


Appunti di Linguaggi Formali e Compilatori

Matteo Gianello

19 gennaio 2015

Quest'opera è stata rilasciata con licenza Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo 4.0 Internazionale. Per leggere una copia della licenza visita il sito web <http://creativecommons.org/licenses/by-nc-sa/4.0/>. .

Indice

1	Introduzione alla teoria dei linguaggi formali	3
1.1	Operazioni sulle stringhe	4
1.2	Operazioni sui linguaggi	5
2	Linguaggi ed espressioni regolari	8
2.1	La famiglia dei linguaggi regolari	8
2.2	Sottoespressione di un'espressione regolare	9
2.3	Altre operazioni con le espressioni regolari	10
2.4	Chiusura dei linguaggi regolari rispetto alle operazioni	10
3	Grammatiche libere	12
3.1	Derivazione e generazione del linguaggio	13
3.2	Grammatiche errate e regole inutili	14
3.3	Ricorsione e linguaggi infiniti	15
3.4	Derivazione canonica e albero della sintassi	15
3.5	Ambiguità	17
3.6	Equivalenza forte ed equivalenza debole	19
3.7	Grammatiche in forma normale	19
3.8	Regole di copiatura e loro eliminazione	20
3.9	Forma normale di Chomsky	21
3.10	Forma normale <i>Real Time</i>	21
3.11	Linguaggi regolari e grammatiche libere	22
3.11.1	Dalle espressioni regolari alle grammatiche libere	23
3.11.2	Grammatiche lineari	23

Introduzione

Prima di entrare nel dettaglio della nostra trattazione definiamo che cos'è un *Linguaggio Formale*. Innanzitutto dobbiamo distinguere tra un *linguaggio natura* ed uno *artificiale*, un linguaggio naturale è quello che utilizziamo tutti i giorni e che si basa sul significato delle parole e soprattutto non possiede una struttura formale. Un linguaggio artificiale, invece, è quello utilizzato per la comunicazione tra macchine, è di tipo non verbale ma soprattutto è formale.

Un linguaggio si definisce **formale** se la sua *sintassi* (ovvero la sua struttura) e la sua *semantica* (ovvero il suo significato) sono definiti da precisi algoritmi. Questo permette di definire delle *procedure* precise per determinare la correttezza di una frase che appartiene ad un linguaggio e il suo significato. In un contesto più ristretto un **linguaggio formale** è un *oggetto matematico* costruito su di un *alfabeto* tramite regole assiomatiche chiamate *grammatiche* o tramite strumenti matematici chiamati *automi*.

1 Introduzione alla teoria dei linguaggi formali

Vediamo ora alcune definizioni che ci saranno utili durante il corso:

Alfabeto: insieme *finito* di elementi

$$\Sigma = \{a_1, a_2, a_3, \dots, a_k\}$$

Cardinalità di un alfabeto: è il numero di elementi che lo compongono

$$|\Sigma| = k$$

Stringa: insieme ordinato di elementi dell'alfabeto che possono essere anche ripetuti.

$$\Sigma = \{a, b, c\}$$

$$\text{Stringe} : abc \vee aab \vee ac \vee bbb$$

Linguaggio: insieme finito o infinito di stringhe di un alfabeto

$$L_1 = \{ab, ac, abc\}$$

$$L_2 = \{bc, bbc, \dots\}$$

La struttura insiemistica del linguaggio formale ha due livelli di profondità:

- Insieme non ordinato di elementi non atomici (*stringhe*) che sono
- sequenze ordinate di elementi atomici (*elementi o terminali*).

Cardinalità di un linguaggio: può essere finita o infinita.

$$|L_1| = |\{ab, ac, abc\}| = 3$$

Lunghezza di una stringa: indica il numero dei suoi elementi

$$|abb| = 3 \quad |abbc| = 4$$

Uguaglianza tra stringhe: due stringhe si definiscono uguali se e solo se hanno la stessa lunghezza, ed i suoi elementi coincidono ordinatamente da sinistra a destra.

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

$$x = y \iff h = k \wedge a_i = b_i \quad \forall i = 1 \rightarrow h$$

$x = abccbc$ <i>prefissi</i> : $a, ab, abc, abcc, abccb, abccbc$ <i>suffissi</i> : $c, bc, cbc, ccbc, bccbc, abccbc$ <i>sottostringhe</i> : $....., bc, cc, cb, ...$

Figura 1: Esempio di sottostringhe con prefissi e suffissi

1.1 Operazioni sulle stringhe

La *concatenazione* è l'operazione fondamentale con le stringhe, si tratta di fare il prodotto tra esse. Questa operazione è un'operazione di base e non fa altro che disporre in fila le due stringhe.

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

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

Questo tipo di operazione gode della proprietà associativa e va ad influire sulla lunghezza della stringa come vediamo dalla 1 e dalla 2.

$$(xy)z = x(yz) \tag{1}$$

$$|xyz| = |x| + |y| + |z| \tag{2}$$

La concatenazione ci permette di introdurre un altro concetto relativo alle stringhe, ovvero quello della *stringa vuota* o *nulla* indicata con ε questa stringa è l'elemento neutro rispetto all'operazione di concatenazione, infatti, concatenando ε a destra o a sinistra di una stringa la stringa non cambia.

$$\varepsilon x = x\varepsilon = x$$

Tuttavia bisogna prestare attenzione a non confondere la stringa nulla con l'insieme vuoto \emptyset . Una volta capita l'operazione di concatenazione si possono individuare in una stringa delle *sotto-stringhe*, ovvero stringhe più piccole concatenate tra loro per formare una stringa più grande. Ad esempio data una stringa x essa può essere vista come un insieme di sotto-stringhe concatenate

$$x = uyv$$

Dove:

- y è detta sotto-stringa.
- u è chiamato prefisso.
- v è chiamato suffisso.

Un esempio di suffissi, prefissi e sotto-stringhe è mostrato in Figura 1.

Si definiscono *sotto-stringhe proprie* tutte quelle stringhe che hanno $u \neq \varepsilon$ e $v \neq \varepsilon$.

La *riflessione* è l'operazione che inverte l'ordine dei caratteri che compongono una stringa. Da quanto si vede in Figura2 notiamo che la riflessione gode di alcune proprietà, la riflessione di una stringa riflessa ci riporta alla stringa originale, la riflessione di due stringhe concatenate è uguale

$$\begin{aligned}
x &= a_1 a_2 \dots a_h \\
x^R &= a_h a_{h-1} \dots a_2 a_1 \\
(x^R)^R &= x \\
(xy)^R &= y^R x^R \\
\varepsilon^R &= \varepsilon
\end{aligned}$$

Figura 2: Proprietà della riflessione

$$\begin{aligned}
x &= ab \quad x^0 = \varepsilon \quad x^1 = x = ab \quad x^2 = (ab)^2 = abab \\
y &= a^3 = aaa \quad y^3 = a^3 a^3 a^3 = a^9 \\
\varepsilon^0 &= \varepsilon \quad \varepsilon^2 = \varepsilon
\end{aligned}$$

Figura 3: Esempio di ripetizione

alla riflessione delle singole stringhe e successivamente la loro concatenazione, infine la riflessione della stringa vuota è uguale ancora alla stringa vuota. La *ripetizione* o anche *iterazione* è quell'operazione che permette di concatenare più volte una stringa con se stessa; dato un indice m maggiore di uno allora la ripetizione non fa altro che concatenare m -volte la stringa con se stessa come si vede dalla Figura3. Come nel caso dell'algebra classica anche qui abbiamo una precedenza tra le operazioni, in particolare la *ripetizione* ha precedenza sulla *concatenazione*, anche la *riflessione* ha precedenza sulla *concatenazione*.

1.2 Operazioni sui linguaggi

Prima di tutto dobbiamo definire che cosa vuol dire effettuare un'operazione su di un linguaggio. In realtà effettuare un'operazione su di un linguaggio significa effettuare tale operazione su tutte le stringhe appartenenti a quel linguaggio. Ad esempio effettuare la riflessione di un linguaggio significa:

$$L^R = \{x \mid x = y^R \wedge y \in L\}$$

Ovvero si viene a creare un nuovo linguaggio di stringhe x che non sono altro che la riflessione di tutte le stringhe y del linguaggio di partenza.

Un altro esempio è il linguaggio definito come *prefisso*(L) ovvero:

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

Da questa definizione possiamo definire un linguaggio *prefix-free* nel quale non esiste una stringa appartenente a questo linguaggio che sia prefisso di un'altra stringa dello stesso linguaggio.

$$L_1 = \{x \mid x = a^n b^n \wedge n \geq 1\}$$

Ad esempio:

$$a^2b^2 \in L_1 \quad a^2b \notin L_1$$

Passiamo ora a definire le *operazioni binarie* ovvero quelle composte da due operandi come la *concatenazione* tra due linguaggi.

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

Da questa definizione possiamo anche dimostrare la ripetizione di un linguaggio tramite la 3

$$\begin{cases} L^m = L^{m-1} \cdot L & \text{se } m > 0 \\ L^m = \{\varepsilon\} & \text{se } m = 0 \end{cases} \quad (3)$$

Una cosa molto importante da ricordare è la differenza tra la stringa nulla ε e il linguaggio vuoto \emptyset , infatti abbiamo che:

$$L \cdot \emptyset = \emptyset \cdot L = \emptyset$$

mentre

$$L \cdot \{\varepsilon\} = \{\varepsilon\} \cdot L = L$$

Dalla definizione di potenza di un linguaggio possiamo ricavare la possibilità di definire un secondo linguaggio che contiene tutte le stringhe che hanno lunghezza minore di un determinato numero k che è la potenza del linguaggio. Ad esempio:

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

$$L = \{\varepsilon, a, b, aa, ab, bb, aaa, \dots, bbb\}$$

L'utilizzo della stringa nulla ε ci permette di esprimere le stringhe di lunghezza 1 e 2.

Oltre alle normali operazioni definiamo anche le operazioni insiemistiche sui linguaggi. Tali operazioni sono l'*unione* (\cup), l'*intersezione* (\cap), la *differenza* (\setminus), l'*inclusione* (\subseteq), l'*inclusione stretta* (\subset) e l'*uguaglianza* ($=$).

Dopo aver introdotto queste operazioni possiamo definire il *linguaggio universale* come insieme di tutte le stringhe di alfabeto Σ

$$L_{\text{universale}} = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

$$L_{\text{universale}} = \neg \emptyset$$

Il *complemento* di un linguaggio L di alfabeto Σ è definito come la differenza insiemistica tra il *linguaggio universale* e il linguaggio L ovvero:

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

ed indica l'insieme delle stringhe di alfabeto Σ e che non appartengono a L .

Sia nel linguaggio naturale che in quello artificiale le stringhe possono avere una qualsiasi lunghezza, tuttavia per definire un linguaggio è necessario utilizzare formule di lunghezza finita. Risulta perciò necessario introdurre alcuni operatori per creare delle stringhe infinite. L'operatore *star* noto anche come *stella di Kleene* indica la chiusura *riflessiva* e *transitiva* rispetto al concatenamento. Questo operatore indica l'unione di tutte le potenze del linguaggio e si indica come:

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

Un esempio di un linguaggio finito è $L = \{ab, ba\}$ che diventa infinito una volta applicato l'operatore stella:

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

Tuttavia è anche possibile che il linguaggio iniziale L ed il linguaggio L^* coincidano.

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

Ogni stringa del linguaggio L^* può essere suddivisa in una sotto-stringa del linguaggio L . Se prendiamo un alfabeto Σ come base di un linguaggio, Σ^* contiene tutte le stringhe (Σ^* è il linguaggio universale). Si può anche dire che L è un linguaggio costruito sull'alfabeto Σ tramite la notazione $L \subseteq \Sigma^*$. L'operatore *stella* possiede alcune proprietà importanti:

- monotonicità

$$L \subseteq L^*$$

- chiusura rispetto al concatenamento

$$\text{se } (x \in L^* \wedge y \in L^*) \Rightarrow xy \in L^*$$

- idempotenza

$$(L^*)^* = L^*$$

- commutatività di stella e riflessione

$$(L^R)^* = (L^*)^R$$

Inoltre applicando l'operazione stella al linguaggio vuoto e alla stringa nulla otteniamo che:

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

Un esempio di idempotenza è già stato mostrato ed era:

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

questo perché il linguaggio L può essere visto come:

$$L = L_0^* \text{ dove } L_0 = \{aa\}$$

L'operatore *croce* è la *chiusura transitiva* rispetto al concatenamento, ovvero è l'unione delle potenze tranne la potenza 0; in molti casi è molto utile tuttavia non è indispensabile in quanto deriva dall'operatore stella.

$$L^+ = \bigcup_{h=1 \dots \infty} = L^1 \cup L^2 \cup L^3 \dots$$

Operatore *quoziente* (/) accorcia una frase di un primo linguaggio eliminando un suffisso appartenente ad un secondo linguaggio.

Notare che a differenza dell'operatore differenza la barra in questo caso è rivolta in avanti

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

Un esempio di questo utilizzo è:

$$\begin{aligned} L' &= \{a^{2n}b^{2n} | n > 0\}, \quad L'' = \{b^{2n+1} | n \geq 0\} \\ L' / L'' &= \{a^r b^s | (r \geq, \text{pari}) \wedge (1 \leq s < r, s \text{ dispari})\} \\ &= \{a^2b, a^4b, a^4b^3, \dots\} \end{aligned}$$

regexp	Linguaggio
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
$s \cup t$	$L_s \cup L_t$
$s \cdot t$	$L_s \cdot L_t$
s^*	L_s^*

Tabella 1: Regole di generazione dei linguaggi da regexp

2 Linguaggi ed espressioni regolari

I *linguaggi regolari* sono la famiglia più semplice di linguaggi formali, questo perché possono essere definiti in diversi modi:

- Algebricamente.
- Attraverso grammatiche generative.
- Attraverso algoritmi di riconoscimento (*automi*)

Le *espressioni regolari* (*regexp*) sono stringhe r definite attraverso i caratteri di un alfabeto Σ e mediante l'utilizzo dei meta-simboli $\emptyset, \cup, \cdot, *$ attraverso l'applicazione e la combinazione delle seguenti regole

- $r = \emptyset$
- $r = a, a \in \Sigma$
- $r = (s \cup t)$
- $r = (s \cdot t)$ o $r = (st)$
- $r = (s)^*$

Oltre a questi simboli talvolta, per semplificare la notazione, si utilizzano anche i simboli $\varepsilon = \emptyset^*$ e $r^+ = e \cdot e^*$.

Il risultato di un'espressione regolare e è un linguaggio L_e costruito sull'alfabeto Σ che rispecchia le regole di Tabella 1. A questo punto possiamo dire che un linguaggio è *regolare* se è generato tramite l'utilizzo di *espressioni regolari* come ad esempio

$$e = (111)^* \quad L_e = \{1^{3n} | n = 0, 1, \dots\}$$

2.1 La famiglia dei linguaggi regolari

Definiamo la *famiglia dei linguaggi regolari* (REG) come la collezione di tutti i linguaggi regolari, inoltre, definiamo la *famiglia dei linguaggi finiti* (FIN) la collezione di tutti i linguaggi aventi *cardinalità finita*.

Ogni linguaggio finito è regolare in quanto è l'unione (\cup) di un numero finito di stringhe le quali sono il concatenamento (\cdot) di un numero finito di caratteri. Tuttavia l'insieme dei linguaggi regolari contiene anche linguaggi non finiti perciò l'inclusione risulta essere stretta.

$$FIN \subset REG$$

$(a_1 \cup (b_2 b_3))^*$	$c_4^+ \cup (a_5 \cup (b_6 b_7))$
$a_1 \cup (b_2 b_3)$	$c_4^+ \quad a_5 \cup (b_6 b_7)$
$a_1 \quad b_2 b_3$	$c_4 \quad a_5 \quad b_6 b_7$
$b_2 \quad b_3$	$b_6 \quad b_7$

Figura 4: Scomposizione in sottoespressioni

Scelta	Operazione
$e_k, 1 \leq k \leq n$	$e_1 \cup e_2 \cup \dots \cup e_n$
e	e^*, e^+, e^n
ε	e^*

Tabella 2: Possibile scelte per le sotto espressioni

2.2 Sottoespressione di un'espressione regolare

Consideriamo un espressione completamente parentizzata come la seguente:

$$e = (a \cup (bb))^*(c^+ \cup (a \cup (bb)))$$

A questo punto distinguiamo tutti i termini numerandoli

$$e = (a_1 \cup (b_2 b_3))^*(c_4^+ \cup (a_5 \cup (b_6 b_7)))$$

Infine mettiamo in evidenza tutte le sotto espressioni come mostrato in Figura4. Gli operatori di unione e di ripetizione all'interno di un'espressione regolare corrispondono a possibili alternative o *scelte*, in quanto scegliendo una delle alternative e sostituendola nell'espressione regolare di partenza si ottiene una sottoespressione che definisce un linguaggio più piccolo dell'originale. Le possibili scelte sono elencate in Tabella2. Data un espressione di partenza e_1 è sempre possibile derivare una espressione e_2 sostituendo una sottoespressione con una delle possibili scelte.

In termini più matematici definiamo questo processo come:

$$e' \Rightarrow e''$$

dove e' e e'' sono definite come:

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

dove β è una sottoespressione regolare di e' e δ è sottoespressione regolare di e'' ed inoltre δ è una scelta di β . Questo processo di derivazione può essere applicato in sequenza più volte.

$$\begin{aligned} e_0 &\xrightarrow{n} e_n & e_0 &\Rightarrow e_1 \dots & e_{n-1} &\Rightarrow e_n \\ e_0 &\xrightarrow{*} e_n & e_0 &\text{deriva } e_n & \text{in } n \geq 0 \text{ passi} \\ e_0 &\xrightarrow{+} e_n & e_0 &\text{deriva } e_n & \text{in } n \geq 1 \text{ passi} \end{aligned}$$

Alcune espressioni che si ottengono dalla derivazione di un'espressione regolare contengono dei meta-simboli (operatori e parentesi), altre, invece contengono solamente i simboli dell'alfabeto Σ e la lettera ε questi simboli sono detti *terminali*. Questo tipo di derivazioni costituiscono il linguaggio definito dall'espressione regolare di partenza. Il linguaggio generato da una determinata *regex* di partenza r è definito come:

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

Un linguaggio generato da un'espressione regolare derivata è contenuto nel linguaggio generato dall'espressione regolare di partenza. Due *regexp* si definiscono equivalenti se *generano* lo stesso linguaggio. Possono esistere diverse derivazioni che portano alla stessa stringa, queste derivazioni risultano essere equivalenti.

Alcune volte, è possibile, che due stringhe equivalenti siano ottenute dallo stesso linguaggio non solo cambiando l'ordine di derivazione ma più in generale tramite una differenza strutturale, in questo caso si parla di *ambiguità* dell'espressione regolare.

Una regexp f si definisce *ambigua* se la sua versione numerata f' genera due stringhe x e y tali che queste due stringhe, private della numerazione, coincidono. Ad esempio l'espressione

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

Può generare le due stringhe $a_1 a_3$ e $a_3 a_4$

2.3 Altre operazioni con le espressioni regolari

Oltre a ciò che abbiamo già visto alle espressioni regolari possiamo applicare l'operatore *potenza* $a^h = aaa \dots h - \text{volte}$, l'operatore *ripetizione* che si indica come:

$$[a]_k^n = a^k \cup \dots \cup a^n$$

L'*opzionalità*, indicata come:

$$[a] = (\varepsilon \cup a)$$

e l'*intervallo ordinato* indica un insieme ordinato di simboli come ad esempio $(0 \dots 9)$. Inoltre, nelle espressioni regolari possiamo utilizzare le operazioni insiemistiche (intersezione, differenza e complemento), tuttavia l'utilizzo di quest'ultime non aumenta la potenza espressiva delle espressioni regolari ma serve solo ad abbreviare la notazione. Quando si utilizzano le operazioni insiemistiche si parla di *espressioni regolari estese*. Un esempio di questo tipo di operazioni è l'*intersezione* che esprime la richiesta della stringa di soddisfare due richieste contemporaneamente come ad esempio:

$$r = (a|b)^* bb(a|b)^* \cap ((a|b)^2)^*$$

Che indica una stringa che contiene almeno i caratteri bb e che la lunghezza della stringa sia pari. Questa richiesta risulta essere più complessa nel caso non si utilizzino le operazioni insiemistiche come si vede dalla seguente formula che esprime la stessa richiesta precedente:

$$((a|b)^2)^* bb((a|b)^2)^* | (a|b)((a|b)^2)^* bb(a|b)((a|b)^2)^*$$

2.4 Chiusura dei linguaggi regolari rispetto alle operazioni

Consideriamo un operatore θ che produce un linguaggio (risultato) quando viene applicato a un linguaggio o ad una coppia di linguaggi. Una famiglia di linguaggi si dice chiusa rispetto all'operatore θ se il linguaggio risultante appartiene alla stessa famiglia dei linguaggi di partenza. In particolare la famiglia dei linguaggi regolari è chiusa rispetto agli operatori *unione*, *concatenazione* e *stella*, inoltre per come sono definite le espressioni regolari, la famiglia è chiusa rispetto all'operatore *croce* e *potenza*.

Da questa proprietà ne deriva che qualsiasi linguaggio regolare può essere combinato con un altro linguaggio regolare tramite uno degli operatori precedenti ed il risultato sarà un altro linguaggio regolare. L'*astrazione* di un linguaggio parte da frasi di un linguaggio reale e le trasforma in

forme più semplici chiamate *rappresentazioni astratte*. Per fare questo i simboli dell'alfabeto reale vengono sostituiti da quelli dell'alfabeto astratto.

A livello di astrazione la struttura di molti linguaggi artificiali può essere ottenuta tramite la composizione di pochi elementi e l'utilizzo di operatori base come l'unione la concatenazione e l'iterazione.

3 Grammatiche libere

Fino ad ora abbiamo visto i linguaggi regolari che sono quelli che vengono generati da espressioni regolari. Questo tipo di linguaggi tuttavia risulta essere limitato per rappresentare alcuni linguaggi come ad esempio il seguente:

$$L_1 = \{x0b^n e^n, n \geq 1\}$$

Questo linguaggio non è regolare in quanto non esiste un'espressione regolare che lo generi. Questo tipo di linguaggi è chiamato *libero dal contesto* e viene generato da *grammatiche libere dal contesto* o più semplicemente *grammatiche libere*.

La *sintassi* è lo strumento utilizzato per definire tali linguaggi, essa utilizza delle *regole* le quali, dopo essere state applicate anche ripetutamente, generano tutte e sole le stringhe di tale linguaggio. L'insieme di queste regole è chiamata *grammatica generativa*.

Un esempio di grammatica generativa è quella che genera il linguaggio che contiene l'insieme di stringhe palindromo:

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

Generato dall'insieme seguente di regole:

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

Una possibile catena di derivazione potrebbe essere

$$frase \Rightarrow a \text{ frase } a \Rightarrow ab \text{ frase } ba \Rightarrow abb \text{ frase } bba \Rightarrow abb\varepsilon bba = abbbba$$

Notiamo come il meta-simbolo \rightarrow serva a separare la parte sinistra di una regola della grammatica dalla parte destra.

Un altro esempio sono le espressioni regolari, le quali definiscono linguaggi regolari tramite l'alfabeto *terminale* $\Sigma = \{a, b\}$, sono anch'esse dei linguaggi e in particolare sono definite sull'alfabeto $\Sigma_{e.r.} = \{a, b, \cup, *, \emptyset, (,)\}$. La sintassi che genera la grammatica $G_{r.e.}$ è:

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

Una grammatica libera dal contesto (nella *forma normale di Backus*) è definita da quattro entità:

- V alfabeto di simboli *non terminali*
- Σ alfabeto dei simboli *terminali*
- P insieme di regole *sintattiche* anche chiamato regole di produzione
- $S \in V$ particolare simbolo non terminale chiamato *assioma*.

terminale : la PD contiene terminali o la stringa vuota	$\rightarrow u \mid \varepsilon$
vuota (o nulla) : la PD e' vuota	$\rightarrow \varepsilon$
iniziale : la PS e' l'assioma	$S \rightarrow$
ricorsiva : la PS compare nella PD	$A \rightarrow \alpha A \beta$
ricorsiva a sinistra : la PS e' prefisso della PD	$A \rightarrow A \beta$
ricorsiva a destra : la PS e' suffisso della PD	$A \rightarrow \beta A$
ricorsiva a destra e sinistra : intersezione dei due casi prec.	$A \rightarrow A \beta A$
copiatura o categorizzazione : la PD e' un nt singolo	$A \rightarrow B$
lineare : al piu' un nt nella PD	$\rightarrow u B v \mid w$
lineare a destra (tipo 3) : come lineare, con nt suffisso	$\rightarrow u B \mid w$
lineare a sinistra (tipo 3) : come lineare, con nt prefisso	$\rightarrow B v \mid w$
normale di Chomsky : due nt o un solo terminale	$\rightarrow BC \mid a$
normale di Greibach : un terminale seguito da nonterminali	$\rightarrow a \sigma \mid b$
a operatori : due nt separati dal terminale (operatore)	$\rightarrow A a B$

Figura 5: Tipi di regole

Ogni insieme di regole di P è una coppia ordinata (X, α) , $X \in V$ e $\alpha \in (V \cup \Sigma)$. Per evitare confusioni è utile che i meta-simboli $\rightarrow, |, \cup, \varepsilon$ non siano presenti negli alfabeti V e Σ , inoltre gli alfabeti terminali e non terminali devono essere disgiunti.

Introduciamo ora alcune convenzioni, la prima che vediamo è quella per la rappresentazione dei simboli terminali e non terminali, in realtà esistono diversi modi per fare questa distinzione:

- utilizzo delle parentesi angolari per i non terminali

$$\langle if_phrase \rangle \rightarrow if \langle cond \rangle then \langle phrase \rangle else \langle phrase \rangle$$

- utilizzo del grassetto per i terminali

$$if_phrase \rightarrow \mathbf{if} \ cond \ \mathbf{then} \ phrase \ \mathbf{else} \ phrase$$

- utilizzo dei caratteri maiuscoli e minuscoli

$$F \rightarrow if \ C \ then \ D \ else \ D$$

La convenzione che useremo maggiormente è l'ultima con l'uso di lettere maiuscole per i non terminali e lettere minuscole per i terminali. Inoltre per indicare le stringhe che contengono solamente terminali utilizzeremo l'alfabeto latino minuscolo $\{r, s, \dots, z\}$, per le stringhe che contengono sia simboli terminali che non terminali utilizzeremo l'alfabeto greco minuscolo $\{\alpha, \beta\}$ infine per indicare le stringhe che contengono solo non terminale utilizzeremo solamente il simbolo σ .

Nello schema di Figura5 vediamo quali sono i tipi di regole che si possono presentare, nello schema si utilizza come convenzione PD per indicare la parte destra, PS per indicare la parte sinistra e nt per indicare il non-terminale.

3.1 Derivazione e generazione del linguaggio

Derivare un linguaggio da un insieme di regole significa procedere nel seguente modo:

$$\beta, \gamma \in (V \cup \Sigma)^*$$

allora γ deriva da β nella grammatica G

$$\beta \Rightarrow \gamma \text{ se } A \rightarrow \alpha \text{ è una regola della grammatica } G$$

ed inoltre

$$\beta = hAd, \gamma = h\alpha d$$

Il linguaggio generato da G quando si parte dal non terminale A viene chiamato $L_A(G)$. Una forma generata da G partendo dal non terminale $A \in V$ è una stringa $\alpha \in (V \cup \Sigma)^*$ tale che:

$$A \xRightarrow{*} \alpha$$

Se il non terminale A è l'*assioma* allora la forma generata è detta frasale. Se la forma frasale generata contiene solamente caratteri terminali allora si chiama semplicemente *frase*.

Analizziamo ora l'esempio di una grammatica G_I che genera la struttura di un libro, il quale contiene un frontespizio f una serie di uno o più capitoli A ed ogni capitolo inizia con un titolo t e contiene una o più righe l . La sintassi di questa grammatica è:

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

Il non terminale A genera la forma $tBtB$ ed eventualmente le stringhe $tltl \in L_A(G_I)$. Il non terminale S genera forme frasali del tipo $fAtlB$ e $FtBtB$. Un linguaggio si definisce *libero dal contesto* se e solo se esiste una grammatica libera che lo genera. In questo esempio il linguaggio generato è anche regolare ma la famiglia di linguaggi liberi LIB contiene strettamente la famiglia di quelli regolari.

$$REG \subset LIB$$

Due grammatiche G e G' si dicono *equivalenti* se e solo se generano lo stesso linguaggio, ovvero se $L(G) = L(G')$

3.2 Grammatiche errate e regole inutili

Quando scriviamo una grammatica dobbiamo prestare attenzione che tutti i simboli non terminali siano definiti e tutti questi contribuiscano a generare la stringa del linguaggio. Una grammatica si dice in *forma ridotta* se entrambi le condizioni seguenti sono soddisfatte:

- ogni non-terminale A è raggiungibile dall'assioma, ovvero esiste una derivazione come la seguente:

$$S \xRightarrow{*} \alpha A \beta$$

- Ogni non terminale A genera un insieme di stringhe non vuoto.

$$L_A(G) \neq \emptyset$$

Per fare ciò si può applicare un algoritmo in due fasi. La prima fase identifica i non terminali non identificati, la seconda fase invece identifica quelli irraggiungibili.

Più in dettaglio la fase uno si svolge lungo i seguenti passi:

- Costruzione dell'insieme complemento DEF dei non-terminali definiti:

$$DEF = V \setminus UNDEF$$

- Per costruire tale insieme si inseriscono tutti i non terminali che nelle loro regole presentano nella parte destra solo elementi terminali.

$$DEF := \{A | (A \rightarrow u) \in P \wedge u \in \Sigma^*\}$$

- Infine si inseriscono ricorsivamente nell'insieme tutti i non terminali che nelle loro regole presentano nella parte destra dei simboli terminali oppure dei simboli non terminali già inclusi in DEF

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

Per ogni interazione della prima fase possono accadere due cose, la prima è che viene individuato un nuovo simbolo non terminale da inserire nell'insieme DEF la seconda è che l'insieme DEF non cambia, in questo caso significa che si è raggiunta la convergenza. I simboli non terminali rimasti fuori dall'insieme DEF possono essere eliminati.

La seconda fase identifica i non terminali che sono raggiungibili dall'assioma S , questo problema si può ridurre a trovare l'esistenza di un percorso all'interno del grafo costruito tramite la relazione:

$$A \xrightarrow{\text{produce}} B \iff A \rightarrow \alpha B \beta \text{ con } A \neq B$$

I non-terminali non raggiungibili possono essere eliminati dalla grammatica.

Una terza proprietà può essere applicata per ridurre ulteriormente una grammatica. Questa proprietà definisce che una grammatica G non deve avere derivazioni circolari, le quali non sono essenziali e possono portare ad ambiguità.

3.3 Ricorsione e linguaggi infiniti

La maggior parte dei linguaggi che ci interesseranno sono infiniti, ovvero contengono un numero infinito di stringhe. Ci dobbiamo chiedere quale sia il meccanismo che permetta ad una grammatica libera di generare un'infinità di stringhe. La risposta a questa domanda è la *ricorsione*.

$$A \xRightarrow{n} xAy \quad n \geq 1$$

Condizione necessaria e sufficiente perché un linguaggio $L(G)$ sia infinito è che la grammatica generativa G si ricorsiva.

3.4 Derivazione canonica e albero della sintassi

Un *albero della sintassi* è un grafo aciclico nel quale ogni coppia di nodi è connessa da uno e un solo percorso non necessariamente diretto. L'albero della sintassi rappresenta graficamente il processo di derivazione, esiste una relazione *padre-figlio* o anche *radice-foglia* tra i nodi. L'insieme di tutte le foglie lette da destra a sinistra è detto *frontiera*. Si definisce *grado* di un nodo (o anche *arità*) il numero di figli di quel nodo.

Un *sotto-albero* con radice N è l'albero che ha come radice N ed include tutti i discendenti di tale nodo. Nel caso di albero della sintassi la radice è l'*assioma* della grammatica e la frontiera è una delle stringhe generate. Vediamo ora una grammatica in Figura6(a) e i corrispettivi sotto-alberi che rappresentano le diverse regole Figura6(b).

Da questa grammatica possiamo generare diverse stringhe e ad ogni stringa generata possiamo creare un albero sintattico come quello mostrato in Figura7, quest'albero lo possiamo rappresentare in versione parentizzata come segue:

$$[[[[i]_F]_T]_E + [[[i]_F]_T \times [i]_F]_T]_E$$

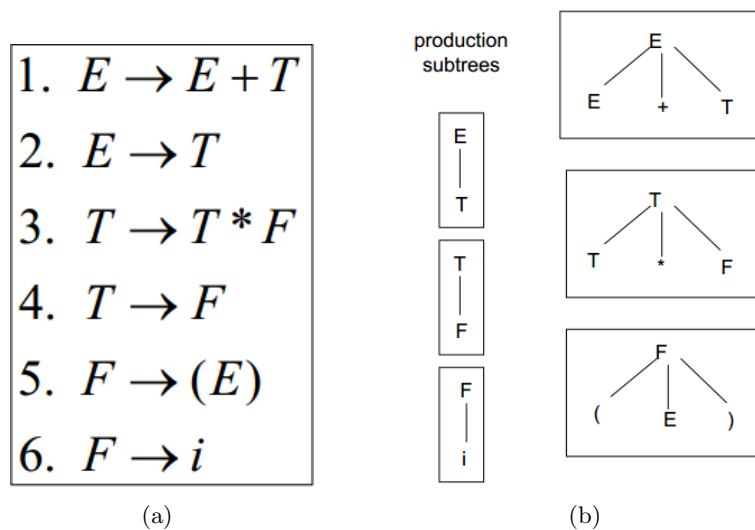


Figura 6: Grammatica 6(a) e corrispettivi sotto alberi 6(b)

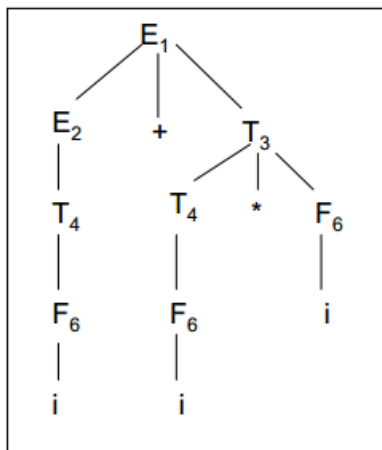


Figura 7: Albero sintattico per la grammatica di Figura6(a)

Molte volte quello che conta è solamente la struttura dell'albero e la sua frontiera e non i nodi intermedi per questo si può prendere in considerazione l'*albero scheletrico* come quello di Figura8. Oppure la sua versione condensata nella quale i percorsi lineari come quelli generati dalle regole 2,4,6 vengono collassati.

Si parla di *derivazione destra (sinistra)* quando in un albero di derivazione si deriva prima il nodo di frontiera più a *destra (sinistra)* ovvero in una frase di non terminali come quella in esempio si applica la derivazione seguente

$$E + T \xRightarrow{d} E + T \times F$$

Ogni frase di una grammatica libera può essere generata tramite una derivazione destra o sinistra. Questa proprietà è di grande importanza per gli algoritmi di analisi sintattica.

Frequentemente i linguaggi artificiali contengono strutture innestate come le parentesi, in questo caso si hanno sempre due elementi uno di apertura e uno di chiusura i quali racchiudono alcune sotto strutture tra cui anche altre coppie di elementi innestati, questo può avvenire in modo

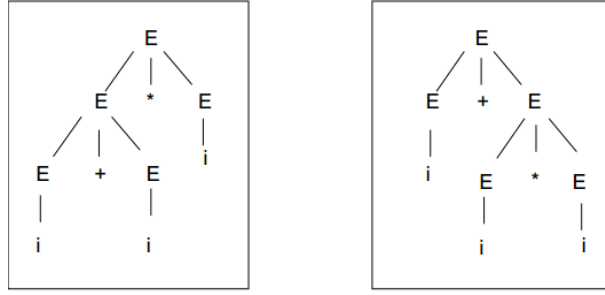


Figura 9: Due alberi sintattici che generano la stessa stringa

L'ambiguità sintattica si ha quando una frase x di un linguaggio generato da una grammatica G ammette più di un albero sintattico e quindi più derivazioni. Una grammatica g si definisce ambigua se ammette anche una sola stringa ambigua. Un esempio è la grammatica G' che risulta essere ambigua:

$$E \rightarrow E + E | E \times E | (E) | i$$

La quale genera la stringa $i + i \times i$ che è ambigua in quanto ammette due alberi sintattici che sono quelli in Figura 9. Si definisce *grado di ambiguità* di una frase x il numero di differenti alberi sintattici che generano la stringa x . Il *grado di ambiguità* di una grammatica è il grado massimo di ambiguità di una stringa x tra tutte quelle generate dalla grammatica. Decidere se una grammatica è ambigua è un problema *indecidibile* quindi non esiste un algoritmo per determinare se una grammatica è ambigua oppure no. Tuttavia per alcune grammatiche è possibile esplorare tutti i grafi e quindi stabilire se una determinata grammatica è ambigua; è quindi auspicabile che la non ambiguità di una grammatica sia determinata in fase di costruzione. Se un simbolo non terminale è ricorsivo sia a destra che a sinistra solitamente permette due o più modi di derivazione e questo porta sempre ad un comportamento ambiguo.

Se due linguaggi $L_1(G_1)$ e $L_2(G_2)$ condividono alcune frasi, ovvero la loro intersezione non è vuota la grammatica che risulta come unione delle due è sicuramente ambigua in quanto esiste un albero in G_1 e uno in G_2 .

Esistono poi dei linguaggi che sono *internamente* ambigui e di conseguenza tutte le grammatiche che generano questi linguaggi sono ambigue. Consideriamo ad esempio il linguaggio

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

In quanto possiamo suddividerlo in due linguaggi ed effettuarne l'unione.

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

In questo caso tutte le stringhe del tipo $\varepsilon, abc, a^2 b^2 c^2, \dots$ sono generabili in entrambi i linguaggi. Questo comportamento è dovuto alla natura del linguaggio e in qualsiasi modo si modifichino le grammatiche che generano il comportamento delle frasi, l'ambiguità è inevitabile.

La concatenazione di due linguaggi può portare ambiguità se esiste un suffisso di una frase del primo linguaggio che è anche prefisso di una frase del secondo linguaggio.

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_{N_1} \cup V_{N_2}, \{S \rightarrow S_1 S_2\} \cup P_1 \cup P_2, S)$$

Assumiamo che le grammatiche G_1 e G_2 non siano ambigue, allora G è ambigua se esistono due frasi $x' \in L_1$ e $x'' \in L_2$ ed una stringa non vuota v tale che:

$$x' = u'v \vee u' \in L_1 \quad x'' = vz'' \vee z'' \in L_2$$

Allora la stringa:

$$u'vz'' \in L_1 \cdot L_2$$

è ambigua infatti le due derivazioni sono:

$$\begin{array}{lcl} S & \Rightarrow & S_1 S_2 \xRightarrow{+} u' S_2 \xRightarrow{+} u' v z'' \\ S & \Rightarrow & S_1 S_2 \xRightarrow{+} u' v S_2 \xRightarrow{+} u' v z'' \end{array}$$

3.6 Equivalenza forte ed equivalenza debole

Due grammatiche G e G' si definiscono debolmente equivalenti se e solo se generano lo stesso linguaggio ovvero se $L(G) = L(G')$. Queste due grammatiche possono avere strutture (alberi sintattici) completamente diversi per generare la stessa frase, può capitare perciò che alcune strutture siano inadeguate per alcuni usi.

L'equivalenza *forte* o *strutturale* si ha quando due grammatiche G e G' generano lo stesso linguaggio e per ogni frase hanno alberi sintattici equivalenti ovvero l'albero scheletrico condensato è identico per entrambe le grammatiche. L'equivalenza forte implica quella debole ma non viceversa, inoltre l'equivalenza forte è una proprietà decidibile, mentre quella debole non lo è.

Quando si definisce formalmente un linguaggio non si può fare a meno di tenere in considerazione un'*adeguatezza strutturale*, in quanto la grammatica generata dal linguaggio è la base sintattica per definire l'interpretazione semantica delle frasi.

Esiste poi quella che viene definita *equivalenza strutturale in senso largo*. Consideriamo due grammatiche come le seguenti:

$$\{S \rightarrow Sa|a\} \quad \{X \rightarrow aX|a\}$$

Queste due grammatiche non sono equivalenti strutturalmente in quanto per generare la stringa aa creano due alberi sintattici diversi, tuttavia l'unica differenza che intercorre tra le due è che l'albero sintattico della seconda è rispecchiato rispetto alla prima. Si dice quindi che le due grammatiche sono equivalenti in senso *largo*.

3.7 Grammatiche in forma normale

Una grammatica in forma normale comporta alcune restrizioni a livello di regole, tuttavia non ne limita la potenza espressiva. Questo tipo di forma è utile per dimostrare i teoremi piuttosto che per la definizione di linguaggi.

Esistono alcune trasformazioni per trasformare una grammatica dalla sua forma generale alla sua forma normale. Queste trasformazioni sono utili per costruire degli analizzatori sintattici.

Espansione di un non terminale Prendiamo in considerazione la grammatica $G = \{\Sigma, V, P, S\}$ composta dalle seguenti due regole

$$A \rightarrow \alpha B \gamma \quad B \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$$

L'espansione di un simbolo non terminale significa sostituire un non terminale in una regola in modo da eliminarlo come in questo caso possiamo fare con il non terminale B .

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

Rimozione dell'assioma dalla parte destra È sempre possibile limitare la parte destra di una regola al solo alfabeto $(\Sigma \cup (V - \{S\}))$ ovvero è sempre possibile eliminare l'assioma dalle parti destre delle regole, è sufficiente introdurre un nuovo assioma S_0 e la corrispettiva regola $S_0 \rightarrow S$

Non-terminali annullabili Un non-terminale si dice *annullabile* se e solo se esiste una derivazione come la seguente

$$A \xRightarrow{+} \varepsilon$$

Questo tuttavia non significa che il non terminale in questione possa generare altre stringhe diverse da ε . Consideriamo $Null \subseteq V$ l'insieme dei non terminali annullabili allora

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

Un esempio è la grammatica:

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

L'insieme dei non-terminali annullabili è

$$Null = \{A, B\}$$

Una grammatica si dice in forma *non annullabile* se e solo se contiene tutti non-terminali non annullabili e l'assioma è annullabile se e solo se il linguaggio contiene la stringa vuota $S \rightarrow \varepsilon$. Per ottenere una grammatica in forma normale non annullabile dobbiamo procedere lungo i seguenti passi:

1. Calcolare l'insieme $Null$
2. Per ogni regola di P aggiungere una regola ottenuta da quelle di P eliminando ogni non-terminale annullabile che si presenta nella parte destra delle regole in ogni sua forma.
3. Eliminare le regole vuote del tipo $A \rightarrow \varepsilon$ a meno che non sia $A = S$
4. Rimuovere $S \rightarrow \varepsilon$ se e solo se il linguaggio non contiene la stringa vuota.
5. Portare la grammatica in forma ridotta rimuovendo le derivazioni circolari.

3.8 Regole di copiatura e loro eliminazione

Prendiamo ad esempio la regola $A \rightarrow B$, in questa regola la classe sintattica B è inclusa in quella di A , rimuovendola si ottiene una grammatica equivalente ma che ha un albero sintattico meno profondo. Consideriamo ora $Copy(A) \subseteq V$ come l'insieme dei non-terminali in cui il terminale A viene copiato. Le regole di copiatura non sono totalmente inutili anzi talvolta ci permettono di condividere alcune parti della grammatica, per questo le regole di copiatura vengono utilizzate nelle grammatiche dei linguaggi tecnici come ad esempio le istanze nei linguaggi di programmazione.

Per eliminare le regole di copiatura per prima cosa dobbiamo calcolare l'insieme $Copy$; per calcolare tale insieme si seguono le seguenti clausole logiche.

$$A \in Copy(A)$$

$$C \in Copy(A) \text{ se } (B \in Copy(A)) \wedge (B \rightarrow C \in P)$$

A questo punto si costruisce la grammatica G' equivalente a G ma senza le regole di copiatura.

$$\begin{array}{l}
S \rightarrow dA \mid cB \quad A \rightarrow dAA \mid cS \mid c \quad B \rightarrow cBB \mid dS \mid d \\
S \rightarrow \langle d \rangle A \mid \langle c \rangle B \\
A \rightarrow \langle d \rangle \langle AA \rangle \mid \langle c \rangle S \mid c \\
B \rightarrow \langle c \rangle \langle BB \rangle \mid \langle d \rangle S \mid d \\
\langle d \rangle \rightarrow d \quad \langle c \rangle \rightarrow c \\
\langle AA \rangle \rightarrow AA \quad \langle BB \rangle \rightarrow BB
\end{array}$$

Figura 10: Esempio di trasformazione in forma normale di chomsky

$$\begin{array}{l}
A \rightarrow A\beta_1 \mid A\beta_2 \mid \dots \mid A\beta_h \\
\text{where no } \beta_i \text{ is empty} \\
A \rightarrow \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_k \\
\\
A \rightarrow \gamma_1 A' \mid \gamma_2 A' \mid \dots \mid \gamma_k A' \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_k \\
A' \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_h A' \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_h \\
\\
A \Rightarrow A\beta_2 \Rightarrow A\beta_3\beta_2 \Rightarrow \gamma_1\beta_3\beta_2 \\
A \Rightarrow \gamma_1 A' \Rightarrow \gamma_1\beta_3 A' \Rightarrow \gamma_1\beta_3\beta_2
\end{array}$$

Figura 11: Esempio di trasformazione di una ricorsione sinistra

3.9 Forma normale di Chomsky

Nella forma normale di Chomsky abbiamo solo due tipi di regole:

- Regole binarie:

$$A \rightarrow BC \text{ dove } B, C \in V$$

- Regole terminali, con una sola lettera sul lato destro:

$$A \rightarrow a \text{ dove } a \in \Sigma$$

Se il linguaggio contiene la stringa vuota è necessario aggiungere la regola:

$$S \rightarrow \varepsilon$$

Un esempio di trasformazione in forma normale di Chomsky è mostrato in Figura10 dove i simboli tra parentesi angolari ($\langle \rangle$) indicano dei simboli non-terminali introdotti appositamente.

3.10 Forma normale *Real Time*

Prima di introdurre la forma normale real time dobbiamo introdurre la *trasformazione della ricorsione sinistra* in una ricorsione destra. Prima di tutto si parte con il trasformare una ricorsione sinistra in una forma non ricorsiva a sinistra come nell'esempio mostrato in Figura11 questa trasformazione è indispensabile per creare analizzatori sintattici discendenti. In questo

$ \begin{array}{l} A_1 \rightarrow A_2 a \quad A_2 \rightarrow A_2 ac \mid bA_1 \mid d \\ A_1 \rightarrow A_2 a \quad A_2 \rightarrow bA_1 A'_2 \mid dA'_2 \mid d \mid bA_1 A'_2 \rightarrow acA'_2 \mid ac \end{array} $	$ \begin{array}{l} A_1 \rightarrow bA_1 A'_2 a \mid dA'_2 a \mid da \mid bA_1 a \\ A_2 \rightarrow bA_1 A'_2 \mid dA'_2 \mid d \mid bA_1 \\ A'_2 \rightarrow acA'_2 \mid ac \end{array} $
(a)	(b)
$ \begin{array}{l} A_1 \rightarrow A_2 a \quad A_2 \rightarrow A_2 ac \mid bA_1 \mid d \\ A_1 \rightarrow A_2 a \quad A_2 \rightarrow bA_1 A'_2 \mid dA'_2 \mid d \mid bA_1 A'_2 \rightarrow acA'_2 \mid ac \end{array} $	
(c)	

Figura 12: Esempio di trasformazione in forma normale

esempiio la ricorsione è diretta, tuttavia nel caso la ricorsione non sia diretta si lavora con un algoritmo in due fasi che però necessita di alcune ipotesi, ovvero, la grammatica G deve essere non-omogenea, non-annullabile e le regole devono presentare al massimo un non terminale. L'algoritmo applicato lavora iterativamente su due passi, il primo consiste nell'espansione delle regole per esporre la ricorsione sinistra, il secondo passaggio trasforma la ricorsione sinistra in una ricorsione destra. La forma *normale real-time* è una forma nella quale le regole della grammatica iniziano sempre con un simbolo terminale (niente ricorsione sinistra).

$$A \rightarrow a\alpha \text{ dove } a \in \Sigma, \alpha \in \{\Sigma \cup V\}^*$$

Esiste poi una particolare forma real time che è quella di *Greibach* nella quale il simbolo terminale è seguito da uno o più non-terminali.

$$A \rightarrow a\alpha \text{ dove } a \in \Sigma, \alpha \in V^*$$

Il termine *real time* viene utilizzato perché l'analizzatore sintattico costruito con questo tipo di grammatiche analizza e consuma un solo carattere terminale ad ogni step e quindi l'algoritmo impiega tanti step quanto è la lunghezza della stringa.

Per trasformare una grammatica in forma normale real time bisogno che essa sia non-annullabile e si procede nei seguenti passi:

- Si elimina la ricorsione diretta, sia immediata che non Figura12(a)
- Mediante trasformazioni elementari, si espandono i non-terminali più a sinistra della parte destra della regola Figura12(b)
- Si introducono nuovi non-terminali per eliminare eventuali simboli terminali che si presentano all'estremità destra della parte destra della regola Figura12(c).

3.11 Linguaggi regolari e grammatiche libere

I linguaggi regolari sono casi particolari di quelli liberi, essi infatti possono essere generati da grammatiche che presentano delle regole fortemente ristrette. Man mano che la lunghezza di una stringa generata da un linguaggio regolare cresce essa presenta delle sottostringhe ripetute.

È possibile dimostrare in modo rigoroso che alcuni linguaggi liberi non possono essere generate da espressioni regolari. Nei linguaggi regolari per identificare una stringa è necessaria solo una quantità finita di memoria, per i linguaggi liberi invece la quantità di memoria deve essere infinita.

1. $r = r_1.r_2....r_k$	1. $E = E_1E_2...E_k$
2. $r = r_1 \cup r_2 \cup \cup r_k$	2. $E = E_1 \cup E_2 \cup ... \cup E_k$
3. $r = (r_1)^*$	3. $E = EE_1 \varepsilon$ or $E = E_1E \varepsilon$
4. $r = (r_1)^+$	4. $E = EE_1 E_1$ or $E = E_1E E_1$
5. $r = b \in \Sigma$	5. $E = b$
6. $r = \varepsilon$	6. $E = \varepsilon$

Figura 13: Regole di trasformazione da *regex* a grammatica libera

3.11.1 Dalle espressioni regolari alle grammatiche libere

Gli operatori iterativi regolari come *stella* e *croce* posso essere sostituiti da regole ricorsive. Per fare ciò si dividono le espressioni regolari in sotto-espressioni e si enumerano progressivamente, dopo aver fatto questo si trasformano le *regex* utilizzando le trasformazioni di Figura13. Nel caso la in cui l'espressione regolare risulti ambigua anche l'equivalente grammatica sarà ambigua. Ogni linguaggio regolare è libero ma non tutti i linguaggi liberi sono regolari.

$$REG \subset LIB$$

3.11.2 Grammatiche lineari

Una grammatica si definisce lineare se ogni regola ha al più un non-terminale nella parte destra della regola.

$$A \rightarrow uBv \text{ dove } u, v \in \Sigma^*, B \in (V \cup \varepsilon)$$

Elenco delle figure

1	Esempio di sottostringhe con prefissi e suffissi	4
2	Proprietà della riflessione	5
3	Esempio di ripetizione	5
4	Scomposizione in sottoespressioni	9
5	Tipi di regole	13
6	Grammatica 6(a) e corrispettivi sotto alberi 6(b)	16
7	Albero sintattico per la grammatica di Figura6(a)	16
8	Albero scheletrico	17
9	Due alberi sintattici che generano la stessa stringa	18
10	Esempio di trasformazione in forma normale di chomsky	21
11	Esempio di trasformazione di una ricorsione sinistra	21
12	Esempio di trasformazione in forma normale	22
13	Regole di trasformazione da <i>regex</i> a grammatica libera	23