

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.¹

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

¹Si 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 parole². 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*³ 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

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

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

²Nei trattati matematici il termine parola è sinonimo di stringa.

³Anche 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 ϵ 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 ϵ , 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 Σ , 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 Σ , 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 Σ 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 Σ , la sua stella Σ^* contiene tutte le stringhe⁴ che possono essere costruite concatenando i caratteri terminali. Tale linguaggio è dunque il *linguaggio universale* di alfabeto Σ , precedentemente definito.⁵

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 Σ .

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

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

⁴Le frasi di Σ^* 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.

⁵Esso 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

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

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

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

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:

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

Esempio 2.12.

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

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

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 sequenziale⁶ in un certo stato; descrivendo il lessico dei linguaggi di programmazione con certe semplici grammatiche; e analizzando il comportamento di modelli semplificati di

⁶Una 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.⁷

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 unitari⁸ $\{a_1\}, \{a_2\}, \dots, \{a_n\}$ 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 Σ e con i metasimboli⁹

. 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 ε (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 Σ , secondo la seguente tabella di corrispondenza:

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

⁸Ossia contenenti una sola stringa.

⁹Al 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_r</i>
1. ε	$\{\varepsilon\}$
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. β di e' una e.r. scelta da essa.

Definizione 2.17. Derivazione¹⁰

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 β è una s.e. di e' , e δ è una scelta di β .

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:

¹⁰Anche 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 ¹¹ 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

¹¹ Il simbolo ε 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 ϑ 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* ϑ se il linguaggio risultante dall'applicazione di ϑ 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 istr₁; istr₂; ... istr_n 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(par₁,par₂,...,par_n)

i e s f

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

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

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 Σ 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 Δ 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 grammatica¹² o *sintassi* generativa.

¹²Il 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:

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

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.:

$$pal \Rightarrow a \ pal \ a \Rightarrow ab \ pal \ ba \Rightarrow abb \ pal \ bba \Rightarrow \dots$$

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:

$$abb \ pal \ bba \Rightarrow abbebb = abbbba$$

(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:

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

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 Σ :

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

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 BNF¹³) G è definita da quattro entità:

1. V , alfabeto non terminale, è un insieme di simboli detti (metasimboli) nonterminali.
2. Σ , 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

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

¹³Backus 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 ', ' ϵ ' 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	$< frase if > \rightarrow$ $if < cond > then < frase >$ $else < frase >$
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.: a then 'a' 'then'	$frase_if \rightarrow$ $if cond then frase else frase$ opp. $frase_if \rightarrow$ $'if' cond 'then' frase 'else' frase$
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 Σ^* (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, α, β denotano delle stringhe (anche vuote) che possono contenere simboli sia terminali che nonterminali; inoltre la lettera σ 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 η e δ sono stringhe qualsiasi anche vuote. Sia $A \rightarrow \alpha$ una regola di G , e sia $\gamma = \eta \alpha \delta$ la stringa ottenuta sostituendo in β la parte destra α al posto del nonterminale A .

La relazione intercorrente tra le due stringhe è detta di *derivazione*. Si dice che β deriva γ 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 α 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 β 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 α, β 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

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

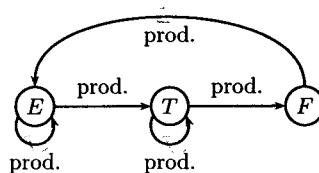
ha le regole

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

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*.

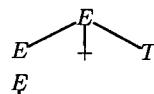
Preso un nodo interno N , il *sottoalbero* di radice N è l'albero avente N come radice e comprendente tutti figli di N , i figli dei figli, ecc., cioè tutti i *discendenti* di N . I nodi privi di figli sono detti *foglie* o *nodi terminali*. La sequenza delle foglie lette da sinistra a destra (con riferimento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

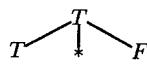
$$1. E \rightarrow E + T$$



$$2. E \rightarrow T$$



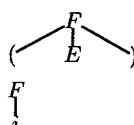
$$3. T \rightarrow T * F$$



$$4. T \rightarrow F$$



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



$$6. F \rightarrow i$$

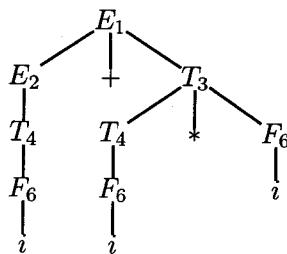


$$6. F \rightarrow i$$

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:

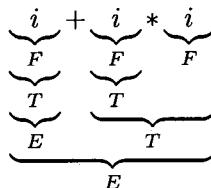
$$E \xrightarrow{1} E+T \xrightarrow{3} E+T*F \xrightarrow{6} E+T*i \xrightarrow{4} E+F*i \xrightarrow{6} E+i*i \xrightarrow{2} T+i*i \xrightarrow{4} F+i*i \xrightarrow{6} i+i*i \quad (2.3)$$

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*.

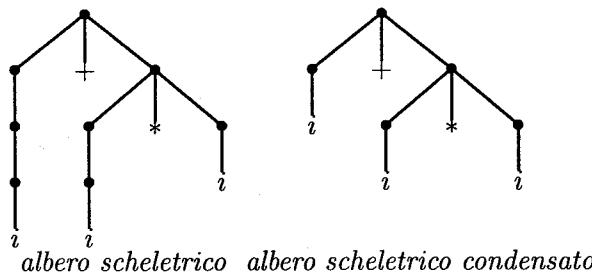
L'albero sintattico d'una frase x può anche essere codificato in forma di testo, racchiudendo ogni sottoalbero tra una coppia di parentesi¹⁴, etichettata con il nome del simbolo nonterminale. All'albero precedente corrisponde la *espressione parentetica*

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

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 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.¹⁵

¹⁴Si suppone che le parentesi quadre non appartengano all'alfabeto terminale.

¹⁵Un 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 tedesca¹⁶ 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.

¹⁶L'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 Σ 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 $S \rightarrow SS_1 | \varepsilon$

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 $S \rightarrow SS_1 | S_1$

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
$S_1 \rightarrow XC$	$S_2 \rightarrow AY$
$X \rightarrow aXb \mid \varepsilon$	$Y \rightarrow bYc \mid \varepsilon$
$C \rightarrow cC \mid \varepsilon$	$A \rightarrow aA \mid \varepsilon$

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'' :

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

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:

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

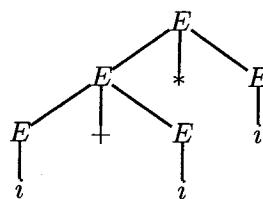
Le derivazioni sinistre

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

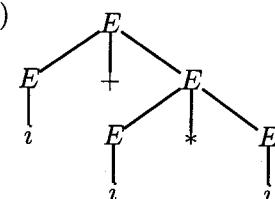
$$E \Rightarrow E + E \Rightarrow i + E \Rightarrow i + E * E \Rightarrow i + i * E \Rightarrow i + i * i \quad (2.6)$$

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.

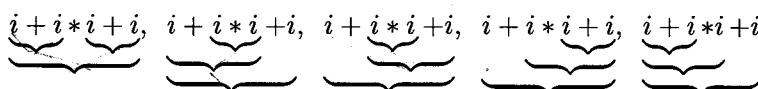
Invece la grammatica G dell'esempio 2.39 genera le stesse frasi con una, e una sola, derivazione sinistra, dunque tali frasi non sono ambigue. Si potrebbe vedere che nessuna frase è ambigua per la grammatica G , che quindi risulta inambigua.

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 decidibile¹⁷. 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,

¹⁷ 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

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

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: ε . 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 Σ , 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 $\Sigma = \{0, 1\}$:

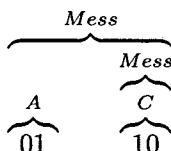
$$\Gamma = \{ \overbrace{A}^{01}, \overbrace{C}^{10}, \overbrace{E}^{11}, \overbrace{R}^{001} \}$$

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 60,¹⁸ 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* { *if b then a else a* } } *if b then* { *if b then a* } *else a*

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_E* definisce un condizionale doppio e tale che i due eventuali condizionali in esso presenti siano ancora del tipo *S_E*. L'altra categoria, *S_T*, definisce un condizionale semplice, oppure un condizionale doppio, tale che il primo condizionale contenuto sia del tipo *S_E* e il secondo sia del tipo *S_T*; 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_E* può precedere *else*, mentre solo *S_T* definisce un condizionale semplice, esclude l'albero scheletro indesiderabile di destra.

¹⁸Nella 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.¹⁹

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.

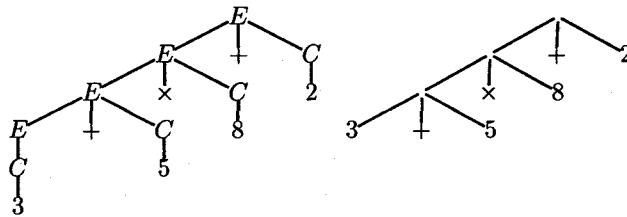
- Prima grammatica G_1 :

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

L'albero sintattico della frase precedente è:



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

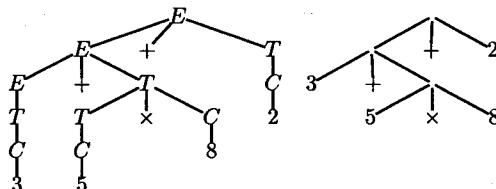


Nella struttura scheletrica di destra sono omessi i nonterminali; anche le regole di categorizzazione, come $E \rightarrow C$, sono state tolte poiché non producono ramificazioni nell'albero.

- 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 5×8 , 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 precedente²⁰:

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

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.

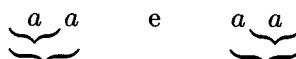
Equivalenza strutturale in senso lato

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.

²⁰Questa 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 β_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'algoritmo²¹ 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:

²¹La 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 $A_2 \rightarrow A_1d$ le regole ottenute con l'espansione di A_1 ottenendo:</p> <p>Elimina la s-ricorsione immediata, ottenendo G'_3:</p>	$\begin{array}{l} A_1 \rightarrow A_2a \mid b \\ A_2 \rightarrow A_2c \mid A_2ad \mid bd \mid e \end{array}$ $\begin{array}{l} A_1 \rightarrow A_2a \mid b \\ A_2 \rightarrow bdA'_2 \mid eA'_2 \mid bd \mid e \\ A'_2 \rightarrow cA'_2 \mid adA'_2 \mid c \mid ad \end{array}$

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.

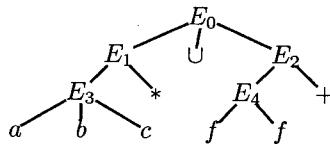
Lo schema di corrispondenza andrà applicato assegnando come nome al non-terminale E un numero che contraddistingue la sottoespressione considerata. L'assioma è associato al primo passo della decomposizione. È sufficiente un esempio per chiarire il procedimento.

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*.²²

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 :

$$S \rightarrow aS \mid bS \mid aaA \quad A \rightarrow aA \mid bA \mid b$$

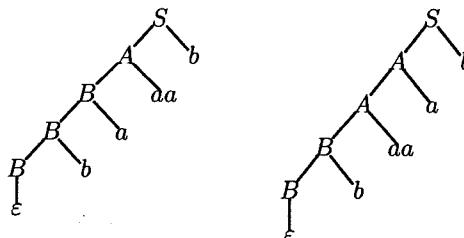
2. Grammatica lineare a sinistra G_s :

$$S \rightarrow Ab \quad A \rightarrow Aa \mid Ab \mid Baa \quad B \rightarrow Ba \mid Bb \mid \epsilon$$

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 $baaab$:



²²Con 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

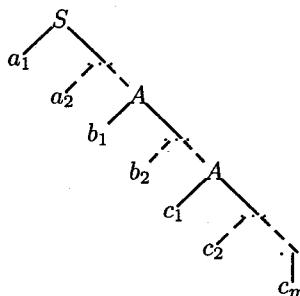
È importante comprendere i limiti della famiglia dei linguaggi regolari, allo scopo di decidere quali costrutti linguistici siano modellabili con le espressioni regolari e quali invece necessitino l'intera potenza delle grammatiche libere. Si vedrà ora un'importante proprietà strutturale della famiglia *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.

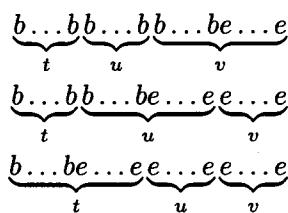
Un costrutto frequente nei linguaggi artificiali è quello che impone che due elementi $begin \equiv b$ e $end \equiv e$ compaiano in questo ordine e abbiano esattamente lo stesso numero di comparse.

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)

$$A \stackrel{+}{\Rightarrow} uAv \quad u \neq \epsilon \wedge v \neq \epsilon$$

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 :

$$S \rightarrow AS \mid bA \quad A \rightarrow aA \mid \epsilon$$

pur non unilineare, non permette derivazioni autoincassate e $L(G)$ è regolare, come si vede risolvendo le equazioni insiemistiche

$$\begin{cases} L_S = L_A L_S \cup b L_A \\ L_A = a L_A \cup \epsilon \end{cases}$$

$$\begin{cases} L_S = L_A L_S \cup b L_A \\ L_A = a^* \end{cases}$$

$$L_S = a^* L_S \cup ba^*$$

$$L_S = (a^*)^* ba^*$$

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 Σ 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.

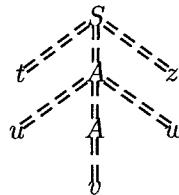
Proprietà 2.81. Linguaggio a tre esponenti.

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:

$$S \stackrel{+}{\Rightarrow} tAz \stackrel{+}{\Rightarrow} tuAwz \stackrel{+}{\Rightarrow} tuvwz$$

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_{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_{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_{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
$R^R \in REG$	$R^* \in REG$	$R_1 \oplus R_2 \in REG$	$\neg R \in REG$	$R_1 \cap R_2 \in REG$
$L^R \in LIB$	$L^* \in LIB$	$L_1 \oplus L_2 \in LIB$	$\neg L \notin LIB$	$L_1 \cap L_2 \notin LIB$ $L \cap R \in LIB$

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)*²³

Siano dati i due alfabeti sorgente Σ e pozzo Δ . 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:

²³Caso 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 Δ . 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*²⁴ 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;

²⁴Extended 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 η 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 α 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 η_1 deriva (immediatamente) η_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 ϑ :

$$e \xrightarrow{*} \vartheta \quad (\text{derivazione per e.r.})$$

Si noti che le stringhe η_1, η_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 η_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.

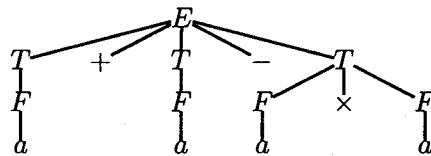
La grammatica estesa G :

$$E \rightarrow [+ | -]T((+ | -)T)^* \quad T \rightarrow F((\times | /)F)^* \quad F \rightarrow (a | '('E')')$$

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 Σ 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.