightarrow \beta a$$

Naturalmente si devono anche aggiustare le regole contenenti  $A$  o  $B$ , per preservare l'equivalenza della grammatica.

La scansione anticipata deve garantire che esiste la derivazione

---

<sup>23</sup>Si rimanda a [6].

$$\langle Aa \rangle \stackrel{+}{\Rightarrow} \gamma a$$

se, e solo se, nella grammatica iniziale esiste la derivazione

$$A \stackrel{+}{\Rightarrow} \gamma$$

e se il carattere  $a$  può seguire  $A$ .

*Esempio 4.69.* Scansione anticipata.

La grammatica  $G_1$

$$\begin{array}{lll} S \rightarrow Abb & A \rightarrow aA & B \rightarrow aB \\ S \rightarrow Bbc & A \rightarrow a & B \rightarrow a \end{array}$$

ha il conflitto tra  $A \rightarrow a \bullet \{b\}$  e  $B \rightarrow a \bullet \{b\}$ . Passando a  $k = 2$ , la prima riduzione ha la prospezione  $\{bb\}$  e la seconda ha la prospezione disgiunta  $\{bc\}$ . La scansione anticipata produce la grammatica equivalente  $LR(1)$ :

$$\begin{array}{lll} S \rightarrow \langle Ab \rangle b & \langle Ab \rangle \rightarrow a \langle Ab \rangle & \langle Bb \rangle \rightarrow a \langle Bb \rangle \\ S \rightarrow \langle Bb \rangle c & \langle Ab \rangle \rightarrow ab & \langle Bb \rangle \rightarrow ab \end{array}$$

### Grammatica LR(2) con conflitto riduzione-spostamento per $k = 1$

Il conflitto più immediato da curare si presenta quando in un macrostato  $I$  stanno due candidate

$$A \rightarrow \alpha \bullet a\beta, \pi \quad B \rightarrow \gamma \bullet, \{a\}$$

confittuali, perché dal macrostato esce l'arco etichettato  $a$  ma  $a$  appartiene all'insieme di prospezione della riduzione.

Intervento: consiste nell'introdurre un nuovo nonterminale  $\langle Ba \rangle$  che equivale a  $B$  seguito da  $a$ , sostituendo la seconda regola con la  $\langle Ba \rangle \rightarrow \gamma a$ . Di conseguenza vanno aggiustate le regole aventi  $B$  nella parte destra, in modo da preservare l'equivalenza della grammatica.

Essendo per ipotesi la grammatica  $LR(2)$ , l'insieme di prospezione associato a  $\langle Ba \rangle \rightarrow \gamma a$  è sicuramente disgiunto da  $a$ , e il conflitto scompare.

Il problema si complica un poco quando il carattere  $a$  di prospezione, che provoca il conflitto tra le candidate  $A \rightarrow \alpha \bullet a\beta, \pi$  e  $B \rightarrow \gamma \bullet, \{a\}$ , proviene da una derivazione d'un nonterminale  $C$ , il quale segue immediatamente  $B$  in una forma di frase:

$$S \stackrel{+}{\Rightarrow} \dots BC \dots \stackrel{+}{\Rightarrow} \dots Ba \dots \dots$$

Intervento: si crea un nuovo nonterminale denominato  $\langle a/S C \rangle$ , che deve generare le stesse stringhe generate da  $C$ , ma decurtate del prefisso  $a$ , garantendo la condizione:

$$\langle a/S C \rangle \stackrel{+}{\Rightarrow} \gamma \text{ se, e solo se, } C \stackrel{+}{\Rightarrow} a\gamma \text{ nella grammatica originale.}$$

Poi si aggiusta la grammatica in modo da preservare l'equivalenza. Questa trasformazione è detta *quoziante sinistro*, e si basa sull'operazione /<sub>S</sub> definita a p. 17.

Alla grammatica così modificata, si può infine applicare la scansione anticipata, che elimina il conflitto.

*Esempio 4.70.* La grammatica  $G_2$

$$\begin{array}{llll} S \rightarrow AC & A \rightarrow a & C \rightarrow c & D \rightarrow d \\ S \rightarrow BD & B \rightarrow ab & C \rightarrow bC \end{array}$$

è  $LR(2)$  ma non  $LR(1)$  a causa del conflitto riduzione spostamento

$$A \rightarrow a \bullet \{b, c\} \quad B \rightarrow a \bullet b\{d\}$$

Dopo la trasformazione con il quoziante sinistro, si ottiene

$$\begin{array}{lll} S \rightarrow Ab\langle b/S C \rangle & A \rightarrow a & \langle b/S C \rangle \rightarrow b\langle b/S C \rangle \\ S \rightarrow Ac\langle c/S C \rangle & B \rightarrow ab & \langle b/S C \rangle \rightarrow c \\ S \rightarrow BD & D \rightarrow d & \langle c/S C \rangle \rightarrow \varepsilon \end{array}$$

Ora il conflitto diventa eliminabile con la scansione anticipata.

*Grammatica non  $LR(k)$*

Diversamente dai casi precedenti, non si possono dare delle trasformazioni sistematiche, ma la grammatica va ristrutturata, studiando i linguaggi generati da ogni nonterminale, identificando le cause dei conflitti, e modificando le corrispondenti sottogrammatiche, in modo da renderle  $LR(1)$ .

Una modifica talvolta efficace è la trasformazione della ricorsione da sinistra a destra. La ragione è che gli algoritmi  $LR$  sono in grado di portare avanti più calcoli non deterministici soltanto fino al momento in cui uno di essi si conclude con una riduzione, che deve essere deterministica. Una regola (più in generale una derivazione) ricorsiva a destra permette di rinviare il momento della decisione, mentre una regola ricorsiva a sinistra lo anticipa. In altre parole, l'automa nel caso destro accumula maggiori informazioni nella pila, durante il calcolo che precede la prima riduzione da compiere.<sup>24</sup>

*Esempio 4.71.* Rovesciamento della ricorsione.

Il linguaggio  $a^+b^+ \cup \{a^n b^n b^* c \mid n \geq 1\}$  è generato dalla grammatica  $G_s$

$$\begin{array}{llll} S \rightarrow X & S \rightarrow Yc \\ X \rightarrow aX & X \rightarrow aB & Y \rightarrow Yb & Y \rightarrow Z \\ B \rightarrow bB & B \rightarrow b & Z \rightarrow aZb & Z \rightarrow ab \end{array}$$

<sup>24</sup> Il fatto che la pila si allunga maggiormente durante l'analisi con grammatiche ricorsive a destra aumenta l'occupazione della memoria, cosa di solito trascurabile nel contesto tecnologico moderno.

La macchina pilota  $LR(1)$  ha un conflitto tra una riduzione e due spostamenti:

$$Z \rightarrow ab\bullet, \{b, c\} \quad \text{e} \quad B \rightarrow \bullet bB, \{\dashv\} \quad B \rightarrow \bullet b, \{\dashv\}$$

Non servirebbe incrementare  $k$  per eliminare il conflitto; ad es. con  $k = 2$ , la stringa  $bb$  è compatibile sia con la riduzione, sia con lo spostamento.

Riscrivendo le regole di  $X$  e di  $B$  in forma lineare a sinistra, i linguaggi generati dai due nonterminali restano invariati, ma l'automa effettua anticipatamente gli spostamenti e soltanto alla fine le riduzioni. La pila così si allunga e può mantenere aperte tutte le vie, fino al momento in cui la presenza della  $c$  o del terminatore rende palese la scelta corretta.

La grammatica equivalente  $G_d$

$$\begin{array}{ll} S \rightarrow X & S \rightarrow Yc \\ X \rightarrow Xb & X \rightarrow Ab \\ A \rightarrow Aa & A \rightarrow a \end{array} \quad \begin{array}{ll} Y \rightarrow Yb & Y \rightarrow Z \\ Z \rightarrow aZb & Z \rightarrow ab \end{array}$$

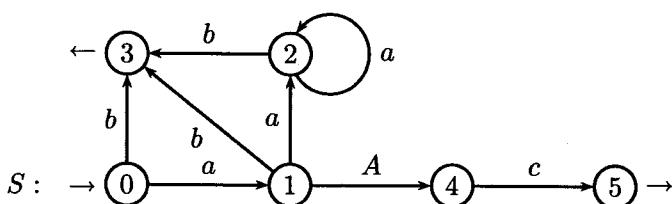
risulta  $LR(1)$ .

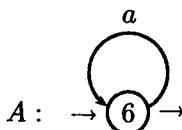
#### 4.5.8 Analisi sintattica $LR(1)$ con grammatiche estese

Anche questi parsificatori possono operare con grammatiche BNF estese, a patto di aggiungere alcune informazioni nella pila, allo scopo di effettuare correttamente le riduzioni. Infatti per una grammatica estesa, quando la macchina pilota entra in un macrostato di riduzione, la parte destra della regola da ridurre non è sempre univocamente determinata, a causa della presenza degli iteratori e delle alternative nella grammatica. La situazione è illustrata dal prossimo esempio.

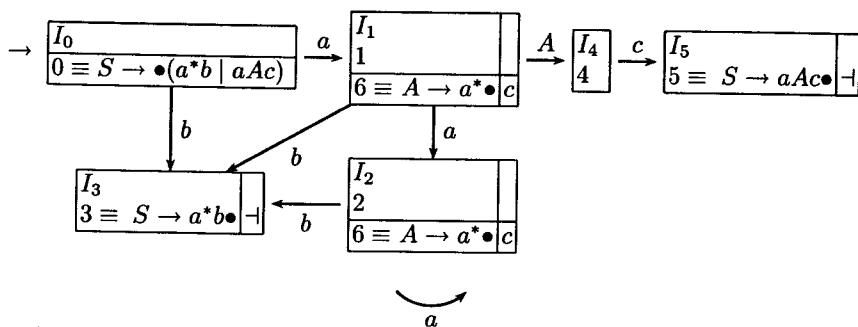
*Esempio 4.72.* Grammatica estesa.

$$\begin{aligned} R_1 : S &\rightarrow a^*b \mid aAc \\ R_2 : A &\rightarrow a^* \end{aligned}$$





Si disegna la macchina pilota  $LR(1)$ :



Al solito, ogni macrostato contiene stati (commentati per comodità di lettura da regole marcate). Gli insiemi di prospezione, calcolati con il metodo  $LR(1)$  (basterebbe il metodo  $LALR(1)$ ), sono riportati accanto alle riduzioni, nei macrostati in cui vi sono riduzioni.

Ora sia data la stringa  $x_1 = aaac \dashv$ . Dopo la lettura del prefisso  $aa$ , la configurazione dell'automa è:

$\overbrace{I_0 a I_1 a I_2}^{\text{pila}} \mid \overbrace{ac \dashv}^{\text{suffisso}}$

e nel macrostato  $I_2$  si sceglie, grazie alla prospezione  $a$ , lo spostamento:

$\overbrace{I_0 a I_1 a I_2 a I_2}^{\text{pila}} \mid \overbrace{c \dashv}^{\text{suffisso}}$

Nel macrostato  $I_2$  si deve scegliere la riduzione  $A \rightarrow a^*$ , in accordo con la prospezione  $c$ .

Ora nasce un nuovo problema, che non esisteva per le grammatiche non estese. Quante sono le coppie di simboli da disimpilare con la mossa di riduzione? A prima vista, osservando che la stringa letta è  $aa$ , vi sono più scelte possibili: zero (caso  $A \Rightarrow \epsilon$ ), una (caso  $a$ ) o due (caso  $a^2$ ).

Ma tale incertezza renderebbe indeterministico il funzionamento dell'automa. Similmente l'analisi della stringa  $x_2 = aaab \dashv$  porta l'automa a pila nella configurazione:

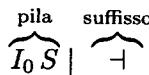
$\overbrace{I_0 a I_1 a I_2 a I_2 b I_3}^{\text{pila}} \mid \overbrace{\dashv}^{\text{suffisso}}$

Il macrostato corrente  $I_3$  prescrive l'operazione di riduzione con la regola  $S \rightarrow a^*b$ , la quale, a causa della presenza della stella, di nuovo non dice quanti simboli siano da disimpilare.

Esaminando la pila dalla cima, la mossa di riduzione dovrebbe controllare che i simboli (terminali o nonterminali) incontrati appartengano al linguaggio definito dalla espressione regolare riflessa  $(a^*b)^R$ . In altre parole, la stringa di tali simboli deve essere riconosciuta dalla macchina, ottenuta dalla macchina  $M_A$  invertendo il verso delle frecce, partendo dallo stato 3, quello che corrisponde alla riduzione da eseguire.

In questo caso, il prefisso letto è  $aaab$ , riflesso in  $baaa$ , e le stringhe  $b, ba, baa, baaa$  soddisfano tale condizione. Il numero di simboli terminali disimpilabili varia da uno a quattro.

La scelta giusta è quattro, che produce la configurazione



che accetta la stringa data.

Per risolvere l'incertezza sulla lunghezza della stringa da disimpilare in fase di riduzione sono stati proposti tanti metodi diversi,<sup>25</sup> tra i quali si presenta ora uno dei più semplici.

### Metodo di Morimoto e Sassa per il controllo delle riduzioni

L'idea è di suddividere le operazioni di spostamento del pilota in due categorie, dette di *apertura* e di *proseguimento*, a seconda che esse corrispondano a una mossa che inizia la ricerca della parte destra d'una regola, oppure a una mossa che prosegue tale ricerca.

Le operazioni di spostamento-apertura inseriscono sulla pila un'informazione aggiuntiva: l'*etichetta* della regola (ovvero il nome della macchina) che inizia la scansione della parte destra.

Invece le operazioni di spostamento-proseguimento non ne hanno bisogno e agiscono esattamente come gli spostamenti dell'algoritmo  $LR(1)$  per le grammatiche non estese.

Quando si deve compiere una riduzione, le etichette presenti nella pila permetteranno di rendere deterministica la scelta del numero di simboli da disimpilare.

Passando alla realizzazione, si descrive la costruzione del pilota  $LR(1)$  di Morimoto e Sassa.

Gli stati presenti in un macrostato si ripartiscono in due insiemi (al più uno dei quali può essere vuoto) il *nucleo* e il *resto*, così caratterizzati:

1. nel macrostato iniziale  $I_0$  tutti gli stati appartengono al resto, ossia il nucleo è vuoto;

---

<sup>25</sup>Si veda la rassegna in [35].

2. se nel pilota vi è la mossa  $I \xrightarrow{X} I'$ , il nucleo di  $I'$  contiene gli stati  $q'$  tali che, esiste uno stato  $q \in I$  per il quale, in qualche macchina della rete, vi è l'arco  $\delta(q, X) = q'$ :

$$\text{nucleo}(I') = \{q' \mid \exists I \text{ tale che } q \in I \wedge q' = \delta(q, X)\}$$

3. il resto d'un macrostato  $I$  (non iniziale) contiene gli stati  $r$  ottenuti mediante l'operazione di chiusura, a partire da uno stato  $q$  presente nel nucleo di  $I$ :

$$\text{resto}(I) = \{r \mid q \in \text{Nucleo}(I) \wedge r \in \text{chius}_k(q)\}$$

La chiusura (p. 201 e 212) è calcolata con la lunghezza di prospezione  $k \geq 0$  desiderata.

Per completare la descrizione delle azioni di spostamento, che il pilota compie, passando dal macrostato  $I$  al macrostato  $I'$  alla "lettura" del simbolo (terminale o non)  $X$ , si introducono due relazioni binarie tra gli stati  $q$  di  $I$  e  $q'$  di  $I'$ , scritti nella forma  $(I, q)$  e  $(I', q')$ .

Una tradizionale mossa di spostamento  $I \xrightarrow{X} I'$  si suddivide in due casi:

*Mossa di proseguimento:* siano  $q \in \text{nucleo}(I)$  e  $q' \in I'$ , due stati per i quali esiste l'arco  $\delta(q, X) = q'$ ;  
allora nel pilota vale la relazione di proseguimento:

$$(I, q) \xrightarrow{X, pr} (I', q')$$

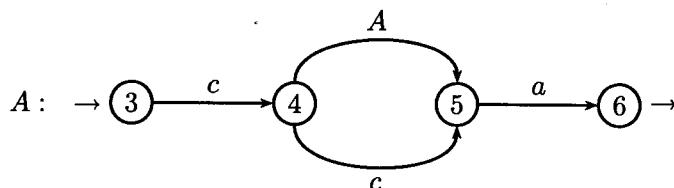
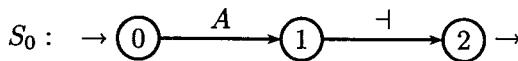
*Mossa di apertura:* siano  $q \in \text{resto}(I)$  e  $q' \in I'$ , due stati per i quali esiste l'arco  $\delta(q, X) = q'$ ;  
allora nel pilota vale la relazione di apertura:

$$(I, q) \xrightarrow{X, ap} (I', q')$$

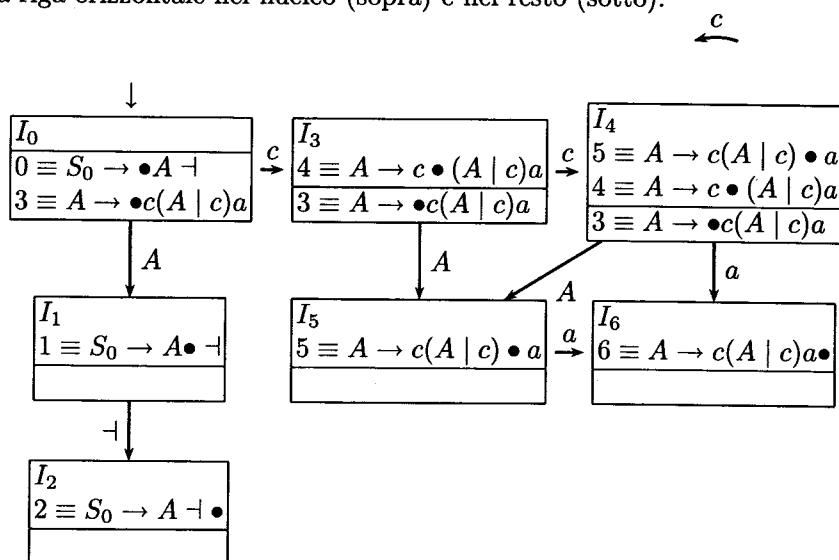
Si noti che è l'appartenenza dello stato d'origine al nucleo o al resto, che determina la classe della mossa, di proseguimento o di apertura.  
La costruzione del pilota con le relative mosse di apertura e proseguimento è illustrata dal prossimo esempio.

*Esempio 4.73.* Macchina pilota specializzata (da [35]).  
La grammatica comprende due regole:

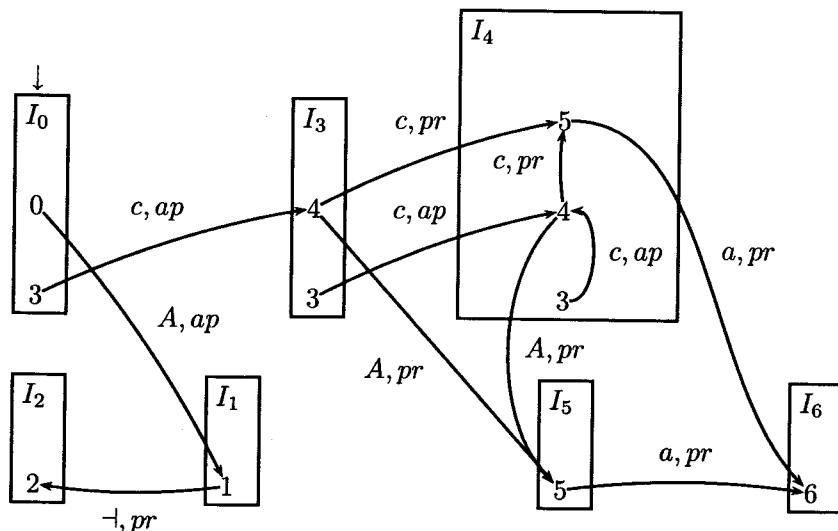
$$S_0 \rightarrow A \dashv \quad A \rightarrow c(A \mid c)a$$



Si costruisce al solito modo la macchina pilota; la prospettiva non serve poiché non vi sono conflitti nei macrostati. Nella figura, i macrostati sono divisi da una riga orizzontale nel nucleo (sopra) e nel resto (sotto).



La prossima figura mostra le relazioni di apertura e proseguimento tra gli stati dei macrostati:



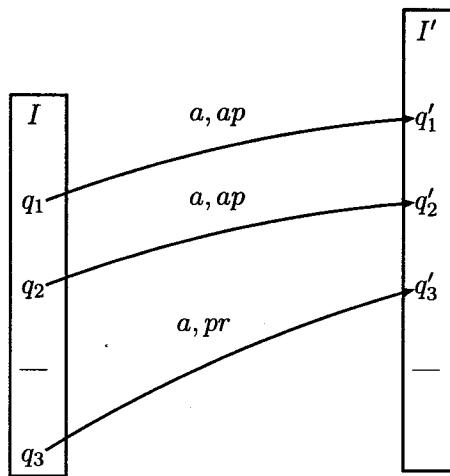
Ad es. l'arco da  $(I_3, 3)$  a  $(I_4, 4)$  è un'operazione di apertura, perché in  $I_3$  lo stato 3 sta nel resto. Similmente, poiché è  $3 \in resto(I_4)$ , l'arco  $(I_4, 3) \rightarrow (I_4, 4)$  è di apertura.

Invece gli archi  $(I_3, 4) \rightarrow (I_4, 5)$  e  $(I_4, 4) \rightarrow (I_4, 5)$  sono di proseguimento poiché gli stati d'origine sono nel nucleo dei rispettivi macrostati. Anche  $(I_3, 4) \rightarrow (I_5, 5)$  è un'operazione di proseguimento poiché in  $I_3$  lo stato 4 sta nel nucleo.

Si noti che tra i macrostati  $I_3$  e  $I_4$  vi sono relazioni di classi diverse, apertura e proseguimento.

Si osservi che nel pilota esiste la mossa  $I \xrightarrow{X} I'$  se, e solo se, esiste almeno una relazione  $(I, q) \xrightarrow{X, (pr|ap)} (I', q')$  tra due stati rispettivamente appartenenti a  $I$  e  $I'$ .

Quando esistono due o più relazioni, esse possono essere della stessa classe o di classi diverse, come sotto schematizzato:



Quando, come in questo caso, tra due macrostati vi sono relazioni sia di proseguimento che di apertura, si dice che vi è un *conflitto d'impilamento*.

La presenza di tale tipo di conflitto nella grammatica non è di impedimento alla costruzione del parsificatore deterministico. In caso di conflitto, il parsificatore applicherà l'operazione di apertura, perché è quella che inserisce nella pila l'etichetta della macchina che potrebbe essere stata attivata.

Si sottolinea che tra due macrostati vi possono essere due o più relazioni di apertura, con la conseguenza che occorre inserire nella pila un insieme di etichette.

### Algoritmo di parsificazione $ELR(1)$

Sia stata costruita la macchina pilota della grammatica, con le transizioni tra i macrostati qualificate come aperture o proseguimenti. Gli insiemi di prospettive sono calcolati come nel caso delle grammatiche non estese.

Si supponga inoltre che ogni macrostato della macchina pilota soddisfi la condizione  $LR(1)$  (p. 216). Ciò significa che in ogni macrostato, il carattere corrente determina la natura della mossa, spostamento o riduzione, e nel secondo caso, determina anche quale sia lo stato finale, corrispondente al riconoscimento della parte destra della regola da applicare nella riduzione.

Resta tuttavia il fatto che la macchina, che rappresenta la regola, può contenere cammini alternativi che conducono allo stato finale prescelto.

Affinché il parsificatore sia deterministico, deve valere la seguente condizione: in ogni stato di riduzione la stringa da disimpilare dalla pila può essere univocamente determinata dal parsificatore.

Il seguente algoritmo offre una possibile soluzione.

Il parsificatore  $ELR(1)$  è molto simile a quello  $LR(1)$  (p. 222) da cui si distacca nelle operazioni di apertura e di riduzione. Esso è descritto da un automa a pila in cui la pila ha la forma

$$I_0 E_0 D_0 I_1 E_1 D_1 I_2 \dots I_{n-1} E_{n-1} D_{n-1} I_n$$

dove gli  $I$  sono macrostati, gli  $E$  sono insiemi di etichette delle regole, e i  $D$  sono simboli (terminali o non) della grammatica.

L'etichetta d'una regola avente il nonterminale  $B$  come parte sinistra e  $q_{B,0}$  come stato iniziale della macchina  $M_B$ , sarà scritta nella forma  $(M_B, q_{B,0})$ . La pila differisce da quella del caso  $LR(1)$  soltanto per la presenza degli insiemi  $E$ , le etichette inserite dalle mosse di apertura. Gli insiemi  $E$  mancano invece nelle mosse di proseguimento. Conviene premettere la descrizione intuitiva dell'algoritmo.

Quando l'automa esegue la mossa d'apertura d'una regola, esso impila l'etichetta della regola, il terminale letto e il macrostato indicato dal pilota come arrivo della mossa.

Quando esegue una mossa di proseguimento, l'automa impila soltanto il simbolo letto e il macrostato d'arrivo.

Quando il macrostato corrente e il carattere di prospezione selezionano una riduzione, ossia uno stato finale  $q_f$  della macchina  $M_B$ , l'automa esegue una serie di passi, per togliere dalla pila una stringa di simboli grammaticalici,  $D_k, D_{k+1}, \dots, D_{n-1}$ , tale che:

- essa appartiene al linguaggio regolare  $R(M_B)$ , di alfabeto terminale e nonterminale, riconosciuto dalla macchina  $M_B$ ;
- essa è riconosciuta nello stato finale  $q_f$  (infatti la macchina  $M_B$  potrebbe avere altri stati finali);
- nella pila tale stringa è compresa tra un punto dove sta l'etichetta  $(M_B, q_{0,B})$  e la cima.

Ciò fatto, la mossa di riduzione esegue lo spostamento del nonterminale  $B$ , partendo dal macrostato affiorato sulla pila.

#### *Algoritmo 4.74. Analizzatore sintattico $ELR(1)$ .*

Sia  $a \in \Sigma$  il carattere corrente.

L'automa inizia nel macrostato  $I_0$ .

**Mossa di proseguimento:** si applica se nel macrostato corrente  $I$  è definita la relazione

$$(I, q) \xrightarrow{a,pr} (I', q')$$

l'automa sposta  $a$  dalla stringa sorgente alla pila e impila il macrostato  $I'$ ;

**Mossa di apertura:** si applica se nel macrostato corrente  $I$  sono definite una o più relazioni di apertura:

$$(I, q_1) \xrightarrow{a, ap} (I', r_1)$$

$$\dots \xrightarrow{a, ap} \dots$$

$$(I, q_m) \xrightarrow{a, ap} (I', r_m)$$

Si indica con  $E$  l'insieme delle etichette delle regole grammaticali corrispondenti.

L'automa impila nell'ordine:

1. l'insieme  $E$  delle etichette;
2. il simbolo  $a$ , togliendolo dal suffisso d'ingresso;
3. il macrostato  $I'$ .

Se nella configurazione corrente è definita sia una mossa di proseguimento, sia una mossa di apertura, l'automa sceglie la seconda.

**Mossa di riduzione:** Si applica se nel macrostato corrente  $I_n$  esiste una coppia  $(q_f, \pi)$ , dove  $q_f$  è uno stato finale della macchina  $M_B$ , e il carattere corrente  $a$  sta nell'insieme di prospezione  $\pi$ .

La pila corrente sia

$$I_0 E_0 D_0 I_1 \dots I_{n-k} \overbrace{E_{n-k} D_{n-k} I_{n-k+1} \dots I_{n-1} E_{n-1} D_{n-1}}^{\beta'} I_n$$

Si disimpilano uno o più elementi (indicati come  $\beta'$ ), ottenendo la pila

$$I_0 E_0 D_0 I_1 \dots I_{n-k}$$

tale da soddisfare entrambe le seguenti condizioni:

1.  $E_{n-k}$  è l'insieme di etichette, più vicino alla cima, il quale contiene l'etichetta  $(M_B, q_f)$  della riduzione selezionata;
2. cancellando i macrostati e le etichette dalla stringa disimpilata  $\beta'$ , la stringa ottenuta  $D_{n-k+1} \dots D_{n-1} \in (V \cup \Sigma)^*$  appartiene al linguaggio regolare accettato dalla macchina  $M_B$  con stato finale  $q_f$ .

Poi l'automa esegue la mossa spontanea che va dal macrostato  $I_{n-k}$  al macrostato  $I'$ , leggendo il nonterminale  $B$ . Anche tale mossa è trattata come una mossa di apertura o di proseguimento, a seconda della relazione esistente tra i macrostati  $I_{n-k}$  e  $I'$  del pilota.

La mossa di riduzione è la più nuova e complessa. Essa può agire nel seguente modo: usando la versione specularmente riflessa  $(M_B)^R$  della macchina  $M_B$ , analizza dalla cima in giù i simboli grammaticali presenti nella pila, fintanto che le due condizioni sono soddisfatte:

la macchina  $(M_B)^R$  ha raggiunto lo stato  $q_{B,0}$  (quello iniziale della macchina  $M_B$ ); l'ultimo elemento tolto dalla pila contiene l'etichetta  $(M_B, q_f)$  della riduzione selezionata.

Esaminando il determinismo del parsificatore, si nota che, se la grammatica soddisfa la condizione  $LR(1)$ , la scelta tra riduzione e spostamento è deterministica. In caso di spostamento, la preferenza accordata all'operazione di

apertura, risolve il conflitto eventuale tra apertura e proseguimento. Se la mossa è di riduzione, l'insieme  $E_{n-k}$  delle etichette potrebbe contenere più regole tra cui scegliere; ma la condizione 2. della mossa di riduzione è soddisfatta da una e una sola regola, a ragione del modo in cui è stata costruita la pila.

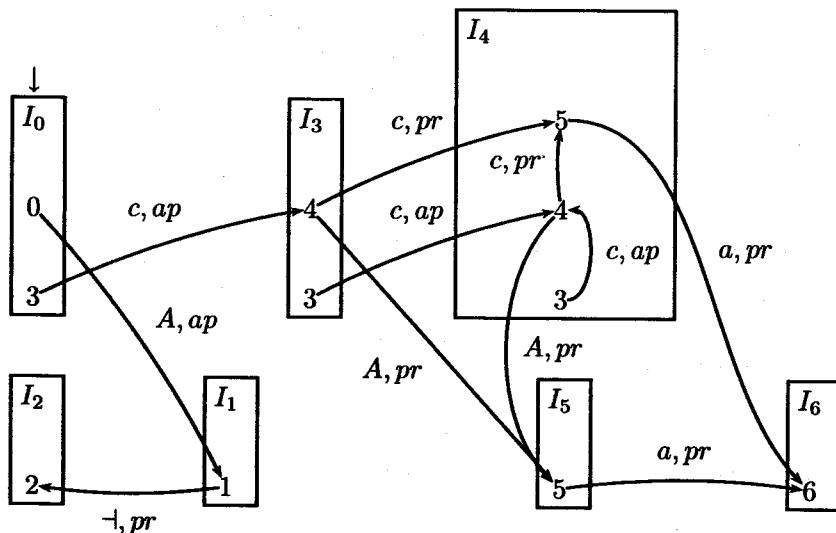
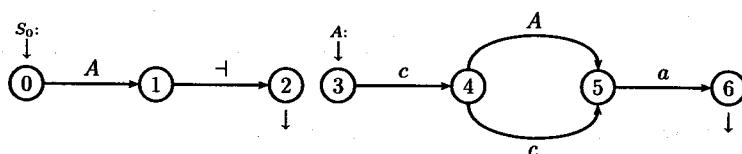
Di conseguenza il funzionamento è deterministico.

Diversi miglioramenti sono stati proposti per accelerare le mosse di riduzione: uso di contatori o di puntatori messi nella pila in fase di apertura, costruzione d'una macchina finita per operare il riconoscimento della stringa dal fondo all'inizio, trasformazione della grammatica per eliminare le situazioni inefficienti.

Segue la traccia dell'analisi sintattica d'una stringa.

*Esempio 4.75.* Traccia dell'analisi (es. 4.73 continuato).

Si riproduce per comodità la rete e il pilota:

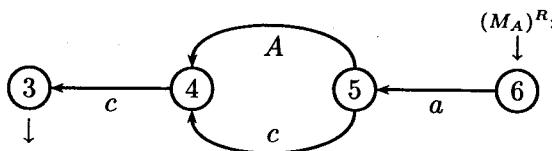


Traccia:

| Pila  | x  | Commento           |
|-------|--|--------------------|
| $I_0$ | $c \quad c \quad c \quad a \quad a \dashv$   | apertura           |
| $I_0$ | $(M_A, 3) \underline{c} I_3 \quad c \quad c \quad a \quad a \dashv$                                  | apertura           |
| $I_0$ | $(M_A, 3) \underline{c} I_3 \quad (M_A, 3) \underline{c} I_4 \quad c \quad a \quad a \dashv$         | apertura           |
| $I_0$ | $(M_A, 3) \underline{c} I_3 \quad (M_A, 3) \underline{c} I_4 \quad c I_4 \quad a \quad a \dashv$     | proseguimento      |
| $I_0$ | $(M_A, 3) \underline{c} I_3 \quad (M_A, 3) \underline{c} I_4 \quad c I_4 \quad a I_6 \quad a \dashv$ | riduzione: stato 6 |

Nel macrostato corrente  $I_6$ , il pilota prescrive una riduzione associata allo stato 6 della macchina  $M_A$ , corrispondente alla regola  $A \rightarrow c(A \mid c)a$ .

Per individuare più agevolmente la stringa da disimpilare, conviene visualizzare la macchina riflessa:



Leggendo la stringa  $ac$  compresa tra la cima della pila e la prima etichetta  $(M_A, 6)$ , relativa alla macchina prescritta, il parsificatore svolge un calcolo con la macchina riflessa. Poiché questa raggiunge lo stato 4 non finale di  $(M_A)^R$ , la stringa  $ac$  non è accettabile, e occorre scavare ancora nella pila.

Il punto corretto di disimpilamento è l'etichetta sottolineata, perché la stringa  $acc$  è accettata dalla macchina riflessa. Effettuata la riduzione  $cca \Rightarrow A$ , ripartendo dalla nuova configurazione, l'algoritmo esegue due successive mosse di proseguimento:

|   |            |                    |
|---|------------|--------------------|
| $I_0 (M_A, 3) \underline{c} I_3$                        | $a \dashv$ | proseguimento      |
| $I_0 (M_A, 3) \underline{c} I_3 \underline{AI_5}$       | $a \dashv$ | proseguimento      |
| $I_0 (M_A, 3) \underline{c} I_3 \underline{AI_5} a I_6$ | $\dashv$   | riduzione: stato 6 |

La riduzione  $cAa \Rightarrow A$ , consistente con l'insieme sottolineato e con l'accettazione di  $aAc$  da parte della macchina riflessa, svuota la pila, e la stringa sorgente è accettata.

## 4.6 Un algoritmo generale di analisi sintattica

Per completare lo studio della parsificazione, si espone il metodo di *Earley*, che permette di trattare qualsiasi grammatica libera e costruisce tutte le derivazioni delle frasi ambigue. La sua complessità di calcolo è proporzionale al

cubo della lunghezza della stringa, ma si riduce al quadrato se la grammatica non è ambigua, e ancora più se essa è deterministica.

Questo sviluppo conclude il percorso iniziato con i metodi deterministicici  $LL(k)$  e proseguito con quelli  $LR(k)$ ; lungo il percorso gli algoritmi si sono via via arricchiti della capacità di risolvere situazioni non deterministiche. Gli algoritmi  $LR(k)$  portano avanti più calcoli in parallelo, ma soltanto fino al momento in cui avviene una riduzione; tale limite è insito nel modello dell'automa a pila deterministico, che non permette di gestire contemporaneamente più pile.

Il prossimo algoritmo prende le mosse da quello  $LR(k)$  ma, abbandonando il modello deterministico a pila, si avvale d'una struttura dati più ricca (vettore di insiemi), che rappresenta efficientemente tante pile aventi parti condivise. Così esso permette la simulazione di un automa a pila indeterministico, senza cadere nella complessità di calcolo esponenziale (p. 150) di tale modello.

Per semplicità, le grammatiche considerate in questa parte non sono estese, ma l'algoritmo di Earley vale senza difficoltà anche per quelle estese. Inoltre, per gradualità espositiva, si impone inizialmente che la grammatica sia priva di regole vuote.

Al solito al posto della grammatica si potrà usare la rete ricorsiva di macchine.

Sia  $x$  la stringa sorgente di lunghezza  $n \geq 1$ ; si denotano con  $x_i$  l' $i$ -esimo carattere e con  $x_{i..j}$  la sottostringa dei caratteri da  $i$  a  $j$ , estremi compresi; l'intera stringa sorgente è dunque  $x \equiv x_{1..n}$ .

Questo algoritmo non usa una macchina pilota (la quale in generale violerebbe la condizione  $LR(k)$ ), bensì registra in un vettore  $E[0..n]$ , dimensionato sulla lunghezza della stringa sorgente, ogni stato in cui la rete ricorsiva potrebbe trovarsi, dopo la lettura dell' $i$ -esimo carattere. Lo stato è affiancato da un intero (o puntatore all'indietro), che dice in quale posizione della stringa l'algoritmo ha iniziato a cercare l'istanza corrente della macchina (ossia della regola grammaticale).

Più precisamente ogni elemento o casella  $E[i]$ ,  $0 \leq i \leq n$ , del vettore contiene un insieme di coppie

$$\langle \text{stato, puntatore} \rangle = (s, p)$$

dove  $s$  è uno stato della rete e  $p$  sta nell'intervallo  $(0 \dots i)$ .

Per immediatezza di lettura, accanto allo stato  $s$ , si potrà scrivere la regola sintattica marcata corrispondente.

L'algoritmo può operare anche con la prospezione, ma la sua complessità asintotica di calcolo non migliora, e conviene per semplicità presentare la versione che non fa uso della prospezione.

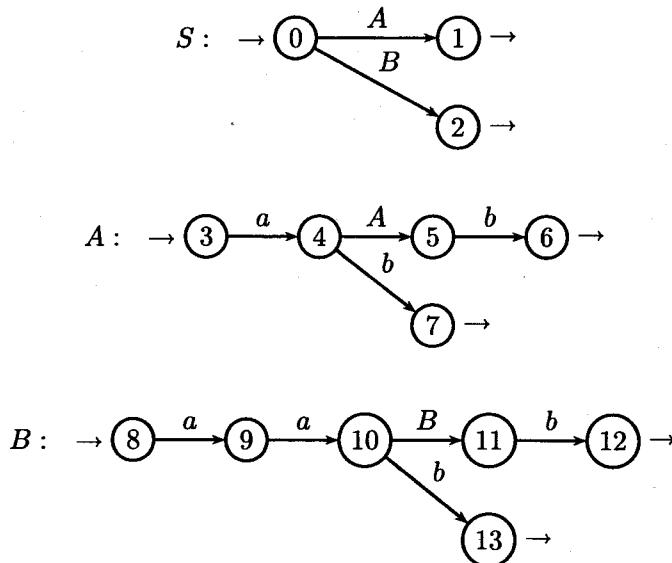
#### *Esempio 4.76. Introduzione al metodo di Earley.*

Il linguaggio

$$\{a^n b^n \mid n \geq 1\} \cup \{a^{2n} b^n \mid n \geq 1\}$$

non è deterministico, e quindi la condizione  $LR(k)$  è violata dalla seguente grammatica:

$$S \rightarrow A \mid B \quad A \rightarrow aAb \mid ab \quad B \rightarrow aaBb \mid aab$$



L'analisi della stringa  $aabb$  costruirà un vettore  $E[0] \dots E[4]$  avente la casella iniziale  $E[0]$  e una casella per ogni posizione della stringa. Ogni casella contiene un insieme di coppie.

$E[0]$  contiene inizialmente una sola coppia:

$$\langle \text{stato} = 0, \text{puntatore} = 0 \rangle$$

scritta come  $(0, p = 0)$ , dove il primo zero è lo stato iniziale della rete, e il puntatore punta alla posizione zero, che precede il primo carattere della stringa.

$$E[0] = \{(0 \equiv S \rightarrow \bullet(A \mid B), p = 0)\}$$

In generale l'algoritmo opera sull'insieme  $E[i]$  nel modo seguente: esamina in ordine le coppie ivi presenti, eseguendo una di tre operazioni, scansione, predizione, completamento, su ciascuna coppia, a seconda della forma di essa. La predizione è soltanto un nuovo nome per l'operazione di chiusura  $LR(0)$  (p. 201); si applica a una coppia il cui stato abbia un arco uscente con etichetta nonterminale (ossia nella regola marcata vi è un nonterminale a destra del

pallino). Essa aggiunge all'insieme  $E[i]$  una nuova coppia con lo stato iniziale della macchina (ossia con il pallino all'inizio della parte destra della regola marcata).

Alla seconda componente della coppia, il puntatore, si assegna il valore  $i$ , poiché la coppia è stata creata al passo  $i$ , partendo da una coppia presente in  $E[i]$ .

Il senso dell'operazione è che la predizione aggiunge a  $E[i]$  tutti gli stati iniziali delle macchine che potrebbero riconoscere una sottostringa che inizia da  $x_{i+1}$ . Nell'esempio la predizione aggiunge a  $E[0]$  le coppie:

$$(3 \equiv A \rightarrow \bullet aAb \mid \bullet ab, p = 0), \quad (8 \equiv B \rightarrow \bullet aaBb \mid \bullet aab, p = 0)$$

Poiché la predizione non è applicabile alle nuove coppie, si passa ora alla seconda operazione, la *scansione*, che è applicabile quando da uno stato esce un arco con etichetta terminale. Se tale terminale egualia  $x_{i+1}$ , lo stato di arrivo è aggiunto all'insieme  $E[i + 1]$  con lo stesso puntatore dello stato di partenza. Ciò sta a indicare che il terminale è stato letto.

Nell'esempio, con  $i = 0$  e  $x_1 = a$ , la scansione inserisce in  $E[1]$  le coppie:

$$(4 \equiv A \rightarrow a \bullet Ab \mid a \bullet b, p = 0), \quad (9 \equiv B \rightarrow a \bullet aBb \mid a \bullet ab, p = 0)$$

Applicando la predizione alle nuove coppie, si aggiunge a  $E[1]$  la coppia

$$(3 \equiv A \rightarrow \bullet aAb \mid \bullet ab, p = 1)$$

dove è da notare il valore 1 del puntatore.

Se al termine dell'elaborazione di  $E[i]$  l'insieme  $E[i + 1]$  restasse vuoto, si sarebbe scoperto un errore nella stringa sorgente.

Ora si applica a  $E[1]$  la scansione di  $x_2 = a$ , che produce in  $E[2]$  le coppie

$$(10 \equiv B \rightarrow aa \bullet Bb \mid aa \bullet b, p = 0), \quad (4 \equiv A \rightarrow a \bullet Ab \mid a \bullet b, p = 1)$$

alle quali la predizione aggiunge le coppie

$$(8 \equiv B \rightarrow \bullet aaBb \mid \bullet aab, p = 2), \quad (3 \equiv A \rightarrow \bullet aAb \mid \bullet ab, p = 2)$$

La scansione di  $x_3 = b$  produce in  $E[3]$  le coppie

$$(13 \equiv B \rightarrow aab \bullet, p = 0), \quad (7 \equiv A \rightarrow ab \bullet, p = 1)$$

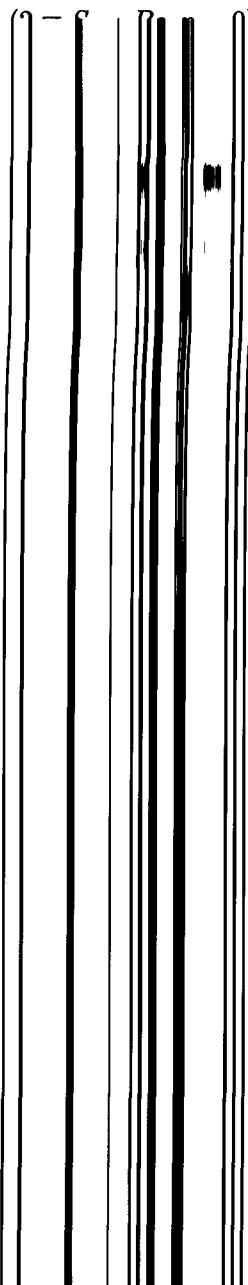
Ora entra in gioco la terza operazione, il *completamento* (che ha un effetto analogo alla riduzione di  $LR(k)$ ). Esso si applica a una coppia  $(s, j) \in E[i]$  il cui stato  $s$  è finale per una macchina  $M_A$ , ossia corrisponde a una regola  $s \equiv A \rightarrow \dots \bullet$ , con marca in fondo.

L'operazione ritorna sull'insieme  $E[j]$  puntato da  $j$  (è necessariamente  $j < i$  non essendovi regole vuote nella grammatica). Ricerca in  $E[j]$  una coppia  $(q, k)$ , tale che da  $q$  esca l'arco  $q \xrightarrow{A} r$ . Infine aggiunge all'insieme corrente

$E[i]$  lo stato  $r$  d'arrivo, con il valore  $k$  del puntatore. Ossia aggiunge la regola marcata ottenuta, da quella trovata in  $E[j]$ , spostando il pallino a destra di  $A$ .

Il completamento si applica a entrambe le coppie di  $E[3]$ , poiché 13 e 7 sono stati finali, rispettivamente di  $B$  e di  $A$ . Il puntatore di 13 rimanda a  $E[0]$ , che intuitivamente è l'insieme in cui l'analisi si trovava quando fu iniziata la ricerca di questa istanza di  $B$ .

In  $E[0]$  si trova la coppia  $(0 \equiv S \rightarrow \bullet A \mid \bullet B, p = 0)$ , dal cui stato origina l'arco  $0 \xrightarrow{B} 2$ ; si inserisce in  $E[3]$  lo stato di arrivo 2, con lo stesso puntatore:



*Descrizione precisa dell'algoritmo di Earley*

L'algoritmo è in effetti un parsificatore che sviluppa simultaneamente tutte le possibili derivazioni sinistre della stringa. L'algoritmo legge la stringa  $x_{1..n}$  da sinistra a destra e, quando esamina il carattere  $x_i$ , produce certe strutture a due campi o coppie, della forma  $\langle$ stato della rete, puntatore $\rangle$ , scritta come  $(s, p = \dots)$ , dove il puntatore  $p$  è compreso tra 0 e  $i$ . Lo stato può essere scritto anche come *regola grammaticale marcata*  $A \rightarrow \alpha \bullet \beta$ .

Intuitivamente, la coppia  $(s \equiv A \rightarrow \alpha \bullet \beta, j)$  rappresenta un'asserzione e un obiettivo:

**Asserzione:** è stata trovata una sottostringa  $x_{j+1..i}$  ( $0 \leq j < i$ ) che deriva da  $\alpha$ , in formula

$$\alpha \xrightarrow{*} x_{j+1..i}$$

**Obiettivo:** trovare tutte le posizioni  $k$  ( $i < k \leq n$ ) tali che la sottostringa  $x_{i+1..k}$  derivi da  $\beta$ , in formula

$$\beta \xrightarrow{*} x_{i+1..k}$$

Se l'algoritmo troverà tale posizione  $k$ , potrà asserire che dal nonterminale  $A$  deriva la sottostringa  $x_{j+1..k}$ , ossia

$$A \xrightarrow{*} x_{j+1..k}$$

Una coppia  $(q \equiv A \rightarrow \alpha \bullet, j)$ , il cui stato  $q$  è finale (ossia ha la marca in fondo alla regola), è detta *completata*.

*Algoritmo 4.77. Riconoscitore di Earley.*

L'algoritmo costruisce un vettore  $E[0..n]$  dimensionato sulla lunghezza della stringa sorgente, le cui caselle contengono degli insiemi di coppie.  $E[0]$  è l'insieme iniziale e  $E[i]$  quello associato alla posizione  $x_i$  della stringa.

Pur se il compito del riconoscitore è trovare la derivazione dell'intera stringa  $x$ , l'algoritmo produce più di quanto richiesto, in quanto dice anche se ogni prefisso della stringa appartiene al linguaggio.

L'insieme iniziale è riempito con certe coppie ricavate dall'assioma; tutti gli altri insiemi sono inizialmente vuoti.

**Passo 0: Inizializzazione.** (Predisponde gli obiettivi per trovare ogni prefisso di  $x$  derivabile dall'assioma  $S$ . Al puntatore è assegnato il valore zero.)

$$E[0] := (q_{ini}, 0), \text{ dove } q_{ini} \text{ è lo stato iniziale della rete.}$$

$$E[i] := \emptyset, \text{ per } i = 1, \dots, n$$

$$i := 0$$

Poi si applicano nell'ordine naturale  $0, 1, \dots, n$  le operazioni di predizione, completamento e scansione descritte nel seguito, per calcolare tutti gli insiemi

$E[i]$ . L'algoritmo al passo  $i$  può aggiungere coppie soltanto all'insieme corrente  $E[i]$  e a quello successivo  $E[i+1]$ . Se nessuna delle operazioni ha aggiunto nuove coppie a  $E[i]$ , si passa al calcolo di  $E[i+1]$ . Se  $E[i+1]$  è vuoto e  $i < n$  la stringa è rifiutata.

**Predizione.** (Ogni obiettivo presente nell'insieme  $E[i]$  può aggiungere altri sotto obiettivi all'insieme stesso; al puntatore è assegnato l'indice corrente.)

per ogni coppia in  $E[i]$  della forma ( $q \equiv A \rightarrow \alpha \bullet B\gamma, j$ ),

dove da  $q$  esce l'arco  $q \xrightarrow{B} s$ ,

aggiungi all'insieme  $E[i]$  la coppia ( $r, i$ ),

dove  $r$  è lo stato iniziale della macchina  $M_B$

**Completamento.** (Una coppia completata ( $q \equiv A \rightarrow \alpha \bullet, j$ ) asserisce che è stata trovata la derivazione della stringa  $x_{j+1..i}$  dal nonterminale  $A$ . Occorre aggiornare l'insieme  $E[i]$  con tale asserzione.)

per ogni coppia completata ( $q \equiv A \rightarrow \alpha \bullet, j$ ) in  $E[i]$ ,

per ogni coppia in  $E[j]$  della forma ( $r \equiv B \rightarrow \beta \bullet A\gamma, k$ ),

tal che da  $r$  esce l'arco  $r \xrightarrow{A} s$ ,

aggiungi a  $E[i]$  la coppia ( $s \equiv B \rightarrow \beta A \bullet \gamma, k$ )

**Scansione.** (Aggiorna gli obiettivi dell'insieme  $E[i+1]$  in accordo con il carattere corrente. Il puntatore è posto eguale a quello della coppia esaminata.)

per ogni coppia in  $E[i]$  della forma ( $q \equiv A \rightarrow \alpha \bullet a\gamma, j$ ), se  $a = x_{i+1}$ :

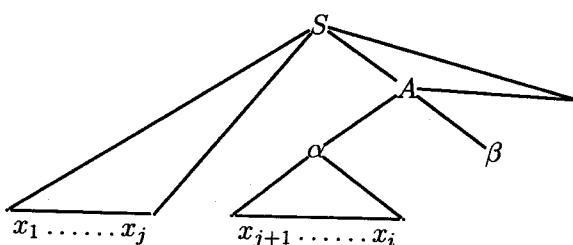
aggiungi all'insieme  $E[i+1]$  la coppia ( $r \equiv A \rightarrow \alpha a \bullet \gamma, j$ )

dove  $r$  è l'arrivo dell'arco  $q \xrightarrow{a} r$

L'algoritmo termina quando la costruzione dell'insieme  $E[n]$  è stata finita; termina prematuramente con insuccesso, se la scansione non ha trovato una coppia da aggiungere a  $E[i+1], i < n$ .

Se l'insieme finale  $E[n]$  contiene (almeno) una coppia completata ( $q_{term} \equiv S \rightarrow \alpha \bullet, 0$ ), dove  $q_{term}$  è lo stato finale della macchina dell'assioma, la stringa sorgente è accettata.

È significativo ripensare le tre operazioni come costruzione progressiva di alberi sintattici. Allora a ogni insieme corrisponde un insieme di alberi. In particolare in  $E[i]$  vi è la coppia ( $A \rightarrow \alpha \bullet \beta, p = j$ ) se, e solo se, per la grammatica data, esiste un albero sintattico della forma:



Al termine dell'algoritmo, la stringa è accettata se, e solo se, tra gli alberi associati all'ultimo insieme vi è un albero sintattico completo con radice nell'assioma.

Note:

Ciascun carattere della stringa è esaminato dall'algoritmo una sola volta nella scansione.

La predizione esamina ripetutamente le coppie dell'insieme corrente, il quale può crescere. Rappresentando l'insieme in una coda FIFO, è facile evitare di esaminare più volte la stessa coppia.

Nel completamento: se nella coppia  $(r \equiv B \rightarrow \beta \bullet A\gamma, k) \in E[j]$  la stringa  $\gamma$  è vuota, la nuova coppia  $(s \equiv B \rightarrow \beta A \bullet \gamma, k)$  aggiunta a  $E[i]$  è completata, quindi occorre iterare di nuovo.

Sarebbe facile dimostrare che l'algoritmo accetta una stringa soltanto se appartenente al linguaggio  $L(G)$ : infatti una coppia viene aggiunta all'insieme, soltanto se la derivazione da essa asserita è possibile; d'altra parte, la dimostrazione che ogni frase del linguaggio è riconosciuta dall'algoritmo presenta qualche difficoltà ed è omessa.<sup>26</sup>

#### *Costruzione dell'albero sintattico*

Si è finora considerato il riconoscimento d'una stringa, ma l'interpretazione delle coppie come alberi permette di costruire l'albero sintattico della frase.

*Esempio 4.78.* Costruzione dell'albero per l'es. 4.76 (p. 240).

Per comodità di lettura, sono riprodotti gli insiemi di coppie per la stringa  $aabb$ . A destra si vedono tre tappe nella costruzione dell'albero:

---

<sup>26</sup> Si veda l'articolo originale di Earley [17] o un testo quale [19, 41, 23].

|     |        |  |   |
|-----|--------|--|---|
|     | $E[0]$ | $(S \rightarrow \bullet(A \mid B), p = 0)$<br>$(A \rightarrow \bullet aAb \mid \bullet ab, p = 0)$<br>$(B \rightarrow \bullet aaBb \mid \bullet aab, p = 0)$   |   |
| $a$ | $E[1]$ | $(A \rightarrow a \bullet Ab \mid a \bullet b, p = 0)$<br>$(B \rightarrow a \bullet aBb \mid a \bullet ab, p = 0)$<br>$(A \rightarrow \bullet aAb \mid \bullet ab, p = 1)$   |  |
| $a$ | $E[2]$ | $(B \rightarrow aa \bullet Bb \mid aa \bullet b, p = 0)$<br>$(A \rightarrow a \bullet Ab \mid a \bullet b, p = 1)$<br>$(B \rightarrow \bullet aaBb \mid \bullet aab, p = 2)$<br>$(A \rightarrow \bullet aAb \mid \bullet ab, p = 2)$ |  |
| $b$ | $E[3]$ | $(B \rightarrow aab\bullet, p = 0)$<br>$(A \rightarrow ab\bullet, p = 1)$<br>$(S \rightarrow B\bullet, p = 0)$<br>$(A \rightarrow aA \bullet b, p = 0)$  |  |
| $b$ | $E[4]$ | $(A \rightarrow aAb\bullet, p = 0)$<br>$(S \rightarrow A\bullet, p = 0)$   |   |

Si inizia dall'ultima casella. Poiché essa contiene la coppia completata ( $S \rightarrow A\bullet, p = 0$ ), deve esistere la derivazione  $S \Rightarrow A \xrightarrow{*} x_{1..4}$ , che è rappresentata dall'albero superiore.

Per l'ultimo indice 4, si esamina l'insieme  $E[4]$ , alla ricerca di una coppia completata del nonterminale  $A$ ; si trova ( $A \rightarrow aAb\bullet, p = 0$ ), che fa disegnare l'albero di mezzo.

Il sottoalbero centrale  $A_{2..3}$  da espandere copre la stringa tra le posizioni 2 e 3; la seconda di esse porta a esaminare l'insieme  $E[3]$ . Al fine di sviluppare l'albero innestato sotto  $A_{2..3}$ , si cerca dunque in  $E[3]$  una coppia completata di  $A$ , che punti alla casella precedente 2, ossia con puntatore eguale a  $(2 - 1) = 1$ ; si trova ( $A \rightarrow ab\bullet, p = 1$ ). L'albero così cresce e si completa nella figura inferiore.

### Analisi d'una frase ambigua

Se la grammatica è ambigua, l'analisi d'una frase produce tutti gli alberi possibili, rappresentati in modo fattorizzato, come si vedrà nel prossimo esempio.

*Esempio 4.79.* Parsificazione d'un linguaggio ambiguo

La grammatica, ricorsiva bilateralmente quindi ambigua, è

$$S \rightarrow E \quad E \rightarrow E + E \quad E \rightarrow a$$

Gli insiemi per la stringa sorgente  $a + a + a$  sono sotto mostrati; una linea orizzontale separa le coppie create da successive iterazioni del completamento.

|        |   |
|--------|---|
| $E[0]$ | $\boxed{S \rightarrow \bullet E, 0}$                    |
|        | $\boxed{E \rightarrow \bullet E + E \mid \bullet a, 0}$ |

|            |  |
|------------|--|
| $a \ E[1]$ | $\boxed{E \rightarrow a\bullet, 0}$      |
|            | $\boxed{S \rightarrow E\bullet, 0}$      |
|            | $\boxed{E \rightarrow E \bullet + E, 0}$ |

|          |   |
|----------|---|
| $+ E[2]$ | $\boxed{E \rightarrow E + \bullet E, 0}$                |
|          | $\boxed{E \rightarrow \bullet E + E \mid \bullet a, 2}$ |

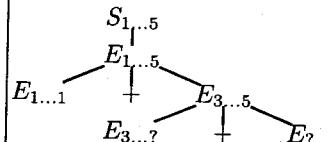
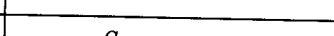
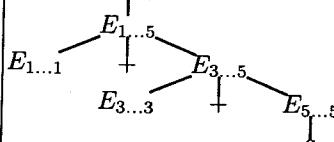
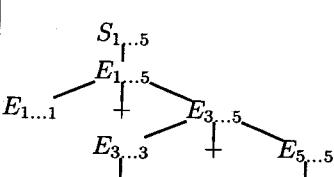
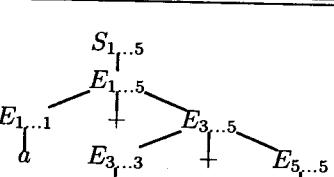
|            |  |
|------------|--|
| $a \ E[3]$ | $\boxed{E \rightarrow a\bullet, 2}$      |
|            | $\boxed{E \rightarrow E + E\bullet, 0}$  |
|            | $\boxed{E \rightarrow E \bullet + E, 2}$ |
|            | $\boxed{S \rightarrow E\bullet, 0}$      |
|            | $\boxed{E \rightarrow E \bullet + E, 0}$ |

|          |   |
|----------|---|
| $+ E[4]$ | $\boxed{E \rightarrow E + \bullet E, 0}$                |
|          | $\boxed{E \rightarrow E + \bullet E, 2}$                |
|          | $\boxed{E \rightarrow \bullet E + E \mid \bullet a, 4}$ |

|            |  |
|------------|--|
| $a \ E[5]$ | $\boxed{E \rightarrow a\bullet, 4}$      |
|            | $\boxed{E \rightarrow E + E\bullet, 2}$  |
|            | $\boxed{E \rightarrow E \bullet + E, 4}$ |
|            | $\boxed{E \rightarrow E + E\bullet, 0}$  |
|            | $\boxed{E \rightarrow E \bullet + E, 2}$ |
|            | $\boxed{S \rightarrow E\bullet, 0}$      |
|            | $\boxed{E \rightarrow E \bullet + E, 0}$ |

Si noti la ripetizione d'una stessa coppia in insiemi diversi. Nell'ultimo insieme,  $E[5]$ , la coppia completa ( $S \rightarrow E\bullet, 0$ ) manifesta la validità della stringa. Si mostra la costruzione dei due alberi della frase considerata.

| Coppie  | Primo albero   |
|---|--|
| $\overline{E[5]}$<br>$S \rightarrow E \bullet, 0$<br>$E \rightarrow E + E \bullet, 0$ | $  \begin{array}{c}  S_{1 \dots 5} \\    \\  E_{1 \dots 5} \\    \\  E_{1 \dots ?} + E_{? \dots 5}  \end{array}  $             |
| $E \rightarrow a \bullet, 4$  | $  \begin{array}{c}  S_{1 \dots 5} \\    \\  E_{1 \dots 5} \\    \\  E_{1 \dots 3} + E_{5 \dots 5} \\    \\  a  \end{array}  $ |
| $\overline{E[3]}$<br>$E \rightarrow E + E \bullet, 0$<br>$E \rightarrow a \bullet, 2$ | $  \begin{array}{c}  S_{1 \dots 5} \\    \\  E_{1 \dots 5} \\    \\  E_{1 \dots 1} + E_{3 \dots 3} \\    \\  a  \end{array}  $ |
| $\overline{E[1]}$<br>$E \rightarrow a \bullet, 0$                                     | $  \begin{array}{c}  S_{1 \dots 5} \\    \\  E_{1 \dots 5} \\    \\  E_{1 \dots 3} + E_{3 \dots 3} \\    \\  a  \end{array}  $ |

| Coppie  | Secondo albero   |
|---|--|
| $E[5]$  |  |
| $\frac{S \rightarrow E \bullet, 0}{E \rightarrow E + E \bullet, 0}$ |   |
| $E \rightarrow E + E \bullet, 2$                                    |   |
| $E \rightarrow a \bullet, 4$  |   |
| $\frac{E[3]}{E \rightarrow a \bullet, 2}$                           |   |
| $\frac{E[1]}{E \rightarrow a \bullet, 0}$                           |  |

### Complessità di calcolo del riconoscitore

Non è difficile calcolare la complessità asintotica di calcolo dell'algoritmo di Earley, nel caso peggiore.

Per una stringa di lunghezza  $n$ , si conteggiano le coppie e le operazioni eseguite su di esse.

1. Ciascun insieme  $E[i]$  può contenere un numero di coppie che cresce linearmente con  $i$ , dunque è maggiorato da  $n$ .
2. Le operazioni di scansione e predizione eseguono, su ogni coppia dell'insieme  $E[i]$ , un numero di passi indipendente da  $n$ .
3. L'operazione di completamento esegue  $\mathcal{O}(i)$  passi per ogni coppia trattata, perché potrebbe dover aggiungere a  $E[i]$  un numero di coppie dell'ordine di  $\mathcal{O}(j)$ , dove  $E[j], 0 \leq j < i$  è un insieme precedente. In complesso il completamento richiede  $\mathcal{O}(n^2)$  passi.

4. Sommando i passi eseguiti per ogni  $i$  da 0 a  $n$ , si ottiene il limite  $\mathcal{O}(n^3)$ .

*Proprietà 4.80.* La complessità asintotica dell'algoritmo di Earley nel caso peggiore è  $\mathcal{O}(n^3)$ , dove  $n$  è la lunghezza della stringa sorgente.

In pratica l'algoritmo è più veloce: per molte grammatiche è  $\mathcal{O}(n)$  e per ogni grammatica inambigua è  $\mathcal{O}(n^2)$ .

L'aggiunta delle operazioni sui puntatori, necessarie per costruire gli alberi sintattici, non modifica la classe di complessità asintotica di calcolo del riconoscitore.

### Trattamento delle regole vuote

Nella presentazione dell'algoritmo di Earley sono state finora evitate le grammatiche con regole vuote, ma anch'esse possono essere ammesse, al costo di alcune modifiche.

Sia  $E[i]$  l'insieme corrente. L'algoritmo (p. 245) sta costruendo due insiemi di coppie: l'insieme  $E[i]$  può ricevere coppie dai passi di *predizione* e *completamento*, mentre l'insieme  $E[i + 1]$  riceve coppie a causa della *scansione*. Il completamento richiede un ripensamento, a causa delle  $\varepsilon$ -regole.

Quando in  $E[i]$  il completamento esamina una coppia completata

$$(q \equiv A \rightarrow \varepsilon \bullet, j)$$

dove lo stato  $q$  della macchina  $M_A$  è iniziale e finale, deve cercare nell'insieme  $E[j]$  le coppie aventi la marca prima della  $A$ . Ma, per le  $\varepsilon$ -regole, il puntatore  $j$  è sempre eguale a  $i$ : infatti tali coppie possono essere soltanto aggiunte dalla *predizione*, la quale assegna al puntatore il valore corrente dell'indice.

Al fine di trattare correttamente questi casi, il *completamento* deve esaminare l'insieme  $E[i]$  parzialmente costruito, poi invocare la *predizione*, la quale deve riattivare il *completamento*, e così via, terminando quando nessuna delle due operazioni ha più nulla da aggiungere all'insieme. Questo modo di procedere è corretto ma inefficiente.

Migliore è il *metodo di Aycock e Horspool*<sup>27</sup>, di cui si indica (in grassetto) la modifica, concernente la sola operazione di predizione. Si ricorda che un nonterminale  $A$  è annullabile (p.60) se da esso può derivare in uno o più passi la stringa vuota.

#### Predizione con $\varepsilon$ -regole

per ogni coppia in  $E[i]$  della forma  $(q \equiv A \rightarrow \alpha \bullet B\gamma, j)$ ,

dove da  $q$  esce l'arco  $q \xrightarrow{B} s$ ,

aggiungi all'insieme  $E[i]$  la coppia  $(r \equiv B \rightarrow \bullet\delta, i)$ ,

dove  $r$  è lo stato iniziale della macchina  $M_B$ .

**Se  $B$  è annullabile, aggiungi a  $E[i]$  anche le coppie**

$(s \equiv A \rightarrow \alpha B \bullet \gamma, j)$

---

<sup>27</sup>Vedasi [5].

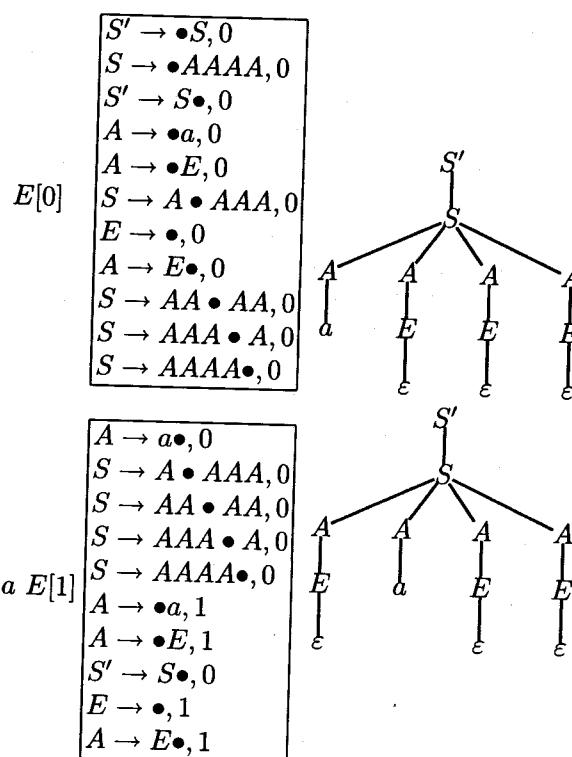
Detto diversamente, l'azione di predizione sposta il pallino dalla sinistra alla destra di un nonterminale, se da esso può derivare la stringa vuota, in accordo con il fatto che la derivazione farebbe sparire il nonterminale stesso.

*Esempio 4.81.* Nella grammatica

$$S' \rightarrow S \quad S \rightarrow AAAA \quad A \rightarrow a \quad A \rightarrow E \quad E \rightarrow \epsilon$$

tutti i nonterminali sono annullabili.

Si mostra la traccia del riconoscimento della stringa  $a$  e due tra le possibili derivazioni:



### Ulteriori sviluppi del metodo di Earley

Si è già detto che le coppie presenti negli insiemi possono essere arricchite con la prospezione, calcolata esattamente come nel metodo  $LR(1)$ . Affiancando a ogni coppia un insieme di prospezione l'algoritmo evita di inserire negli insiemi certe coppie corrispondenti a scelte destinate al fallimento. A prima vista questo può apparire come un perfezionamento da cui ci si potrebbe attendere migliore efficienza, quanto meno per le grammatiche che soddisfano

la condizione  $LR(1)$ ; ma a un esame più approfondito si trova che il numero di coppie da elaborare per molte grammatiche può aumentare anziché diminuire, a causa della differenziazione portata dai caratteri di prospezione. In definitiva, i vantaggi della prospezione sono controversi, e le implementazioni più efficienti del metodo di Earley non ne fanno uso.

Con lievi ritocchi, l'algoritmo di Earley continua a funzionare anche per le grammatiche con regole BNF estese.<sup>28</sup>

## 4.7 Scelta del parsificatore

Per molti linguaggi tecnici esistono compilatori realizzati con entrambi i metodi deterministici  $LL(1)$  e  $LR(1)$  (spesso nella variante  $LALR(1)$ ); ciò attesta che nella maggior parte dei casi la scelta del metodo non è critica.

Se la grammatica di riferimento non soddisfa né le condizioni  $LL$  né quelle  $LR$ , e non si desidera modificarla, è possibile usare una parsificatore di tipo generale, come quello di Earley, certamente più lento e ingombrante d'uno deterministico.

Esistono poi altri approcci, di complessità intermedia tra quelli deterministici e quelli validi per ogni grammatica libera. Partendo dagli algoritmi  $LL(k)$ , vi sono i già ricordati parsificatori che, per decidere la mossa, usano una prospettiva di lunghezza illimitata.

Per l'analisi ascendente, un algoritmo abbastanza diffuso in diverse varianti è quello di Tomita<sup>29</sup> che tratta le grammatiche in cui il numero di scelte non deterministiche è limitato.

Una strategia di parsificazione molto diversa è talvolta applicata quando la grammatica è fortemente ambigua, per evitare di costruire tutti i numerosi alberi sintattici. Infatti molti di essi non corrispondono a interpretazioni sensate del testo, e possono essere eliminati, già in fase di parsificazione, applicando dei controlli semantici. Si parla allora di parsificazione guidata dalla semantica, un concetto che sarà esposto nel prossimo capitolo.

Ritornando al più comune caso deterministico, alcune considerazioni possono guidare il progettista nella scelta del metodo di analisi,  $LL(k)$  o  $LR(1)$ .

Dal punto di vista della velocità di esecuzione, le differenze di prestazioni tra gli analizzatori dei due tipi sono trascurabili.

La famiglia dei linguaggi  $LR(1)$  (anche nella variante  $LALR(1)$ ) è più ampia della famiglia  $LL(1)$ . Di fatto per molti linguaggi tecnici la grammatica di riferimento non è  $LL(1)$ , vuoi per la presenza di ricorsioni sinistre, vuoi per la sovrapposizione tra gli insiemi guida delle alternative. Di conseguenza, per attuare l'analisi discendente, il progettista deve trasformare la grammatica. Di solito le semplici trasformazioni studiate (come la fattorizzazione sinistra p. 4.35) permettono d'ottenere una grammatica equivalente  $LL(k)$ ,  $k \geq 1$ , perché

---

<sup>28</sup>Il modo di fare ciò è delineato da Earley [17].

<sup>29</sup>Si veda [50].

è raro che il linguaggio sorgente non sia  $LL(k)$ . Ma il risultato è abbastanza diverso dalla grammatica di riferimento, spesso è meno leggibile, e comporta l'onere del mantenimento di due versioni della grammatica, se il linguaggio è soggetto a cambiamenti.

Per costruire un analizzatore ascendente è indispensabile l'impiego d'uno strumento generatore (ad es. yacc o bison), mentre gli analizzatori a discesa ricorsiva possono anche essere progettati a mano, visto che il loro codice è sostanzialmente allineato alla struttura delle regole. Il codice dei compilatori a discesa ricorsiva è più facilmente comprensibile, anche da parte di tecnici non specializzati nel progetto dei compilatori.

Anche per il caso  $LL$  vi sono strumenti per il calcolo degli insiemi guida e la generazione delle procedure a discesa ricorsiva, che agevolano il lavoro del progettista. Di solito tali strumenti trattano anche le grammatiche BNF estese con espressioni regolari.

Al contrario gli strumenti  $LR$  o  $LALR$  più comuni non accettano le grammatiche BNF estese, e impongono al progettista una fastidiosa, anche se ovvia conversione, della grammatica originale.

Un analizzatore non è mai un programma isolato, ma deve interfacciarsi con altri algoritmi di traduzione guidata dalla sintassi (trattati nel prossimo capitolo). Anticipando l'esposizione, una proprietà teorica, relativa alle traduzioni calcolabili durante l'analisi sintattica, rende gli analizzatori discendenti un po' più potenti degli altri; in breve, certe traduzioni dal linguaggio sorgente al linguaggio pozzo possono essere svolte aggiungendo delle azioni semantiche nel corpo delle procedure a discesa ricorsiva; mentre ciò non è sempre possibile nell'analisi ascendente. Ciò favorisce la realizzazione di compilatori discendenti capaci di operare in una sola passata.

Una capacità talvolta richiesta a un parsificatore (più in generale a un compilatore) è l'incrementalità, un concetto che si manifesta in due specie assai diverse.

*Incrementalità della grammatica.* Talvolta, in circostanze molto particolari, si deve modificare di continuo la grammatica del linguaggio sorgente, e di conseguenza si deve aggiornare il corrispondente parsificatore. Una situazione del genere si presenta nei cosiddetti linguaggi estensibili, che offrono all'utente la libertà di inventare nuove istruzioni. Diventa allora indispensabile rigenerare automaticamente il parsificatore, dopo ogni modifica della sintassi.<sup>30</sup>

*Parsificazione incrementale.* Più comune è l'esigenza di ricalcolare rapidamente l'albero sintattico, dopo una modifica del testo sorgente.

Un buon ambiente di lavoro per la scrittura di programmi o di documenti strutturati dovrebbe essere interattivo, e permettere la modifica del testo sorgente, senza imporre attese per ottenere la traduzione (compilazione) del testo modificato. Per ridurre il tempo di ritraduzione dopo una modifica, sono stati studiati gli algoritmi di compilazione incrementale. Poiché la compilazione richiede l'analisi sintattica e poi quella semantica, per entrambi i passi sono

---

<sup>30</sup>Per questo problema si veda [25].

stati studiati e utilizzati dei metodi incrementali.

Si supponga che il parsificatore abbia analizzato un testo, accettandolo o diagnosticando degli errori; l'autore lo abbia poi corretto, producendo un nuovo testo, che differisce in pochi punti. Il parsificatore è detto *incrementale* se, per analizzare il nuovo testo, impiega un tempo inferiore a quello che spenderebbe per analizzarlo la prima volta. Affinché ciò sia possibile, l'algoritmo deve conservare il risultato dell'analisi precedente, e modificare l'albero soltanto dove necessario. L'algoritmo incrementale mantiene un'opportuna rappresentazione delle configurazioni attraversate dall'automa a pila, organizzata in modo che, dopo una modifica del testo, sia facile individuare e aggiornare la parte che risente del cambiamento.<sup>31</sup>

---

<sup>31</sup>Sull'analisi sintattica incrementale si veda [21, 31].

## 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 ÷ 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:

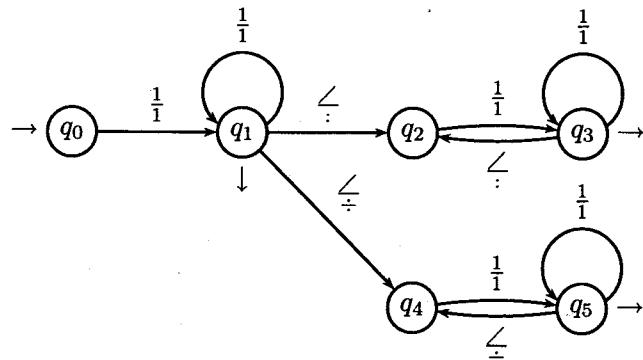
- esiste una stringa  $z \in C^*$  appartenente all'insieme regolare definito da  $r$ ;
- $x$  e  $y$  sono rispettivamente la proiezione di  $z$  sulla prima componente e sulla seconda.

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

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

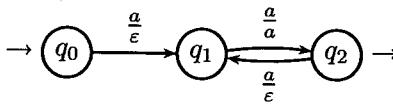
dove  $h_\Sigma$  e  $h_\Delta$  sono rispettivamente le proiezioni dall'alfabeto  $\Sigma \cup \Delta$  agli alfabeti sorgente e pozzo.

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

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

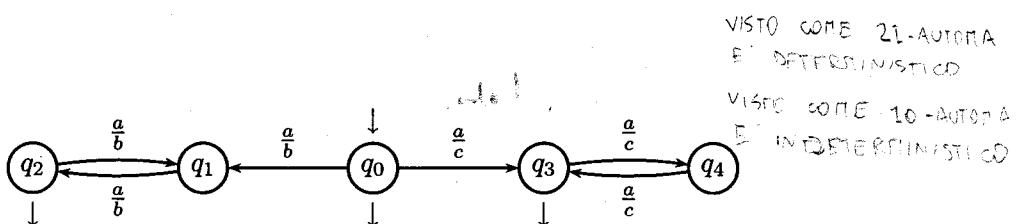
definisce la funzione di traduzione

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

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.

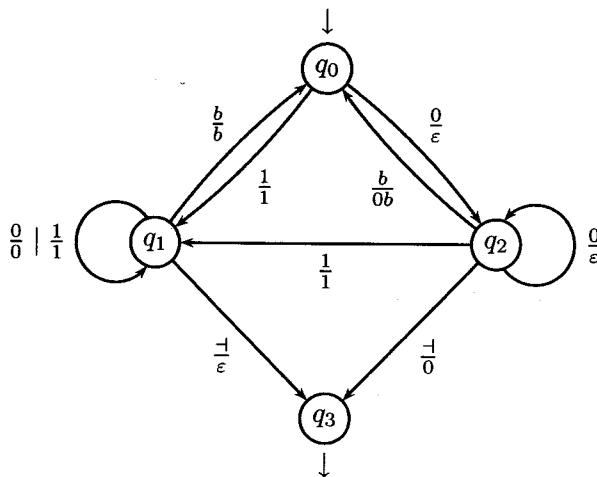
Si dice *funzione sequenziale* una funzione calcolabile mediante un traduttore sequenziale.

*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  $G_2$*

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

$$E \rightarrow T$$

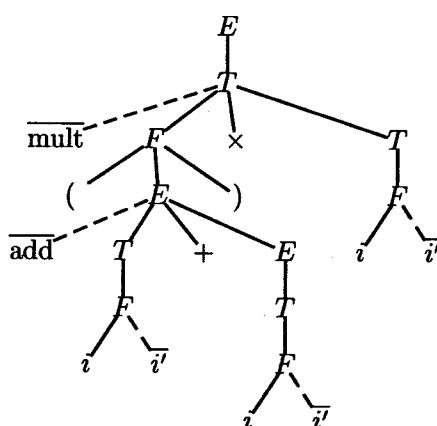
$$T \rightarrow \text{mult } FT$$

$$T \rightarrow F$$

$$F \rightarrow E$$

$$F \rightarrow i'$$

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

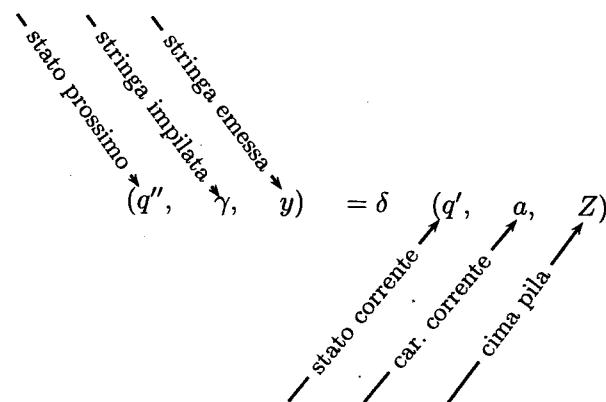
Il concetto astratto d'una *IO-macchina*, che riconosce la stringa sorgente e la traduce nella sua immagine, vale evidentemente anche per le traduzioni libere. Come per il riconoscimento delle frasi dei linguaggi liberi occorre una pila, così per eseguire le traduzioni definite da una grammatica di traduzione,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

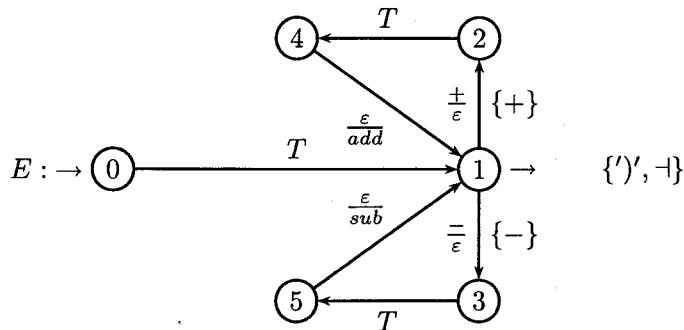
$$E \rightarrow T \left( \frac{+}{\varepsilon} T \frac{\varepsilon}{add} \mid \frac{-}{\varepsilon} T \frac{\varepsilon}{sub} \right)^*$$

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

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

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

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

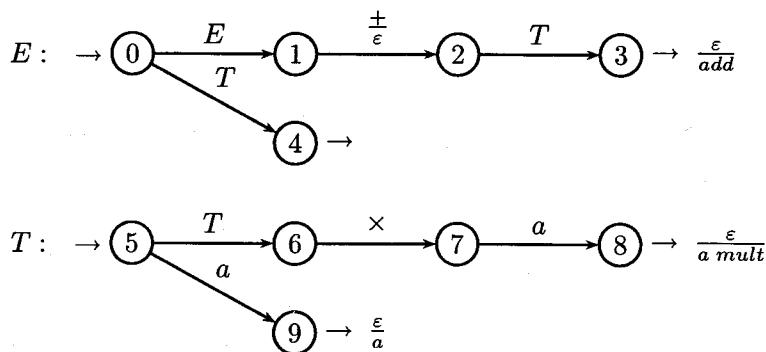
diverse, il che richiederebbe diverse e contradditorie emissioni di caratteri. Questo ragionamento spiega la difficoltà di effettuare l'emissione della traduzione in concomitanza con le mosse di spostamento.

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.

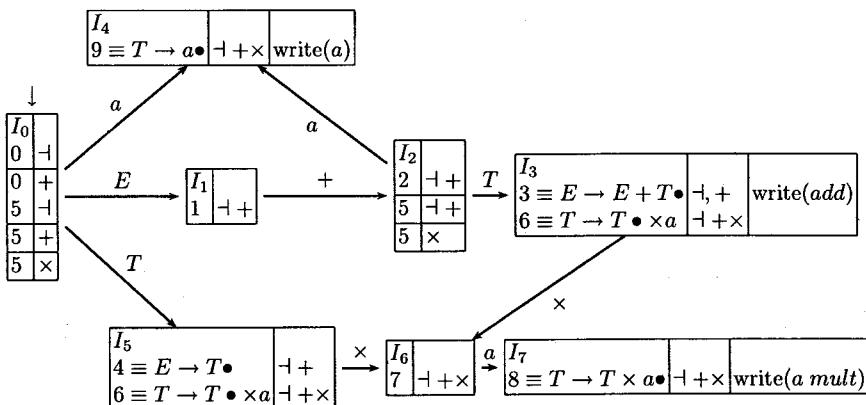
*Esempio 5.27.* Traduzione di espressioni in postfisso.

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.

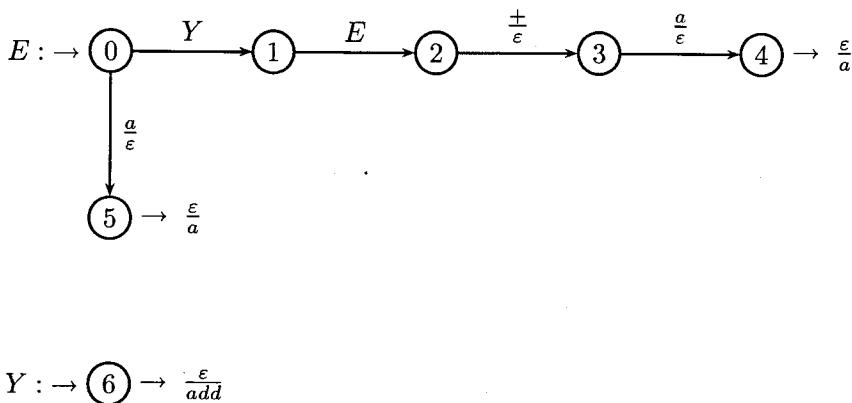
L'introduzione di nonterminali ausiliari come  $Y$  rende meno leggibile la forma postfissa. Inoltre, poiché la normalizzazione aggiunge delle regole vuote alla grammatica sorgente, essa può talvolta perdere la proprietà  $LR(1)$  o richiedere un allungamento della prospezione.

*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.

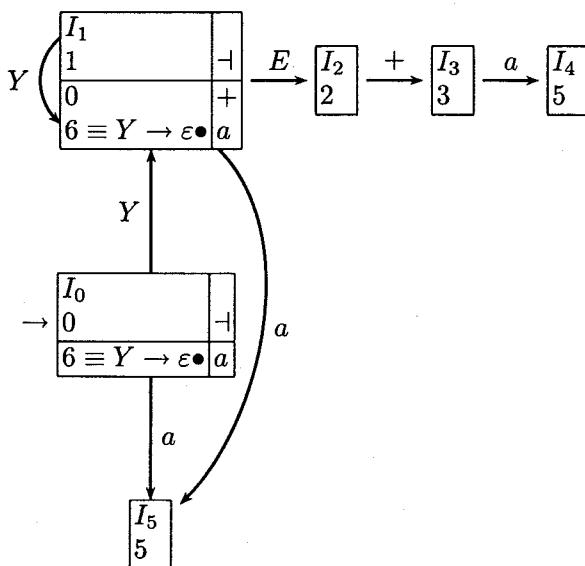
L'interesse della forma postfissa deriva dal fatto che essa rinvia l'emissione della traduzione ai momenti adatti all'analisi sintattica ascendente. Continuando l'es. precedente si sperimenteranno i limiti di questa tecnica di trasformazione della grammatica.

*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  $Y \rightarrow \epsilon$  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  $v_0$ .

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

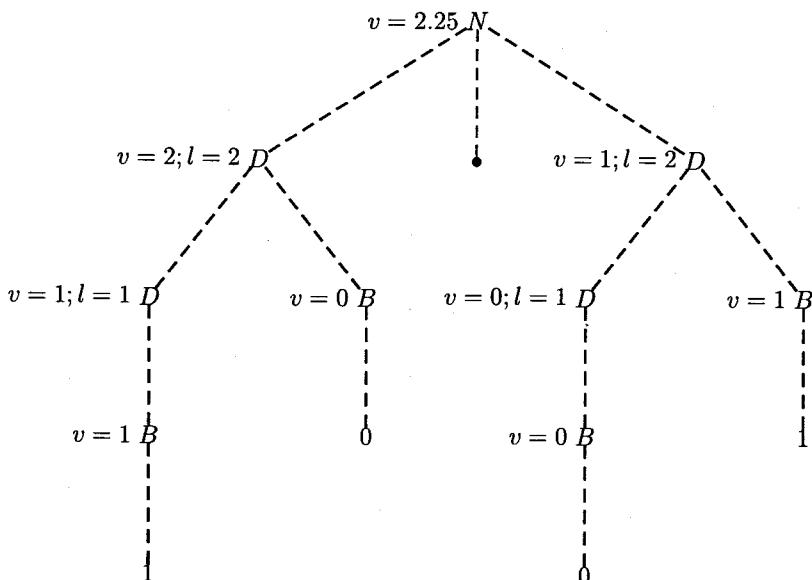
ossia  $v_0$  è associato alla parte sinistra  $N$ ,  $v_1$  al primo nonterminale della parte destra, ecc.. Se in una produzione il simbolo nonterminale non è ripetuto, come avviene con  $N$  nella prima produzione, si può, senza rischio di confusione, scrivere  $v_N$  invece di  $v_0$ .

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.

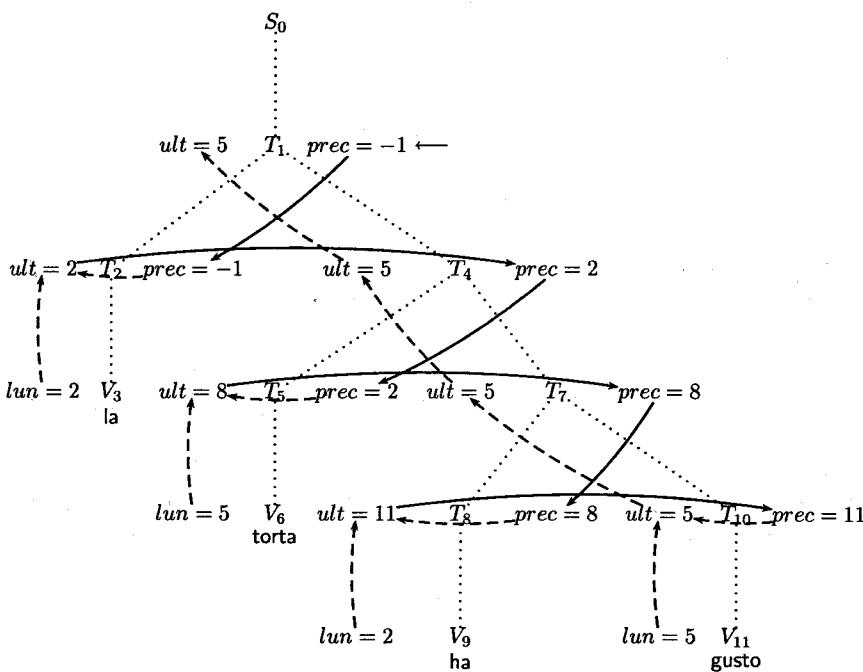
Secondo, la sintassi è ambigua, a causa della regola  $T \rightarrow T \sqcup T$  bilateramente ricorsiva. Ma gli inconvenienti che, in fase di analisi sintattica, derivano dall'ambiguità, qui non sono rilevanti, perché si suppone che al valutatore semantico arrivi dal parsificatore un solo albero sintattico.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

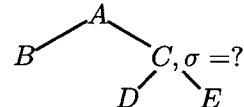
supporto  $p$ , un attributo estraneo alla produzione  $p$  stessa; come avverrebbe modificando le regole 2 nel modo seguente:

| 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*.

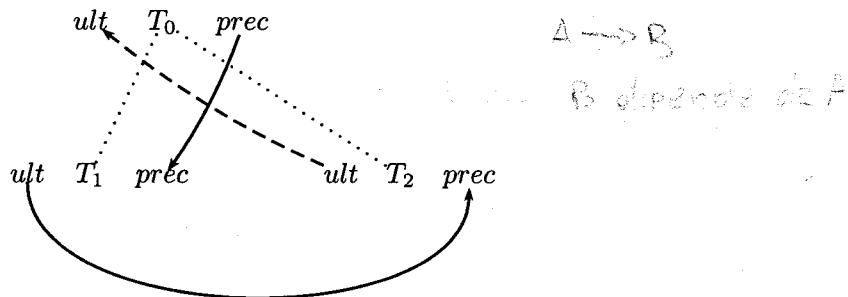
*Esempio 5.38.* Grafi delle dipendenze delle produzioni.

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:

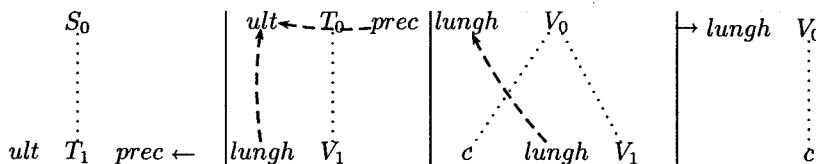
| sintassi                          | attributi destri                           | attributi sinistri   |
|-----------------------------------|--|--|
| 1 $S_0 \rightarrow T_1$           | $prec_1 := -1;$                            |  |
| 2 $T_0 \rightarrow T_1 \perp T_2$ | $prec_1 := prec_0$<br>$prec_2 := ultimo_1$ | $ultimo_0 := ultimo_2$   |
| 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

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

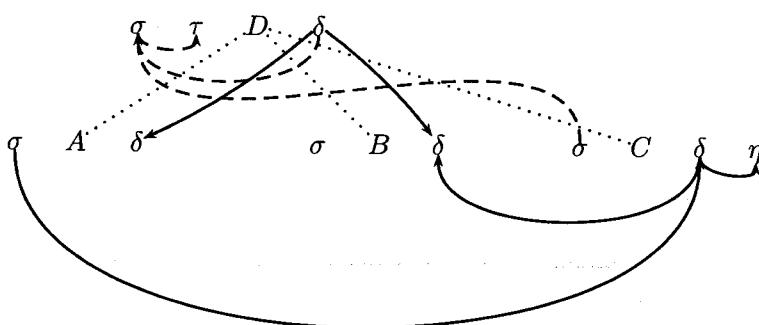
1. Costruisci un ordine topologico, detto  $OTF$ , dei nonterminali  $D_1, D_2 \dots D_r$  risp. al grafo dei fratelli  $frat_p$ .
2. Per ogni simbolo  $D_i, 1 \leq i \leq r$ , costruisci un ordine topologico, detto  $OTD$ , degli attributi destri del simbolo  $D_i$ .

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.

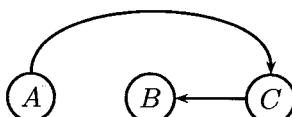
*Esempio 5.44.* Procedura semantica a una scansione.

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  $L$ <sup>21</sup> 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      | $D, B$                        |

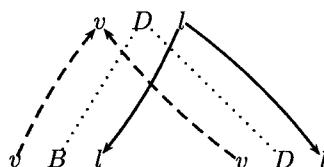
Si noti che il valore d'un bit è pesato con un esponente negativo pari alla sua distanza dal punto frazionale.

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  $D \rightarrow BD$   
*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  $D \rightarrow B$   
*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  $B \rightarrow 0$   
*'1': v<sub>0</sub> := 2<sup>-l<sub>0</sub></sup>*  
     – caso  $B \rightarrow 1$

*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        |                |              |
|-------------|-----------------|----------------|----------------|--------------|
| $I_0$       | $a_3$           | $+ a_5 \vdash$ |                |              |
| $I_0$       | $a_3$           | $I_4$          | $+ a_5 \vdash$ |              |
| $T$         |                 |                |                |              |
| $I_0$       | $v = 3$         | $I_5$          | $+ a_5 \vdash$ |              |
| $o = false$ |                 |                |                |              |
| $E$         |                 |                |                |              |
| $I_0$       | $v = 3$         | $I_1$          | $+ a_5 \vdash$ |              |
| $o = false$ |                 |                |                |              |
| $E$         |                 |                |                |              |
| $I_0$       | $v = 3$         | $I_1 + I_2$    | $a_5 \vdash$   |              |
| $o = false$ |                 |                |                |              |
| $E$         |                 |                |                |              |
| $I_0$       | $v = 3$         | $I_1 + I_2$    | $a_5 \vdash$   | $I_4 \vdash$ |
| $o = false$ |                 |                |                |              |
| $E$         |                 |                |                |              |
| $I_0$       | $v = 3$         | $I_1 + I_2$    | $v = 5 \vdash$ | $I_3 \vdash$ |
| $o = false$ |                 | $T$            |                |              |
| $E$         |                 |                |                |              |
| $I_0$       | $v = 3 + 5 = 8$ |                |                |              |
| $o = false$ |                 |                |                |              |

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   |
|---------------------------|--|
| $S \rightarrow P$         | $t_1 := \emptyset$ – 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 $errato$                                   |
| $R \rightarrow id$        | -- 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 $errato$ |

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:

$$\overbrace{D_1}^{D_1} \quad \overbrace{D_2}^{D_2} \quad \overbrace{D_3}^{D_3} \quad \overbrace{A_4}^{A_4} \quad \overbrace{A_5:ai=true}^{A_5:ai=true} \quad D_6 \quad \overbrace{D_7:dd=true}^{D_7:dd=true} \quad \overbrace{A_8:ai=true}^{A_8:ai=true} \\ a[10] \quad i \quad b \quad i := 4 \quad c := a[i] \quad c[30] \quad i \quad a := c$$

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>$td_2 := td_0$  |
| $V \rightarrow var\_id$   | $td_1 := td_0$  |
| $\text{type\_id} \rightarrow id$  |   |
| $var\_id \rightarrow id$  |   |

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  $\text{type\_id}$  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  $\text{type\_id} \equiv cc_2$  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 :  $b := a + 2$

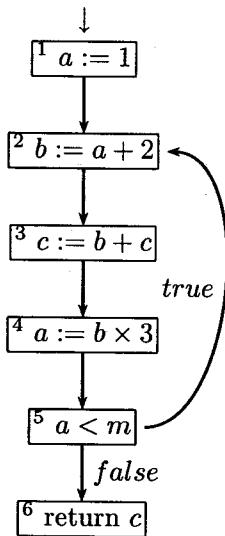
$c := b + c$

$a := b \times 3$

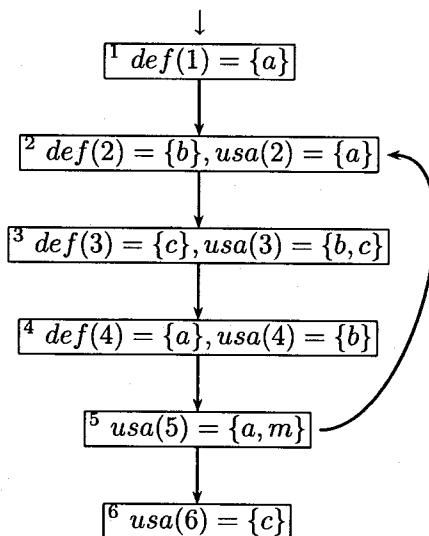
if  $a < m$  goto e1

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 ≥ 0 then istr2 else istr3
```

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

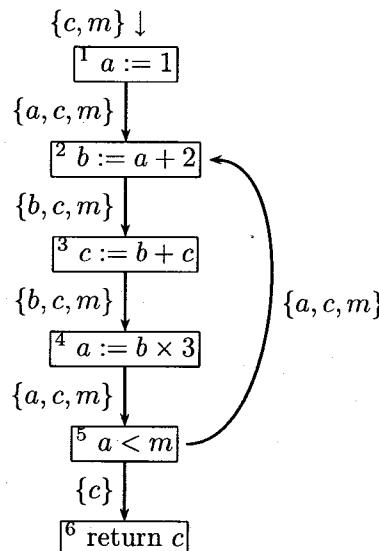
Un compilatore di buona qualità ripassa più volte sul programma intermedio per migliorarlo. Un'analisi molto redditizia ai fini di vari miglioramenti del programma è lo studio degli intervalli di vita<sup>28</sup> delle variabili.

**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.

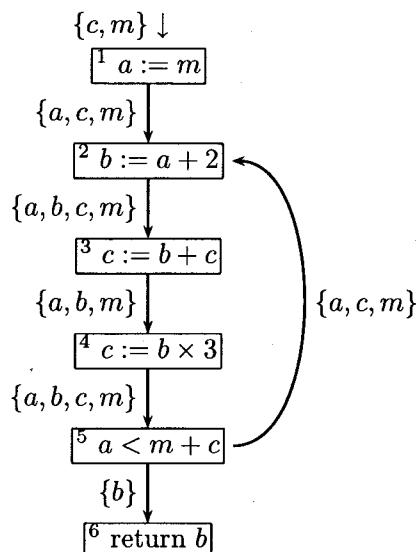
### Definizioni inutili

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.

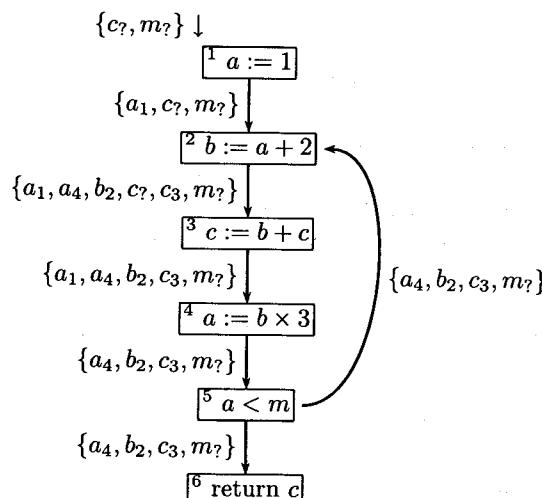
La eq. 5.6 pone nell'uscita di  $p$  le definizioni proprie di  $p$  e quelle raggiungenti l'ingresso di  $p$ , purché non sopprese da  $p$ .

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.

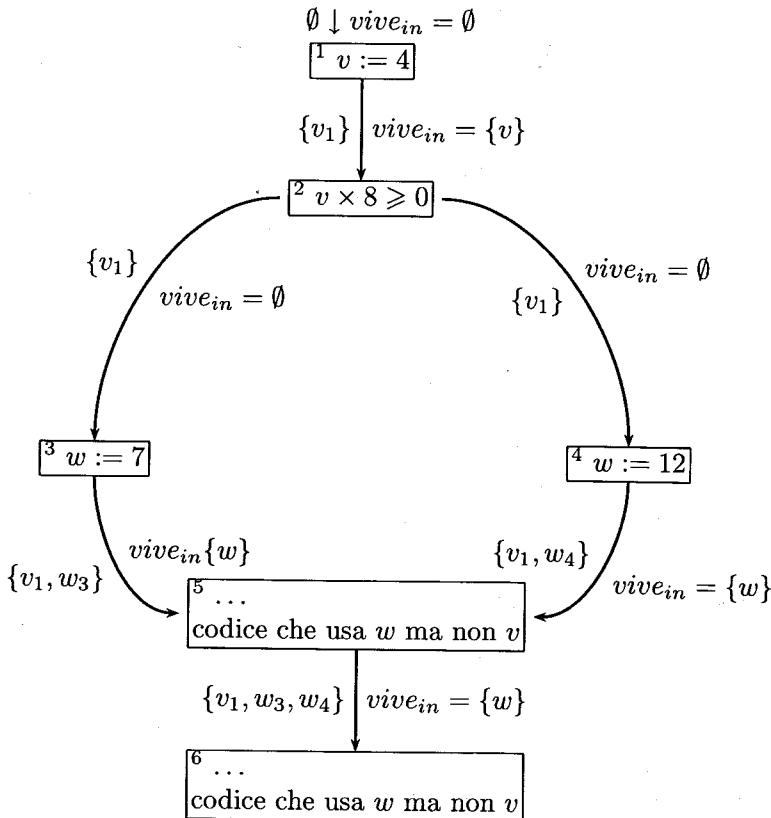
Da questo ragionamento è facile formulare una condizione generale. È lecito sostituire nella istruzione  $p$ , al posto della variabile  $a$  ivi usata, la costante  $k$  se

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.

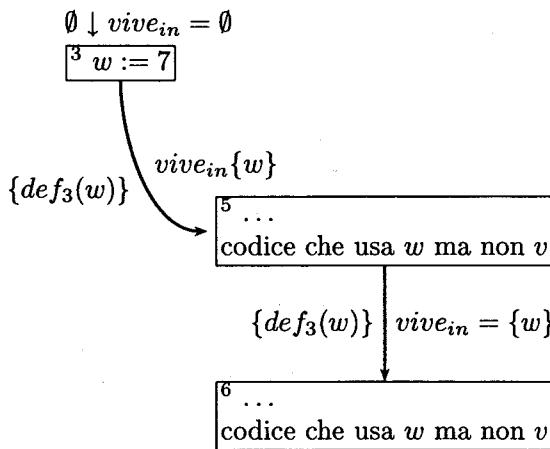
Nella prossima figura sono indicate le definizioni raggiungenti e le variabili vive nei punti del programma.



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*

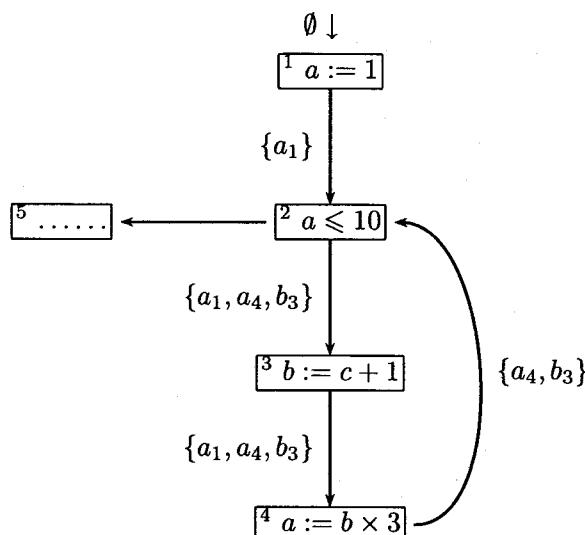
Un'istruzione  $p$  non è ben inizializzata se vale il predicato:

$$\exists q \in pred(p) \text{ tale che } usa(p) \not\subseteq out'(q) \quad (5.8)$$

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].

---

## Riferimenti bibliografici

1. A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: principles, techniques and tools*, Prentice-Hall, 2006.
2. A. Aho and J. Ullman, *The theory of parsing, translation, and compiling*, vol. 1, Prentice-Hall, 1972.
3. \_\_\_\_\_, *The theory of parsing, translation and compiling, volume 2 : Compiling*, Prentice-Hall, 1973.
4. A. Appel, *Modern compiler implementation in Java*, second edition ed., Cambridge University Press, Cambridge, UK, 2002.
5. J. Aycock and R. Horspool, *Practical Earley parsing*, Computer Journal **45** (2002), no. 6, 620–630.
6. J. Beatty, *Two iteration theorems for the ll( $k$ ) languages.*, Theor. Comput. Sci. **12** (1980), 193–228.
7. G. Berry and R. Sethi, *From regular expressions to deterministic automata*, Theor. Comput. Sci. **48** (1986), no. 1, 117–126.
8. J. Berstel, *Transductions and context-free languages*, Teubner, Stuttgart, 1979.
9. J. Berstel and L. Boasson, *Formal properties of XML grammars and languages*, ACTAINF: Acta Informatica **38** (2002).
10. J. Berstel and J.E. Pin, *Local languages and the berry-sethi algorithm*, Theor. Comput. Sci. **155** (1996), no. 2, 439–446.
11. D. Bovet and P. Crescenzi, *Introduction to the theory of complexity.*, Prentice Hall, 1994.
12. J. Cleaveland and R. Uzgalis, *Grammars for programming languages*, North Holland, 1977.
13. S. Crespi Reghizzi, *Le grammatiche ad attributi: semantica dei linguaggi artificiali*, Utet Città Studi, Milano, 1996.
14. S. Crespi Reghizzi, P. Della Vigna, and C. Ghezzi, *Linguaggi formali e compilatori*, ISEDI, Milano, 1976.
15. S. Crespi Reghizzi and M. Pradella, *Tile rewriting grammars and picture languages*, Theor. Comput. Sci. **340** (2005), no. 2, 257–272.
16. R. De Nicola and A. Piperno, *Semantica operazionale e denotazionale dei linguaggi di programmazione*, Utet Città Studi, Milano, 1999.
17. J. Earley, *An efficient context-free parsing algorithm*, CACM: Communications of the ACM **13** (1970), 94–102.

18. J. Engelfriet, *Attribute grammars: Attribute evaluation methods*, pp. 103–138, Cambridge University Press, 1984.
19. R. Floyd and R. Beigel, *The language of machines: an introduction to computability and formal languages*, Computer Science Press, New York, 1994.
20. F. Gecseg and M. Steinby, *Tree languages*, (1997), 1–68.
21. C. Ghezzi and D. Mandrioli, *Incremental parsing*, ACM Transactions on Programming Languages and Systems (TOPLAS) **1** (1979), no. 1.
22. D. Giammarresi and A. Restivo, *Two-dimensional languages*, (1997), 215–267.
23. D. Grune and C. Jacobs, *Parsing techniques: a practical guide*, Vrije Universiteit, Amsterdam, 2004.
24. M. Harrison, *Introduction to formal language theory*, Addison Wesley, Reading, Mass., 1978.
25. J. Heering, P. Klint, and J. Rekers, *Incremental generation of parsers*, IEEE Transactions on Software Engineering **16** (1990), no. 12, 1344–1351.
26. S. Heilbrunner, *A direct complement construction for LR(1) grammars*, Acta Informatica **33** (1996), no. 8, 781–797.
27. J. Hopcroft and J. Ullman, *Formal languages and their relation to automata*, Addison-Wesley, Wokingham, 1969.
28. ———, *Introduction to automata theory, languages, and computation*, Addison-Wesley, Wokingham, 1979.
29. D. Knuth, *Semantics of context-free languages*, Mathematical Systems Theory **2** (1968), no. 2, 127–145.
30. ———, *Semantics of context-free languages, errata corrige*, Mathematical Systems Theory **5** (1971), no. 2, 95–99.
31. J. Larchevêque, *Optimal incremental parsing*, ACM Transactions on Programming Languages and Systems **17** (1995), no. 1, 1–15.
32. D. Mandrioli and C. Ghezzi, *Theoretical foundations of computer science*, John Wiley, New York, 1987.
33. R. McNaughton, *Elementary computability, formal languages and automata*, Prentice-Hall, Englewood Cliffs, NJ, 1982.
34. R. McNaughton and S. Papert, *Counter-free automata*, MIT Press, Cambridge, USA, 1971.
35. S. Morimoto and M. Sassa, *Yet another generation of LALR parsers for regular right part grammars*, Acta Informatica **37** (2001), 671–697.
36. S. Muchnick, *Advanced compiler design and implementation*, Morgan Kaufmann, 1997.
37. P. Naur, *Revised report on the Algorithmic Language ALGOL 60*, Communications Association Computer Machinery **6** (1963), 1–33.
38. F. Nielson, H. Nielson, and C. Hankin, *Principles of program analysis*, Springer-Verlag, 2005.
39. D. Perrin and J.E. Pin, *Infinite words*, Elsevier, New York, 2004.
40. R. Quong and T. Parr, *ANTLR: A predicated-LL( $k$ ) parser generator*, Software—Practice And Experience **25** (1995), 789–810.
41. G. Révész, *Introduction to formal languages*, Dover, New York, 1991.
42. J. Sakarovitch, *Éléments de théorie des automates*, Vuibert informatique, Paris, 2003.
43. A. Salomaa, *Formal languages*, Academic Press, New York, 1973.
44. G. Senizergues,  *$L(A)=L(B)?$  A simplified decidability proof*, TCS: Theoretical Computer Science **281** (2002), 555–608.

45. D. Simovici and R. Tenney, *Theory of formal languages with applications*, World Scientific, Singapore, 1999.
46. S. Sippu and E. Soisalon-Soininen, *Parsing theory: Languages and parsing*, vol. 1, Springer Verlag, Berlin, Heidelberg, New York, 1988.
47. ———, *Parsing theory, vol.II: Parsing theory*, EATCS Monographs on Theoretical Computer Science, vol. 20, Berlin: Springer, 1990.
48. W. Thomas, *Languages, automata, and logic*, (1997), 389–455.
49. K. Thompson, *Regular expression search algorithm*, Communications of the ACM **11** (1968), no. 6, 419–422.
50. M. Tomita, *Efficient parsing for natural language: A fast algorithm for practical systems*, Kluwer, Boston, 1986.
51. A. Van Wijngarten, *Report on the Algorithmic Language ALGOL 68*, Numerische Mathematik **22** (1969), 79–218.
52. B. Watson, *A taxonomy of finite automata minimization algorithms*, Report, Department of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands, 1994.
53. G. Winskel, *The formal semantics of programming languages*, MIT Press, Cambridge MA, 1993.
54. W. Yang, *Mealy machines are a better model of lexical analyzers*, Computer Languages **22** (1996), 27–38.

---

## Indice analitico

- aciclica
  - grammatica con attributi, 306
- adeguatezza strutturale, 58
- albero
  - decorato, 298, 300
  - scheletrico, 42
  - condensato, 42
- sintattico, 40
- sintattico astratto, 278
- alfabeto, 8
- alfabeto unario, 76
- algoritmo
  - di decisione, 94
  - di riconoscimento, 94
- algoritmo di riconoscimento
  - indeterministico, 174
- allungamento della prospettiva, 194
- ambiente, 321
- ambiguità, 47
  - delle frasi condizionali, 55
  - di automa, 115
  - di concatenamento, 52
  - di grammatica libera estesa, 86
  - di istruzione condizionale, 55, 279
  - di ricorsione bilaterale, 49
  - di unione, 50
  - grado di , 48
  - inerente, 56
- e indeterminismo dell'automa, 165
- analisi
  - lessicale, 312
  - semantica, 296
  - a più scansioni, 311
- a una scansione, 308
- sintattica
  - ascendente deterministica, 199
  - ascendente e discendente, 168
  - discendente deterministica, 176, 186
  - guidata dalla semantica, 326
  - incrementale, 254
  - LR(0), 199
  - LR(1) per EBNF, 231
- sintattica-semantica integrata, 312
- statica
  - interprocedurale, 331
  - intraprocedurale, 331
- statica dei programmi, 324, 330, 346
- analizzatore
  - a spostamento e riduzione, 206
  - di Earley, 240
  - ELR(1), 236
  - LR(0), 207
  - LR(1), 222
  - sintattico-semantico, 315
- annidata
  - struttura, 30
- annullabile
  - espressione regolare, 131
- ANTLR, 196
- aperiodico
  - linguaggio, 141
- approssimazione cautelativa, 333
- assegnamento inutile, 334
- assegnazione dei registri, 338
- astrazione linguistica, 25
- attributo

- destro, 299
- sua utilità, 301
- ereditato, 299
- esterno, 303
- interno, 303
- lessicale, 313
- semantico, 296
- sinistro, 299
- sintetizzato, 299
- autoinclusione, 44
- automa
  - 2I, 265
  - di Rabin e Scott, 265
  - a due ingressi, 264, 265
  - a pila, 146
    - complessità di calcolo dell', 150
    - da grammatica, 147
    - definizione dell', 150
    - deterministico, 160
    - indeterminismo dell', 160
    - modalità accettazione dell', 153
  - a pila deterministico
  - sottoclassi dell', 166
  - a pila deterministico  $LR(0)$ , 200
  - a pila indeterministico, 147
  - a pila IO, 280
  - ambiguo, 115
  - calcolo espr. regolare, 116
  - con mosse spontanee, 112
  - da espressione regolare, 123, 126
  - del grafo di controllo, 332
  - deterministico
    - da espr. regolare, 133
  - di linguaggio locale, 127
  - finito, 98
  - finito deterministico, 100
  - generale, 95
  - indeterministico, 110
    - determinizzazione di, 118
  - IO, 268
  - IO sequenziale, 270
  - minimo, 103
  - monodirezionale, 97
  - pulito, 102, 333
- automi
  - prodotto di, 138
- Aycock e Horspool, 251
- Berry e Sethi, 133
- Berstel, 126, 260
- binario
  - operatore, 275
- BMC, 116
- Brzozowski e McCluskey, 116
- BS, 133
- cancellazione di Dyck, 44
- cardinalità
  - di linguaggio, 8
- chiusura
  - $LR(0)$ , 201
  - $LR(1)$ , 212
  - rispetto alle traduzioni, 293
- cifrario, 262
- classe lessicale, 313
  - finita e non, 313
- classificazione
  - di Chomsky, 87
- codice irraggiungibile, 333, 343
- codici e ambiguità, 53
- compilazione
  - a più stadi semantici, 311
- complemento
  - di linguaggio, 14
- complessità di calcolo
  - dell'alg. Earley, 250
- completamento
  - Earley, 242
- concatenamento, 9
  - di linguaggi, 12
  - di linguaggi di Dyck, 52
  - di linguaggi liberi, 46, 79
- condizionale
  - istruzione, 55, 279
- condizione
  - $ELR(1)$ , 235
  - L, 315
  - $LALR(1)$ , 220
  - $LL(1)$ , 182
  - $LL(k)$ 
    - violazione della, 190
  - $LR(0)$ , 203
  - $LR(1)$ , 216
  - per valutabilità ascendente, 320
- conflitto
  - riduzione-riduzione, 226
  - riduzione-spostamento, 227
- confronti REG,LL(k),LR(k), 224

- confronti tra traduttori sintattici, 293
- controlli semantici, 320
- controllo dei tipi, 321
- copiatura, 61
- corrispondenza
  - automa finito e grammatica lineare a sin., 115
  - automa finito e grammatica unilineare, 113
- costanti decimali
  - automa, 99
- decomposizione del compilatore, 311
- definizione
  - di variabile, 331
  - inutile, 338
  - raggiungente, 339
  - sempre, 345
- definizioni raggiungenti, 343
- derivazione
  - autoincassata, 75
  - autoinclusiva, 75
  - circolare, 38
  - circolare e ambiguità, 278
  - destra inversa, 208
  - destra o sinistra, 43
  - di espressione regolare, 21
  - in grammatica libera estesa, 85
  - per grammatica, 35
- DET, 160
- DET=LR(1), 224
- diagramma sintattico, 172
- diagramma stato-transizione, 99
  - di automa a pila, 152
- differenza
  - di linguaggi, 14
- digrammi
  - insieme dei, 126
- discesa ricorsiva, 188, 314
- disponibile
  - variabile, 344
- distinguibilità tra stati, 103
- Dyck, 44
- Earley, 239
  - algoritmo di, 244
  - regole vuote, 251
- elemento lessicale, 312
- eliminazione ricorsioni sinistra, 65
- eliminazione stati inutili, 102
- ELR(1)
  - condizione, 235
- equazioni
  - di flusso, 335
  - per definizioni raggiungenti, 340
  - per variabili vive, 336
  - soluzione iterativa, 336
- equazioni di grammatica unilineare, 72
- equivalenza
  - di grammatiche
    - in senso debole, 58
    - in senso strutturale, 59
- errori
  - nell'analisi sintattica, 189
  - semantici dinamici, 324
  - semantici statici, 321
- espansione
  - di nonterminale, 60
- espressione regolare, 18
  - ambigua, 22
  - annullabile, 131
  - con complemento, 136
  - con intersezione, 136
  - costruzione dell'automa, 123
  - estesa, 23, 140
  - lineare, 129
  - metodo di Thompson, 123
  - numerata, 132
- espressione regolare di traduzione, 264
- espressioni aritmetiche
  - rappresentazioni delle, 275
- etichetta del calcolo, 110
- famiglia
  - DET, 160, 208, 224
  - FIN, 19
  - LIB, 69
  - REG, 69
- fattorizzazione sinistra, 191
- FIN, 19
- fini
  - insieme delle, 126
- flex, 273
- forma normale
  - di Chomsky, 63
  - di Greibach, 66, 188
  - non annullabile, 61
  - non ricorsiva a sinistra, 64

- postfissa, 289
- senza copiature, 61
- senza regole vuote, 60
- frase, 8
- frasi condizionali
  - ambiguità delle, 55
- fronte, 324
- funzione
  - di traduzione, 260
  - semantica, 302
  - sequenziale, 271
- generazione del codice, 324
- Glushkov, McNaughton e Yamada, 126
- GMY, 126
- grado di ambiguità, 48
- grafo
  - dei fratelli, 309
  - delle dipendenze, 300, 308
    - di albero, 305
    - di funzione semantica, 304
    - di produzione, 304
  - di controllo del flusso, 330, 331
- grammatica
  - ambigua, 48
    - da espr. regolare ambigua, 69
  - con attributi, 296
    - a una scansione, 308, 309
  - aciclica, 306
  - definizione, 302
    - soltanto sinistri, 318
  - contestuale, 87
  - deterministica semplice, 166
  - di linguaggio regolare, 68
  - di tipo 0, 87
  - di tipo 1, 87
  - di traduzione, 268, 273
    - e traduttore a pila, 281
    - postfissa, 289
  - di traduzione EBNF, 286
  - di Van Wijngarten, 92
  - erronea, 37
  - espressioni aritmetiche, 40
  - libera
    - da automa a pila, 156
    - definizione di, 32
    - estesa o EBNF, 83
    - introduzione a, 30
  - libera estesa o EBNF, 170, 229
- lineare, 69
- lineare a destra, 70, 106
- lineare a sinistra, 70, 115
- LL(1), 182
- LR(0), 201
- LR(k), 224
- non ricorsiva a sinistra, 64
- parentesizzata, 45
- pozzo, 273
- pulita, 37
- rappresentazioni, 33
- sorgente, 273
- sorgente ambigua, 279
- strettamente unilineare, 71
- tipo 3, 70
- unilineare, 70
- guida
  - insieme, 180
- identità di Arden, 72
- inclusione
  - di REG in LIB, 69
- indeterminismo
  - motivazione dell', 108
- indisponibile
  - variabile, 344
- indistinguibilità tra stati, 103
- infinitezza del linguaggio
  - derivazioni ricorsive, 39
- infisso
  - operatore, 276
- inizi
  - insieme degli, 126, 177
- inizializzazione di variabile, 345
- insieme
  - degli inizi, 177
  - dei seguiti, 133, 177
  - delle parti finite
    - algoritmo dell', 121
  - di prospettiva, 212
  - guida, 180
- insieme guida
  - calcolo semplificato dell', 185
- insiemi locali
  - calcolo degli, 131
- interferenza tra variabili, 338
- interpretazione semantica, 58, 59
- intersezione
  - di linguaggi liberi, 79

- di linguaggi regolari, 137
  - automa prodotto dell', 139
  - di linguaggio libero e regolare, 79, 158
- intervallo di vita, 335
- istruzione condizionale, 279
  - ambigua, 55
- istruzioni di controllo
  - traduzione in salti, 325
- LALR(1), 220
  - condizione, 220
- lessema, 312
- lex, 273
- LIB, 69
- LIB=PILA, 155
- linguaggi locali
  - composizione di, 128
- linguaggio
  - a parentesi, 167
  - aperiodico, 141
  - artificiale, 5
  - complemento
    - riconoscitore del, 136
  - con due esponenti eguali, 74
  - con prefissi, 210
  - con tre esponenti eguali, 76, 89
  - definito da espr. regolare, 21
  - del grafo di controllo, 332
  - delle repliche, 77, 90
  - deterministico non  $LL(k)$ , 196
  - di Dyck, 44, 210, 213, 216, 220
  - finito, 8
  - formale, 6
  - formalizzato, 5
  - generato, 36
  - infinito, 8
  - intermedio, 324
  - libero
    - di alfabeto unario, 76
    - e automa a pila, 155
  - locale, 127, 141, 333
  - localmente testabile, 127
  - pozzo, 260
    - o immagine, 82
  - regolare, 19
  - senza contatore, 141
  - senza prefissi, 209
  - senza stella, 141
  - sorgente, 260
- speculare
  - riconoscitore del, 109
- universale, 13
- vuoto, 9
- lista
  - a più livelli, 28
  - a precedenze, 28
  - astratta, 26
  - con separatori, 26
  - concreta, 26
- livello
  - lessicale, 313
  - sintattico, 313
- LL(1), 182
- LL( $k$ )
  - condizione, 190
  - trasformazione della grammatica, 190
- locale
  - linguaggio, 127, 141
- LR(0)
  - condizione, 203
- LR(1)
  - condizione, 216
  - trasformazione della grammatica, 226
- Lukasiewicz, 276
- lunghezza
  - di stringa, 9
- macchina di Turing, 87, 97
- macchina pilota
  - del traduttore, 288
- LR(0), 202
- LR(1), 212
- marca di apertura, 26
- marca di chiusura, 26, 279
- massimo prefisso riconosciuto, 314
- McNaughton, 142
- metagrammatica, 32
- metalinguaggio
  - delle espr. regolari, 31
  - semantico, 295, 303
- minimizzazione di automa, 104
- mistofisso
  - operatore, 276
- monoide libero, 15
- Morimoto e Sassa, 231
- mossa epsilon, 107
- mossa spontanea, 107
  - eliminazione di, 119

- Nerode, 103
- Nivat, 267
- nonterminale
  - annullabile, 60
  - ben definito, 37
  - raggiungibile, 37
- nucleo del macrostato, 231
- omomorfismo
  - a parole, 81
  - alfabetico, 262
- omomorfismo alfabetico, 80
- operatore
  - binario, 275
  - infisso, 276
  - mistofisso, 276
  - postfisso, 276
  - prefisso, 276
  - unario, 275
  - variadicò, 275
- operazioni
  - insiemistiche, 13
- ordinamento topologico, 306
- ottimizzazione, 330
- palindromi, 31
- Papert, 142
- parentesi ridondanti, 278
- parsificatore
  - ascendente con attributi, 317
  - con procedure ricorsive, 188
  - e attributi, 314
  - con traduzione, 284
  - scelta del, 253
- pilota LR(0)
  - macchina, 202
- polacca
  - rappresentazione, 276
- postfissa
  - grammatica di traduzione, 289
- postfisso
  - operatore, 276
- potenza
  - di linguaggio, 12
  - di stringa, 11
- precedenza
  - tra operatori, 11
- predicato
  - guida, 327
- semantico, 321
- predizione (Earley), 241
- prefisso, 10
  - operatore, 276
- privo di prefissi
  - linguaggio, 12
- procedura semantica, 311
- prodotto di automi, 138
- propagazione delle costanti, 342
- proprietà di chiusura
  - di REG, 24
  - di DET, 161
  - di LIB, 46
  - di LIB e REG, 78
  - risp. a sostituzione, 82
- proprietà di pompaggio
  - di linguaggi regolari, 73
- prospezione
  - allungamento della, 194
  - insieme di, 212
- pulizia della grammatica, 37
- quoziente
  - di linguaggi, 17
  - sinistro, 228
- Rabin e Scott, 265
- rappresentazione intermedia, 330
- REG, 69
- registri
  - assegnazione dei, 338
- regola
  - a operatori, 34
  - di cancellazione di Dyck, 44
  - di copiatura, 34
  - eliminazione di, 119
  - di sottocategorizzazione, 34, 59
  - lineare, 34
    - a destra, 34
    - a sinistra, 34
  - normale
    - di Chomsky, 34
    - di Greibach, 34
  - omogenea binaria, 63
  - ricorsiva, 34
    - a destra, 34
    - a sinistra, 34
    - a sinistra e cond. LL(1), 186
  - terminale, 34

- vuota, 34
- relazione di traduzione, 260
- replica, 77
- Reps, 299
- resto del macrostato, 231
- rete di automi finiti, 170
- retro, 325
- ricerca d'una parola, 111
- riconoscitore a discesa ricorsiva
  - deterministico, 186
  - indeterministico, 175
- riconoscitore dei prefissi ascendenti, 202
- ricorsione
  - sinistra, 64
    - immediata, 64
    - sinistra e cond. LR(1), 228
  - non immediata, 65
- riflessione, 11
  - di linguaggio, 12
  - di linguaggio libero, 47, 79
- Sakarovitch, 260
- scansione (Earley), 242
- scansione anticipata, 227
- scansore, 312
- schedulazione, 330
- schema di traduzione sintattica, 273
- segmentazione in lessemi, 314
- seguiti
  - insieme dei, 133, 177
- semantica, 257, 295, 296
- senza contatore
  - linguaggio, 141
- senza stella
  - linguaggio, 141
- significato, 296
- sostituzione di linguaggio, 27, 81
- sottocategorizzazione, 61
- sottostringa, 10
- spostamento
  - apertura, 231
  - e riduzione, 206
  - proseguimento, 231
- stato
  - accessibile, 102
  - d'errore, 101
  - iniziale unico, 113
  - post-accessibile, 102
- pozzo, 101
- raggiungibile, 102
- utile, 102
- stella, 14
  - di linguaggi liberi, 46
  - di linguaggio libero, 79
  - proprietà della, 15
- stringa, 8
- struttura
  - a blocchi, 30
  - annidata, 30
- suffisso, 10
- supporto sintattico, 297
- tabella dei simboli, 321
- teorema di Nivat, 267
- testabilità locale, 127
- Thompson, 123
- Tomita, 253
- traduttore
  - a pila, 280
    - da grammatica di traduzione, 281
    - indeterministico, 283
    - con procedure ricorsive, 286
    - con prospettiva, 273
    - finito, 268
    - sequenziale, 270
  - traduzione, 261
    - a più valori, 261
    - biunivoca, 261
    - deterministica ascendente, 287
    - deterministica discendente, 284
    - deterministica LL(1), 285
    - funzione di, 260
    - guidata dalla sintassi, 295
    - iniettiva, 261
    - inversa, 261
    - libera dal contesto, 274
    - proprietà di chiusura, 293
    - regolare, 263
    - relazione di, 260
    - semantica, 295
    - sequenziale
      - con passate riflesse, 272
    - sintattica pura, 273
    - suriettiva, 261
    - totale, 261
    - univoca, 261, 279
  - traslitterazione, 80, 262

- a parole, 81
- trattamento errori
  - nell'analisi sintattica, 189
- tree pattern matching, 324
- tronco, 324
- unario
  - operatore, 275
- unione
  - di linguaggi liberi, 45, 79
- uso
  - di variabile, 331
- valutatore semantico, 296
  - a più scansioni, 311
  - a una scansione, 308
  - costruzione, 309
- con procedure ricorsive, 311
- Van Wijngarten, 92
- variabile disponibile, 344
- variabile indisponibile, 344
- variabile non inizializzata, 345
- variabile viva, 334, 343
- variadico
  - operatore, 275
- vocabolario, 8
- vuota
  - stringa, 10
- VW grammatica, 92
- XML, 44, 167
- Yang, 273



Area S.B.A.  
Biblioteca Centrale  
di Ingegneria

063817