

# Teoria dei Linguaggi

# Indice

<b>1. Lezione 18 [09/05]</b>	<b>4</b>
1.1. Ancora pumping lemma	4
1.2. Fail del pumping lemma	4
1.3. Lemma di Ogden	6
1.4. Applicazioni del lemma di Ogden	9
1.5. Ambiguità	10
<b>2. Lezione 19 [14/05]</b>	<b>13</b>
2.1. Ambiguità	13
2.2. Ambiguità e non determinismo	16
<b>3. Lezione 20 [16/05]</b>	<b>20</b>
3.1. Operazioni insiemistiche	20
3.1.1. Unione	20
3.1.1.1. CFL	20
3.1.1.2. DCFL	20
3.1.2. Intersezione	21
3.1.2.1. CFL	21
3.1.2.2. DCFL	21
3.1.3. Intersezione con un regolare	21
3.1.3.1. CFL	21
3.1.3.2. DCFL	21
3.1.4. Complemento	21
3.1.4.1. CFL	21
3.1.4.2. DCFL	22
3.1.5. Riassunto	25
3.2. Operazioni regolari	25
3.2.1. Prodotto	25
3.2.1.1. CFL	25
3.2.1.2. DCFL	25
3.2.2. Star	26
3.2.2.1. CFL	26
3.2.2.2. DCFL	26
3.2.3. Riassunto	26
<b>4. Lezione 21 [21/05]</b>	<b>28</b>
4.1. Riassunto chiusura operazioni	28
4.2. CFL vs DCFL	28
4.3. Ricorsione	32
4.4. Linguaggio di Dyck	33
<b>5. Lezione 22 [23/05]</b>	<b>36</b>
5.1. Alfabeti unari	36
5.1.1. Linguaggi regolari	36
5.1.2. Equivalenza tra linguaggi regolari e CFL	37
5.1.3. Teorema di Parikh	38
5.2. Automi a pila two-way	39
5.2.1. Definizione	39
5.2.2. Esempi	40

5.3. Problemi di decisione dei CFL .....	42
5.3.1. Appartenenza .....	42
5.3.2. Linguaggio vuoto e infinito .....	42
5.3.3. Universalità .....	42

# 1. Lezione 18 [09/05]

## 1.1. Ancora pumping lemma

La scorsa lezione abbiamo visto il pumping lemma per i CFL. Rivediamo un secondo la dimostrazione del punto  $vx \neq \varepsilon$  perché non era molto chiara.

Quando risaliamo l'albero di derivazione supponiamo di incontrare la variabile che viene ripetuta dopo. Chiamiamo questa variabile  $A$ . Visto che siamo in un nodo interno, e siamo nella FN di Chomsky, questa variabile arriva una biforcazione di una variabile del livello superiore. Sia  $P$  questa variabile, che genera anche la variabile  $B$  allo stesso livello di  $A$ .

Qua abbiamo due casi:

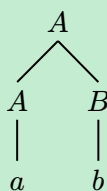
- se  $P = A$  allora abbiamo subito la ripetizione e potremmo avere  $v$  o  $x$  uguali ad  $\varepsilon$  ma non tutti e due, perché da  $B$  tiriamo fuori almeno un terminale, non avendo  $\varepsilon$ -produzioni;
- se  $P \neq A$  allora ancora meglio di prima perché tutti e due potrebbero non essere nulli, visto che ci biforchiamo ancora in su.

Vediamo un esempio per capire meglio.

**Esempio 1.1.1:** Abbiamo una grammatica in FN di Chomsky con le regole di produzione

$$\begin{aligned} A &\rightarrow a \mid AB \\ B &\rightarrow b. \end{aligned}$$

Ci viene dato l'albero di derivazione della stringa  $z = ab$  in questa grammatica.



La  $A$  più in basso viene ripetuta al livello superiore, quindi essa genera il fattore  $w$  della nostra scomposizione. Questo implica che la parte prima, definita dal fattore  $uv$ , è vuota.

La  $A$  più in alto invece genera il fattore  $vwx$ , ma visto che  $w = a$  e che  $v = \varepsilon$ , allora sicuramente  $x = b$ , che come vediamo non è vuoto.

Gli altri due fattori esterni sono invece vuoti, ma su loro non abbiamo condizioni.

## 1.2. Fail del pumping lemma

Il pumping lemma viene usato classicamente per dimostrare che un linguaggio non è CFL. Purtroppo per noi, questo lemma però ogni tanto fallisce nelle dimostrazioni.

**Esempio 1.2.1:** Definiamo il linguaggio

$$L = \{a^n b^n c^k \mid k \neq n\}.$$

Questo linguaggio non è CFL, perché possiamo controllare le  $a$  con le  $b$  ma non possiamo controllare poi  $n$  con  $k$  perché abbiamo perso informazioni.

Per assurdo sia  $L$  un CFL e sia quindi  $N$  la costante del pumping lemma per  $L$ . Mostriamo che qualsiasi stringa lunga almeno  $N$  non riesce a rompere almeno una delle tre condizioni del pumping lemma. Prendiamo quindi la stringa

$$z = a^n b^n c^k \mid k \neq n \wedge 2n + k = |z| \geq N.$$

Decomponiamo  $z$  nella stringa  $z = uvwxy$ . A noi interessano le tre parti centrali, perché qua dentro possiamo pompare (o almeno, le due parti esterne, quella centrale no).

Abbiamo diversi casi da controllare:

$$\begin{array}{lcl} vwx \in a^+ & \mid & vwx \in b^+ \\ vwx \in b^+ c^+ & \mid & vwx \in a^+ b^n c^+ \\ vwx \in c^+ & \mid & vwx \in a^+ b^+. \end{array}$$

I casi della prima riga sono molto facili: pompando con  $i \neq 1$  rompiamo l'uguaglianza tra  $a$  e  $b$ .

I casi della seconda riga sono facili: controllando dove cadono i vari limiti della stringa rompiamo l'uguaglianza tra  $a$  e  $b$  oppure la struttura.

I casi dell'ultima riga sono invece **molto molto difficili**. Vediamoli entrambi.

$[vwx \in c^+]$

Consideriamo la stringa  $uv^i wx^i y$ : in questa stringa, al variare della  $i$ , l'unico valore che cambia rispetto alla stringa  $z$  è il numero di  $c$ . Per rompere questa condizione dobbiamo rendere il numero di  $c$  uguale al numero di  $a$  e  $b$ , ma questo non è sempre possibile.

Infatti, questo dipende dalla  $z$  che abbiamo a disposizione:

- se  $z = a^n b^n c$  basta scegliere  $i = n - 1$  per rompere la condizione di non uguaglianza;
- se  $z = a^{n+n!} b^{n+n!} c^n$ , sapendo che  $vx = c^j$ , possiamo dire che

$$uv^i wx^i y = a^{n+n!} b^{n+n!} c^{n+(i-1)j}$$

ma allora scegliendo

$$(i-1)j = n! \implies i = \frac{n!}{j} + 1$$

noi possiamo rendere il fattore  $(i-1)j$  un fattoriale in  $n$ , e rompere di fatto la condizione sulla non uguaglianza.

Come vediamo, ci sono casi favorevoli, ovvero quando  $k < n$ , ma non tutti sono così.

$[vwx \in a^+ b^+]$

In questo caso, se  $v$  e  $x$  hanno il limite tra le due lettere andiamo a perdere la struttura.

Se invece il limite è in  $w$ , ovvero se  $v = a^l$  e  $x = b^r$ , allora abbiamo due casi:

- se  $l \neq r$  questo è facile, con  $i = 0$  abbiamo ottenuto  $\#_a \neq \#_b$ ;
- se  $l = r$  con la ripetizione arbitraria noi aggiungiamo lo stesso numero di  $a$  e di  $b$ , ovvero otteniamo la stringa

$$uv^iwx^iy = a^{n+(i-1)l}b^{n+(i-1)r}c^k$$

che, per essere resa non in  $L$ , deve avere lo stesso numero di  $a$ ,  $b$  e  $c$ . Per fare questo è comodo quanto le  $c$  sono tante, che va contro il caso precedente, dove volevamo invece poche  $c$ , e non sempre è possibile aggiungere  $a$  e  $b$  per raggiungere lo stesso numero di  $c$ .

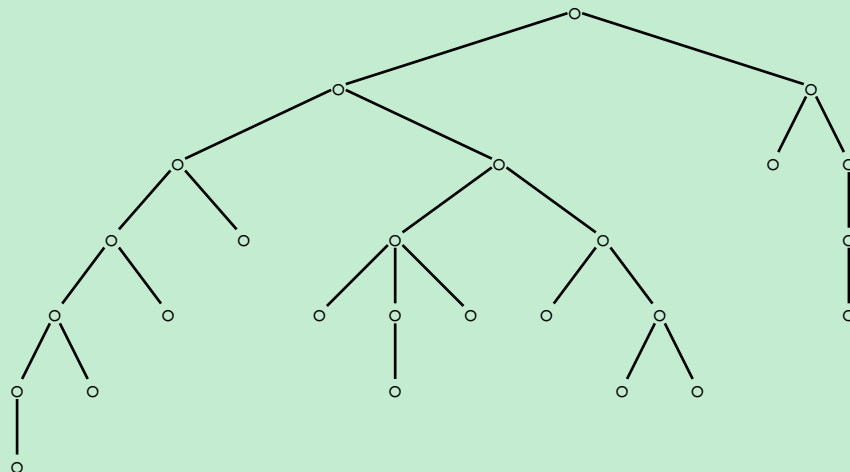
Quindi anche in questo caso ci sono casi favorevoli ma non tutti lo sono.

Il pumping lemma **non funziona**, o meglio, in questo caso non funziona, anche se  $L$  non è CFL. Per risolvere questo problema, dobbiamo dare condizioni più forti del pumping lemma.

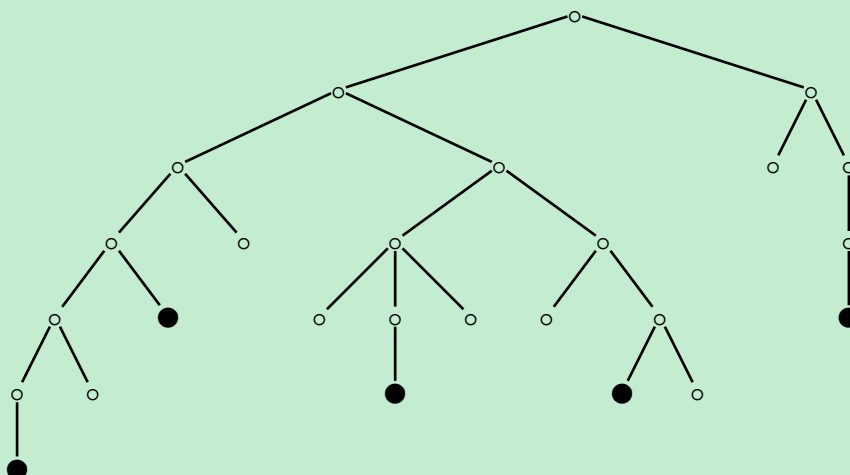
### 1.3. Lemma di Ogden

Partiamo con un paio di esempi utili per fissare alcuni concetti.

**Esempio 1.3.1:** Ci viene dato un albero di derivazione di una grammatica generica.



Andiamo a **marcare**, con un pallino nero grosso, alcune foglie dell'albero.

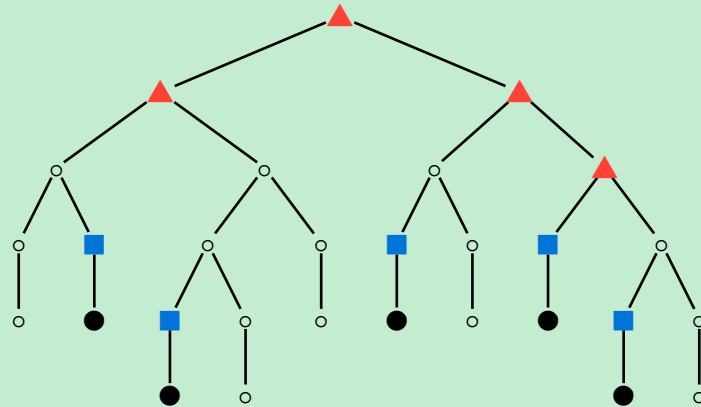


Una volta marcate alcune foglie dell'albero, individuiamo i **branch point**: essi sono nodi interni con almeno due figli, o loro discendenti, che sono nodi marcati. Indichiamo questi nodi con un triangolo rosso pieno.

Identifichiamo infine con un quadrato blu pieno tutti i **padri** dei nodi marcati. Se un nodo è già stato segnato come branch point viene lasciato così.

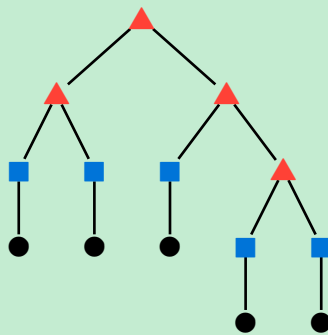
Chiamiamo **nodi speciali** tutti i branch point e i padri dei nodi marcati. Costruiamo ora un albero, detto **albero semplificato**, in cui teniamo solo le foglie marcate e i nodi speciali.

**Esempio 1.3.2:** Vediamo ora un albero in FN di Chomsky, già con nodi marcati e speciali.



Si può dimostrare che, in un albero binario, i branch point sono uno in meno dei nodi marcati.

Ora vediamo l'albero semplificato.



Come vediamo, l'albero semplificato **mantiene** una FN di Chomsky.

Vediamo ora un lemma molto simile ad uno che abbiamo già visto prima del pumping lemma.

**Lemma 1.3.1:** Sia  $G = (V, \Sigma, P, S)$  una grammatica in FN di Chomsky. Sia

$$T : A \stackrel{*}{\Rightarrow} w \mid w \in \Sigma^*$$

un albero di derivazione in  $G$ . Supponiamo di marcare  $d$  posizioni in  $w$ . Se il numero massimo di nodi speciali in un cammino dalla radice alle foglie in  $T$  è  $k$  allora  $w$  contiene al più  $2^{k-1}$  posizioni marcate, ovvero

$$d \leq 2^{k-1}.$$

Questo lemma dà una nuova idea di **misura**: non misuriamo più tutta la stringa, ma solo le posizioni marcate, e consideriamo l'albero semplificato al posto di quello normale.



**Dimostrazione 1.3.1.1:** Si dimostra per induzione, come il lemma della lezione scorsa. ■

Introduciamo finalmente il **lemma di Ogden**.

**Lemma 1.3.2** (Lemma di Ogden): Sia  $L \subseteq \Sigma^*$  un linguaggio CFL. Allora  $\exists N > 0$  tale che  $\forall z \in L$  in cui vengono marcate almeno  $N$  posizioni può essere decomposta come  $z = uvwxy$  tale che:

1.  $vwx$  contiene al più  $N$  posizioni marcate;
2.  $vx$  contiene almeno una posizione marcata;
3.  $\forall i \geq 0 \quad uv^iwx^iy \in L$ .

Notiamo che marcando tutte le posizioni troviamo esattamente il **pumping lemma**.

**Dimostrazione 1.3.2.1:** La dimostrazione di questo teorema è analoga a quella del pumping lemma, ma ragiona sull'albero semplificato associato a quello di derivazione di  $z$ . ■

## 1.4. Applicazioni del lemma di Ogden

Applichiamo quindi il lemma di Ogden per risolvere il problema che abbiamo avuto con il pumping lemma nel primo esempio della lezione.

**Esempio 1.4.1:** Sia di nuovo

$$L = \{a^n b^n c^k \mid k \neq n\}.$$

La difficoltà di questo linguaggio risiede nel fatto di rendere  $=$  un  $\neq$ .

Per assurdo sia  $L$  un CFL e sia  $N$  la costante del lemma di Ogden. Sia  $z$  una stringa molto lunga, molto molto lunga, ad esempio

$$z = a^N b^N c^{N+N!}.$$

Marchiamo, dentro questa stringa, almeno  $N$  posizioni: scegliamo di marcare tutte le  $a$ . Facendo così abbiamo la garanzia che nelle stringhe da pompare abbiamo almeno una  $a$ , e questo sarà comodo per rompere l'uguaglianza con le  $b$  o la struttura.

Decomponiamo quindi  $z$  come  $z = uvwxy$ . Sappiamo che  $vx$  ha almeno una posizione marcata, quindi in  $vx$  abbiamo almeno una  $a$ . Questo restringe il campo di possibili configurazioni da 6 a 3:

$$vw x \in a^+ \quad | \quad v w x \in a^+ b^+ \quad | \quad v w x \in a^+ b^N c^+.$$

Il primo caso è banale, lo risolvevamo anche prima con  $i = 0$  per avere  $\#_a \neq \#_b$ .

Il secondo caso invece era quello ostico, ma ora non più. Dobbiamo capire dove si trova il confine tra le  $a$  e le  $b$ , quindi:

- se  $v \in a^+b^+ \vee x \in a^+b^+$  scegliamo  $i = 2$  per rompere la struttura;
- se  $v \in a^l \wedge x \in b^r$  anche qui abbiamo due casi:
  - se  $l \neq r$  scegliamo  $i = 0$  per avere  $\#_a \neq \#_b$ ;
  - se  $l = r$  qua avevamo dei problemi, mentre ora possiamo farlo, perché se prendiamo la stringa pompata

$$uv^iwx^iy = a^{N+(i-1)l}b^{N+(i-1)r}c^{N+N!} \stackrel{l=r}{=} a^{N+(i-1)l}b^{N+(i-1)l}c^{N+N!}$$

sapendo che  $1 \leq l \leq N$  dobbiamo imporre

$$(i-1)l = N! \implies i = \frac{N!}{l} + 1.$$

Il terzo e ultimo caso l'avevamo già visto prima, dove perdiamo la struttura se  $v$  o  $x$  hanno almeno due tipi di lettere, mentre rompiamo l'uguaglianza quando  $v$  è formato da sole  $a$ . Ma questo è assurdo, quindi  $L$  non è CFL.

**Esempio 1.4.2:** Definiamo ora

$$L = \{a^p b^q c^r \mid p = q \vee q = r \text{ ma non entrambi}\}.$$

Possiamo vedere questo linguaggio come

$$L = L_{\text{prima}} / \{a^n b^n c^n \mid n \geq 0\}.$$

Questo linguaggio non è CFL: la scommessa che facciamo all'inizio verifica che almeno una delle due scelte vada bene, ma non esattamente una delle due.

Anche questo si dimostra con il lemma di Ogden, ma devo farlo io, e non ho voglia ora.

## 1.5. Ambiguità

Vediamo un esempio che ci serve per introdurre il concetto di **ambiguità**.

**Esempio 1.5.1:** Definiamo il linguaggio

$$L = \{a^p b^q c^r \mid p = q \vee q = r\}.$$

Questo linguaggio è un **CFL**: infatti, possiamo fare una scommessa iniziale per verificare almeno una delle due condizioni del linguaggio.

Nel linguaggio appena visto però potrebbero essere vincenti entrambi i rami: in questo caso, noi abbiamo due modi diversi di riconoscere la stringa che ci viene data.

**Esempio 1.5.2:** Diamo una grammatica per il linguaggio precedente. Le produzioni sono

$$S \rightarrow S_1 C \mid A S_2$$

$$S_1 \rightarrow a S_1 b \mid \varepsilon$$

$$S_2 \rightarrow b S_2 c \mid \varepsilon$$

$$A \rightarrow a A \mid \varepsilon$$

$$C \rightarrow c C \mid \varepsilon.$$

Le variabili  $S_1$  e  $S_2$  sono usate per generare rispettivamente delle stringhe di  $a$  e  $b$  in egual numero e delle stringhe di  $b$  e  $c$  in egual numero. Le variabili  $A$  e  $C$  invece generano rispettivamente sequenze di  $a$  e sequenze di  $c$  in numero casuale. Riassumendo:

$$S_1 \xRightarrow{*} a^n b^n$$

$$S_2 \xRightarrow{*} b^n c^n$$

$$A \Rightarrow a^k$$

$$C \xRightarrow{*} c^k.$$

Per la stringa  $z = abc$  abbiamo due alberi di derivazione differenti:



Una grammatica è **ambigua** quando riusciamo a trovare due diverse derivazioni per una stringa del linguaggio generato da quella grammatica.

**Definizione 1.5.1** (Grado di ambiguità di una stringa): Sia  $G = (V, \Sigma, P, S)$  una grammatica CF. Sia  $w \in \Sigma^*$ . Chiamiamo **grado di ambiguità** di  $w$  rispetto a  $G$  il numero di alberi di derivazione di  $w$ , oppure il numero di derivazioni leftmost di  $w$ .

Ovviamente, se una stringa non appartiene a  $L(G)$  ha grado di ambiguità pari a zero.

**Definizione 1.5.2** (Grado di ambiguità di una grammatica): Il **grado di ambiguità** di una grammatica  $G$  è il massimo grado di ambiguità delle stringhe  $w \in \Sigma^*$ .

Il concetto di **ambiguità** è legato al **non determinismo**: abbiamo visto nell'equivalenza tra grammatiche di tipo 2 e automi a pila che questi ultimi potevano simulare le derivazioni leftmost della grammatica. Se ad un certo punto la grammatica ha più derivazioni leftmost che mi

portano poi nella stessa stringa allora stiamo introducendo del non determinismo. Viceversa, quando guardavamo le computazioni possibili in un automa a pila e dovevamo generare le regole di produzione, quando eravamo di fronte ad una scelta dovevamo generare delle regole ambigue.

**Definizione 1.5.3** (Grado di ambiguità di un automa a pila): Il **grado di ambiguità** di un automa a pila è il numero di computazioni accettanti.

La relazione però non è biunivoca: infatti, nel linguaggio delle palindrome pari abbiamo del non determinismo ma non abbiamo ambiguità perché la metà della stringa è una sola. Viceversa, se abbiamo ambiguità sicuramente abbiamo non determinismo, perché c'è un punto di scelta dove noi possiamo sdoppiare il riconoscimento.

**Definizione 1.5.4** (Grammatiche inerentemente ambigue): Sia  $L$  un CFL. Allora  $L$  è **inerentemente ambigua** se ogni grammatica CF per  $L$  è ambigua.

Andiamo avanti la prossima volta.

## 2. Lezione 19 [14/05]

### 2.1. Ambiguità

Per parlare di ambiguità ci servirà il **lemma di Ogden**, ma in una forma leggermente diversa.

**Lemma 2.1.1** (Lemma di Ogden): Sia  $G = (V, \Sigma, P, S)$  una grammatica CF. Allora  $\exists N$  tale che  $\forall z \in L$  in cui sono marcate almeno  $N$ , possiamo scrivere  $z$  come  $z = uvwxy$  con:

1.  $vw$  contiene al più  $N$  posizioni marcate;
2.  $vx$  contiene almeno una posizione marcata;
3.  $\exists A \in V$  tale che

- $S \xRightarrow{*} uAy$ ,
- $A \xRightarrow{*} vAx$ ,
- $A \xRightarrow{*} w$ ,

e dunque  $\forall i \geq 0 \quad uv^iwx^iy \in L(G)$ .

Abbiamo una differenza sostanziale con il lemma dell'altra volta: in quest'ultimo ci veniva detto  $L$  è CF, mentre ora stiamo dicendo che la grammatica lo è, e dalla grammatica noi siamo in grado di ricavare il linguaggio, quindi è una condizione più forte di quella di prima.

Questo inoltre vale per ogni grammatica CF e non solo per quelle in FN di Chomsky.

**Dimostrazione 2.1.1.1:** La dimostrazione cambia leggermente dalla scorsa volta.

Visto che possiamo avere nodi interni con più di due figli, sia  $d$  il numero massimo di elementi sul lato destro di una produzione. Come costante prendiamo

$$N = d^{k+1}$$

e poi la dimostrazione va avanti allo stesso modo. ■

Riprendiamo l'esempio che abbiamo visto la lezione scorsa per il discorso sull'ambiguità.

**Esempio 2.1.1:** Definiamo il linguaggio

$$L = \{a^p b^q c^r \mid p = q \vee q = r\}.$$

Un automa a pila per  $L$  all'inizio scommette quale condizione verificare con il non determinismo usando una grammatica in due parti:

- una genera stringhe con  $\#_a = \#_b$  e con un numero di  $c$  qualsiasi;
- una fa lo stesso ma con  $\#_b = \#_c$  e con un numero di  $a$  qualsiasi.

Avevamo visto poi le definizioni di **grado di ambiguità di una stringa** e di un **linguaggio**.

Avere tanti alberi di derivazione è scomodo, nei compilatori soprattutto, perché ho più espressioni per lo stesso concetto, e questo dà molto fastidio.

Possiamo togliere l'ambiguità da un linguaggio? Ovvero, data  $G$  una grammatica ambigua per  $L$ , riusciamo a trovarne un'altra che generi ancora  $L$  ma non ambigua?

In generale la risposta è **NO**: esistono linguaggi che hanno solo grammatiche ambigue che li generano, e sono detti **linguaggi inerentemente ambigui**.

**Teorema 2.1.1:** Il linguaggio  $L$  dell'Esempio 2.1.1 è inerentemente ambiguo.

**Dimostrazione 2.1.1.2:** Dobbiamo dimostrare che ogni grammatica  $G$  che genera  $L$  è ambigua, quindi esiste almeno una stringa in ogni  $G$  che è generata in almeno due modi.

Sia  $G = (V, \Sigma, P, S)$  una grammatica per  $L$ . Vogliamo dimostrare che  $\exists \beta \in L$  che ammette due alberi di derivazione differenti. Useremo il lemma di Ogden molte volte.

Sia  $N$  la costante del lemma di Ogden per  $G$ , e sia  $m = \max(3, N)$ .

Definiamo la stringa

$$z = a^m b^m c^{m+m!}$$

in cui andiamo a marcare tutte le  $a$ , così ho marcato almeno  $N$  posizioni.

Decomponiamo poi  $z$  come  $z = uvwxy$  e utilizziamo la terza proprietà, quindi scegliamo come moltiplicatore  $i = 2$  generando la stringa

$$\alpha = uv^2wx^2y \in L.$$

Di questa stringa sappiamo che

$$\#_b(\alpha) = \#_b(z) + \#_b(vx) \leq m + m = 2m \stackrel{m \geq 3}{<} m + m! < \#_c(\alpha).$$

Noi sappiamo che  $\alpha \in L$ , quindi se  $b$  e  $c$  sono diverse allora sono uguali le altre due lettere:

$$\#_a(\alpha) = \#_b(\alpha).$$

Partendo da una stringa con  $a$  e  $b$  uguali, visto che abbiamo ottenuto ancora una stringa con la stessa proprietà, allora abbiamo aggiunto lo stesso numero di  $a$  e di  $b$ , quindi

$$\#_a(vw) = \#_b(vw).$$

Questa proprietà, che abbiamo appena dimostrato per  $i = 2$ , diventa una proprietà della decomposizione che abbiamo fatto prima.

Sfruttiamo la seconda condizione: sapendo di avere almeno una posizione marcata, e che queste sono solo  $a$ , possiamo dire che

$$\#_a(vx) \geq 1 \implies \#_b(vw) \geq 1.$$

Vediamo in dettaglio come è fatta  $vx$ : se  $v$  e  $x$  contengono almeno due lettere diverse stiamo perdendo la struttura della stringa, perché  $v^2$  o  $x^2$  rompono il pattern.

Mettiamoci quindi nel caso in cui  $v = a^j$  è formata da sole  $a$  e  $x = b^j$  è formata da sole  $b$ , imponendo inoltre che  $1 \leq j \leq m$ .

Riprendiamo la terza condizione, considerando stringhe generiche nella forma

$$\forall i \geq 0 \quad uv^iwx^iy = a^{m+(i-1)j}b^{m+(i-1)j}c^{m+m!} \in L.$$

Vogliamo una stringa con  $\#_a = \#_b = \#_c$ , e questo lo facciamo imponendo

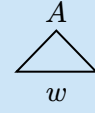
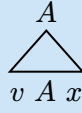
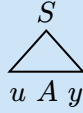
$$(i-1)j = m! \implies i = \frac{m!}{j} + 1$$

e questa divisione è intera per la condizione imposta sulla  $j$ .

Con questa imposizione otteniamo la stringa

$$\beta = a^{m+m!}b^{m+m!}c^{m+m!} \in L.$$

Sempre grazie alla terza condizione possiamo vedere gli alberi di derivazione di questa stringa:



L'albero di derivazione di  $\beta$  è formato dal primo albero, poi ha ripetuto  $i$  volte quello centrale, e infine ha usato l'ultimo albero come tappo per terminare.

Mettiamo da parte questi risultati per adesso. Prendiamo ora  $z' = a^{m+m!}b^m c^m$  in cui marchiamo tutte le  $c$ . Facciamo poi la decomposizione di  $z'$  come  $z' = u'v'w'x'y'$ .

Ripetiamo la dimostrazione appena fatta, ma ragionando sulla seconda parte della stringa.

Quello che otteniamo è che:

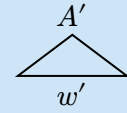
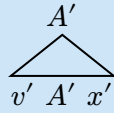
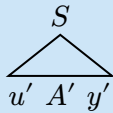
- i fattori  $v'$  e  $x'$  di  $z'$  sono formati rispettivamente dalle sole  $b$  e dalle sole  $c$ , ovvero sono le stringhe

$$v' = b^k \wedge x' = c^k \mid 1 \leq k \leq m;$$

- possiamo pompare la stringa  $z'$  ottenendo una stringa con  $\#_a = \#_b = \#_c$ , ovvero

$$i = \frac{m!}{k} + 1 \implies \beta = a^{m+m!}b^{m+m!}c^{m+m!} \in L.$$

Come prima, vediamo gli alberi di derivazione di questa stringa:



Se costruiamo l'albero totale di  $\beta$  ora uniamo la parte esterna,  $i$  volte la parte interna e infine il tappo, ma questo albero è diverso da quello di prima perché qua stiamo pompando le  $b$  e le  $c$ , mentre prima pompavamo le  $a$  e le  $b$ .

Ma allora abbiamo due alberi diversi per la stessa stringa, quindi  $G$  è ambigua. Visto che abbiamo preso una grammatica  $G$  generica, allora ogni  $G$  per  $L$  è ambigua, e quindi  $L$  è inerentemente ambiguo. ■

Nel caso del linguaggio Esempio 2.1.1, il **grado di ambiguità** è 2. In alcuni casi, il grado di ambiguità cresce in base alla lunghezza della stringa che si sta riconoscendo, rendendo di fatto **infinito** il grado della grammatica e/o del linguaggio.

Il concetto di ambiguità è importante perché parlare di ambiguità nelle grammatiche è equivalente a parlare di ambiguità negli **automi a pila**.

Avevamo visto che potevamo trasformare una grammatica  $G$  in un PDA simulando con quest'ultimo le derivazioni leftmost. Ecco, questa trasformazione riesce a mantenere il grado di ambiguità  $k$  della grammatica  $G$ . Vale anche il viceversa: infatti, possiamo trasformare un PDA che riconosce una stringa in  $k$  modi diversi in una grammatica con grado di ambiguità  $k$  usando una costruzione leggermente diversa della nostra, che invece aumentava e non di poco il grado di ambiguità.

## 2.2. Ambiguità e non determinismo

L'ambiguità è legata parzialmente anche al discorso del **non determinismo**.

Se una stringa può essere generata in due modi diversi allora l'automa è in grado di riconoscerla in due modi diversi, quindi l'automa è per forza non deterministico.

In poche parole, se abbiamo una grammatica  $G$  **ambigua** per il linguaggio  $L$ , allora  $L$  deve essere riconosciuto per forza da un automa non deterministico.

Riprendiamo velocemente la definizione di **automi a pila**. Essi sono delle tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

definiti da una funzione di transizione

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow \text{PF}(Q \times \Gamma^*).$$

Avevamo già visto la definizione di **PDA deterministico**, ma riprendiamola.

**Definizione 2.2.1** (PDA deterministico): Il PDA  $M$  è **deterministico** se:

1.  $\forall q \in Q \quad \forall A \in \Gamma \quad \delta(q, \varepsilon, A) \neq \emptyset \implies \forall a \in \Sigma \quad \delta(q, a, A) = \emptyset$ ;
2.  $\forall q \in Q \quad \forall A \in \Gamma \quad \forall \sigma \in \Sigma \cup \{\varepsilon\} \quad |\delta(q, \sigma, A)| \leq 1$ .

In poche parole, le due condizioni ci dicono che:

1. se nello stato definito dalla coppia stato-pila abbiamo delle  $\varepsilon$ -mosse allora non possiamo anche leggere un carattere dal nastro;
2. la dimensione dell'immagine della funzione di transizione è al massimo 1.

Abbiamo visto due diverse **accettazioni** per gli automi a pila, e abbiamo dimostrato che nel caso non deterministico queste sono equivalenti. Nella trasformazione da stati finali a pila vuota, ogni volta che si finiva in uno stato finale si scommetteva di aver finito l'input svuotando la pila, ma lo facendo non deterministicamente. La trasformazione da pila vuota a stati finali



invece ogni volta che svuotava la pila andava in uno stato finale, ma questo non introduceva non determinismo perché facevamo una pura simulazione e aggiungevamo regole che non interferivano tra loro.

Sappiamo inoltre che i CFL sono **equivalenti** ai PDA. Cosa possiamo dire dei CFL deterministici?

Definiamo la classe **DCFL** classe dei **linguaggi CF deterministici**, equivalente ai **DPDA** (PDA deterministici) che accettano per **stati finali**. Abbiamo specificato l'accettazione perché nel caso deterministico non abbiamo la stessa accettazione: con una pila vuota infatti andiamo ad accettare meno linguaggi. Addirittura ci sono dei **linguaggi regolari** che non riusciamo ad accettare.

**Esempio 2.2.1:** Definiamo il linguaggio regolare

$$L = a(aa)^*$$

che possiamo riconoscere tranquillamente con un DFA a due stati.

Abbiamo un DPDA che accetta per pila vuota. Prendiamo la stringa  $z = aaa$ . L'automa è programmato per riconoscere le stringhe di lunghezza pari, quindi appena legge la prima  $a$  si deve fermare per accettare, ma questo non accade perché si pianta svuotando la pila ma con ancora dell'input da leggere.

In generale, una struttura con stringhe prefisse di altre non riesce ad essere riconosciuta da DPDA per pila vuota. Come vediamo, è una classe particolare, con alcuni regolari e alcuni CF.

Nei parser il trucco è mettere un marcatore alla fine per indicare all'automa di svuotare la pila.

La questione dell'ambiguità si collega al non determinismo. Infatti, se  $L$  è un CFL ed è anche **inerentemente ambiguo**, allora ogni PDA per  $L$  deve essere non deterministico, quindi

$$L \in \text{CFL} / \text{DCFL} .$$

Questa affermazione mostra che i CFL sono diversi dai DCFL, e che questi sono meno potenti perché alcuni linguaggi non li possono proprio riconoscere.

Negli automi a stati finiti avevamo la **costruzione per sottoinsiemi** per rimuovere il non determinismo. Con gli automi a pila non possiamo utilizzare questa costruzione perché avendo una sola pila non riusciamo a tenere traccia di tutto quello che viene fatto su essa.

Breve **OT**: se abbiamo due pile il modello diventa potente quanto le macchine di Turing.

Ma vale anche il viceversa? Ovvero dato un automa non deterministico allora abbiamo per forza un linguaggio o una grammatica ambigua?

**Esempio 2.2.2:** Sia  $L$  il linguaggio delle palindrome pari, ovvero

$$L = \{ww^R \mid w \in \{a,b\}^*\}.$$

Questo linguaggio non è deterministico, ma non l'abbiamo ancora dimostrato, vedremo.

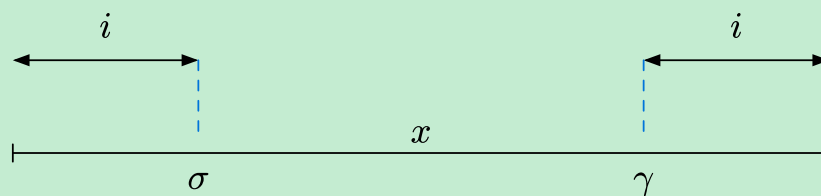
Anche se gli automi per questo linguaggio sono non deterministici, mi va bene una sola scommessa se la stringa è palindroma, quindi non abbiamo ambiguità.

Vediamo una grammatica  $G$  per  $L$ :

$$S \longrightarrow aSa \mid bSb \mid \varepsilon.$$

Come vediamo,  $G$  non è ambigua, e infatti nemmeno  $L$  lo è.

**Esempio 2.2.3:** Vediamo il complemento del linguaggio precedente, ovvero il linguaggio delle stringhe nelle quali esiste almeno una posizione alla stessa distanza dai bordi in cui i caratteri sono diversi.



Ovviamente questo è non deterministico: dobbiamo scommettere su un simbolo che non ci piace  $\sigma$ , far passare un po' di stringa, trovare il suo compare  $\gamma$ , controllare che sono diversi e vedere se la distanza dalla fine è uguale a quella tra il primo e l'inizio.

Un automa a pila per questo linguaggio carica sulla pila delle  $X$ , arrivando ad altezza  $i$  a  $\sigma$ , poi scorre lasciando stare la pila, infine controlla  $\sigma$  con  $\gamma$  e inizia a scaricare. In poche parole, usiamo la pila come contatore.

Questo automa è ovviamente ambiguo perché ci possono essere più coppie possibili che rendono vera l'accettazione della stringa.

Possiamo evitare l'ambiguità in questo automa?

Dobbiamo scegliere la **scommessa giusta**, ovvero dobbiamo verificare di avere una parte iniziale  $i$  poi la stessa  $i$  alla fine ma rovesciata. Per indovinare subito la prima posizione che non va bene sulla pila non salviamo più la distanza, ma quello che leggiamo. Dopo un po' scommettiamo, arriviamo alla fine, controlliamo e svuotiamo.

Con questo magheggio riusciamo a renderlo non ambiguo, perché l'automa fa tante scommesse ma riesce a beccare solo la prima posizione sbagliata, perché le parti prima e dopo saranno invece uguali.

**Esempio 2.2.4:** Per sfizio scriviamo  $L^C$  in termini di grammatica.

Abbiamo una posizione che è fallata, quindi prima inseriamo qualcosa di uguale ai bordi, poi inseriamo l'elemento sbagliato, e poi aggiungiamo quello che vogliamo.

Le regole di produzione sono

$$S \longrightarrow aSa \mid bSb \mid T$$

$$T \longrightarrow aUb \mid bUa$$

$$U \longrightarrow aU \mid bU \mid \varepsilon.$$

### 3. Lezione 20 [16/05]

Oggi vediamo le **proprietà di chiusura** dei linguaggi CFL e DCFL.

Introduciamo subito due linguaggi che ci serviranno durante la lezione.

**Definizione 3.1** (Linguaggi comodi): Definiamo il **linguaggio**

$$L' = \{a^i b^j c^k \mid i = j\}$$

e il **linguaggio**

$$L'' = \{a^i b^j c^k \mid j = k\}$$

entrambi DCFL, molto facile da dimostrare.

#### 3.1. Operazioni insiemistiche

Partiamo con le **operazioni insiemistiche**.

##### 3.1.1. Unione

###### 3.1.1.1. CFL

Due linguaggi CFL possono essere «uniti» in uno solo con l'operazione di **unione** mantenendo la proprietà di essere CFL, e lo possiamo vedere fornendo una grammatica per questa operazione.

Siano

$$G' = (V', \Sigma, P', S') \quad | \quad G'' = (V'', \Sigma, P'', S'')$$

due grammatiche CF. Creiamo la grammatica

$$G = (V, \Sigma, P, S)$$

formata da:

- $V$  insieme delle **variabili** formato dall'unione dei due insiemi, ovvero

$$V = V' \cup V'';$$

- $S$  nuovo **assioma**, dal quale decideremo quale strada prendere nella derivazione delle stringhe di questo nuovo linguaggio;
- $P$  insieme delle regole di produzione, che manteniamo tutte ma alle quali ne aggiungiamo due per fare da ponte, ovvero

$$P = P' \cup P'' \cup \{(S \rightarrow S' \mid S'').\}$$

###### 3.1.1.2. DCFL

Nel caso deterministico è più complicato: nella grammatica che abbiamo definito poco fa abbiamo introdotto del non determinismo, che però in questo caso non possiamo avere. Come lo dimostriamo? Esistono invece due linguaggi che rompono la chiusura per l'**unione**?

**Esempio 3.1.1.2.1:** Prendiamo i due linguaggi definiti nella Definizione 3.1.

Abbiamo detto che sono entrambi DCFL, ma la sua unione

$$L = L' \cup L'' = \{a^i b^j c^k \mid i = j \vee c = k\}$$

deve essere riconosciuta da un automa non deterministico.

Quindi **non** siamo chiusi rispetto all'unione, ma almeno siamo caduti ancora nel tipo 2.

### 3.1.2. Intersezione

Per quanto riguarda l'**intersezione** va male per entrambi.

#### 3.1.2.1. CFL

**Esempio 3.1.2.1.1:** Prendiamo ancora i due linguaggi della Definizione 3.1.

Definiamo il linguaggio

$$L = L' \cap L'' = \{a^i b^j c^k \mid i = j = k\}$$

che abbiamo visto non essere nemmeno CFL.

Nei linguaggi regolari utilizzavamo l'**automa prodotto**, ma in questo caso non possiamo: infatti, dovendo mandare avanti due automi allo stesso momento servirebbero due pile, e noi avendone solo una possiamo avere delle operazioni discordanti.

#### 3.1.2.2. DCFL

Vale lo stesso discorso dei CFL.

### 3.1.3. Intersezione con un regolare

L'operazione di **intersezione con un regolare** non l'abbiamo vista nei linguaggi regolari perché ovviamente in quel campo già ci eravamo lol.

#### 3.1.3.1. CFL

Nel caso CFL **abbiamo chiusura**: infatti, usando un automa prodotto che manda in parallelo un automa a pila e un automa a stati finiti ci serve una sola pila.

#### 3.1.3.2. DCFL

Stesso discorso per i DCFL.

### 3.1.4. Complemento

Ora veniamo all'ostica operazione di **complemento**.

#### 3.1.4.1. CFL

Con le **leggi di De Morgan** possiamo esprimere l'operazione di intersezione con unione e complemento, ovvero

$$L_1 \cap L_2 = (L_1^C \cup L_2^C)^C.$$

Visto che i CFL sono chiusi rispetto all'unione, se lo fosse anche il complemento allora lo sarebbe anche l'intersezione, ma questo abbiamo fatto vedere che non è vero.

**Esempio 3.1.4.1.1:** Definiamo il linguaggio

$$K = \{a^i b^j c^k \mid i \neq k \vee j \neq k\} \cup \{x \notin a^* b^* c^*\}.$$

Questo linguaggio è CFL, ma il suo complemento

$$K^C = \{a^n b^n c^n \mid n \geq 0\} = L' \cap L''$$

invece non è CFL.

**Esempio 3.1.4.1.2:** Al contrario, dato il linguaggio

$$L = \{ww \mid w \in \{a, b\}^*\}$$

che non è CFL, se ne calcoliamo il complemento questo è CFL.

### 3.1.4.2. DCFL

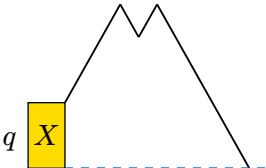
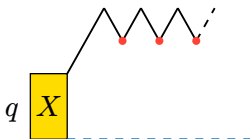
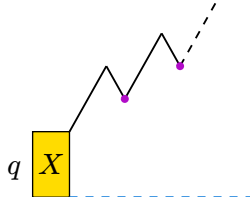
Nel caso deterministico non possiamo usare De Morgan perché non siamo chiusi l'unione, quindi nessun ragionamento di quel tipo ha senso.

Incredibilmente, i DCFL **sono chiusi** rispetto all'operazione di complemento.

Nei linguaggi regolari ci bastava **completare** l'automa e poi invertire «banalmente» gli stati finali con i non finali e viceversa, ma qui non è così facile perché non siamo sempre sicuri che l'automa arrivi in fondo alla sua computazione. Inoltre, nei DCFL abbiamo a disposizione le  $\varepsilon$ -mosse, che non possiamo togliere perché perdiamo potenza, a differenza degli automi a stati finali dove si manteneva lo stesso potere riconoscitivo.

Ok teniamo le  $\varepsilon$ -mosse, perché sono fastidiose? Perché l'automa, con una sequenza di  $\varepsilon$ -mosse, potrebbe entrare in loop infinito e quindi non accettare la stringa. Potremmo fregarcene, ma invece ci interessa molto, perché questa situazione di non accettazione deve essere presa in considerazione ed essere accettata.

Che possibili casi abbiamo durante una sequenza di  $\varepsilon$ -mosse? Supponiamo di essere nello stato  $q$  e di avere sulla pila il carattere  $X$ , senza mosse che possono leggere l'input.

Nessun loop	Loop sullo stesso piano	Loop crescente
		

In ordine abbiamo:

- una sequenza di  $\varepsilon$ -mosse che poi cancella il simbolo  $X$  sulla pila;
- una sequenza di  $\varepsilon$ -mosse che ogni tanto ritorna in una configurazione con stato  $p$  e  $Y$  sulla pila alla stessa altezza della configurazione analoga precedente;

- una sequenza di  $\varepsilon$ -mosse che ogni tanto ritorna in una configurazione con stato  $p$  e  $Y$  sulla pila ad altezza maggiore della configurazione analoga precedente.

Ci interessa sapere queste informazioni, e data la funzione di transizione è possibile conoscere queste informazioni per ogni coppia di stato  $q$  e carattere  $X$ .

Vediamo quindi come costruire questo automa per il complemento, anche se dobbiamo passare per molti passaggi intermedi. Buona fortuna.

Sia  $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  un automa a pila deterministico che accetta il linguaggio  $L$ . Trasformiamo  $M$  nell'automa  $M'$ , sempre per  $L$ , che ha la proprietà di **non finire mai in loop**, ovvero riesce sempre a leggere tutto l'input. Inoltre,  $M'$  risulterà essere ancora deterministico.

Costruiamo quindi l'automa

$$M' = (Q', \Sigma, \Gamma', \delta', q'_0, X_0, F')$$

definito da:

- **$Q'$  insieme degli stati** formato da quelli di  $M$  e da tre nuovi stati, che ci permetteranno di fare il solito truccaccio di riempire la pila e di evitare i loop, ovvero

$$Q' = Q \cup \{q'_0, d, f\};$$

- **$\Gamma'$  alfabeto di lavoro** che aggiunge solo il carattere del truccaccio, ovvero

$$\Gamma' = \Gamma \cup \{X_0\};$$

- **$F'$  insieme degli stati finali** che aggiunge il nuovo stato appena aggiunto, ovvero

$$F' = F \cup \{f\}.$$

La funzione di transizione viene arricchita di un sacco di mosse, che ora vediamo in ordine.

Prima di tutto facciamo il classico truccaccio, ovvero lasciamo qualcosa sotto la pila così che  $M'$  non si blocchi se durante la simulazione di  $M$  quest'ultimo dovesse svuotare la pila. Aggiungiamo quindi la regola

$$\delta'(q'_0, \varepsilon, Z_0) = (q_0, Z_0 X_0)$$

che appunto mette il tappo in fondo e ci permette di iniziare la simulazione di  $M$ .

Se in una certa configurazione non abbiamo  $\varepsilon$ -mosse o mosse normali a disposizione allora finiamo in uno stato trappola, il **death state**, ovvero

$$\forall q \in Q \quad \forall a \in \Sigma \cup \{\varepsilon\} \quad \forall X \in \Gamma \quad \delta(q, a, X) = \emptyset \implies \delta'(q, a, X) = (d, X).$$

Se ad un certo punto troviamo  $X_0$  sulla pila vuol dire che quest'ultima è stata svuotata da  $M$ , quindi l'automa si è bloccato e con  $M'$  devo andare nel death state, ovvero

$$\forall a \in \Sigma \quad \delta'(q, a, X_0) = (d, X_0).$$

Nello stato trappola leggo l'input per intero senza toccare altro, quindi

$$\forall a \in \Sigma \quad \forall X \in \Gamma \quad \delta'(d, a, X) = (d, X).$$

Se in una certa configurazione ci sono  $\varepsilon$ -mosse potrei entrare in un loop, ma questa informazione l'abbiamo già calcolata, quindi con la presenza di  $\varepsilon$ -mosse e di un loop da  $(q, X)$  abbiamo

$$\delta'(q, \varepsilon, X) = \begin{cases} (d, X) & \text{se il loop non visita uno stato finale} \\ (f, X) & \text{altrimenti} \end{cases}.$$

Se sono finito nello stato finale è perché ho trovato un loop con stato finale, ma se l'ho trovato non alla fine della stringa devo andare nello stato trappola, quindi

$$\forall a \in \Sigma \quad \forall X \in \Gamma \quad \delta'(f, a, X) = (d, X).$$

In tutti gli altri casi teniamo le mosse che già c'erano, quindi

$$\forall q \in Q \quad \forall a \in \Sigma \cup \{\varepsilon\} \quad \forall X \in \Gamma \quad \delta'(q, a, X) = \delta(q, a, X).$$

**Perfetto**, ora che abbiamo un automa che non si blocca mai costruiamo un automa per il complemento. Facciamo un po' di renaming per semplicità.

Sia  $M$  un DPDA per  $L$  che scandisce sempre l'intero l'input. Vogliamo costruire

$$M' = (Q', \Sigma, \Gamma, \delta', q'_0, Z_0, F')$$

per il complemento di  $M$ , ovvero quando  $M$  risponde **SI** noi rispondiamo **NO** e viceversa.

Ora  $M$  arriva sempre in fondo all'input, ma può fare comunque  $\varepsilon$ -mosse alla fine. Se durante queste ultime sequenze visitiamo almeno uno stato finale dobbiamo rifiutare, altrimenti accettiamo.

Definiamo quindi  $M'$  formato da:

- **$Q'$  insieme degli stati** che memorizza, oltre allo stato nel quale si trova, anche se nella sequenza di  $\varepsilon$ -mosse che stiamo facendo abbiamo visto o meno uno stato finale. In realtà, ci dice di più questa flag, che il prof chiama un **bit e mezzo**, perché infatti  $Q'$  è definito come

$$Q' = Q \times \{y, n, A\}$$

con le flag che indicano rispettivamente se nella sequenza abbiamo visitato uno stato in  $F$ , se non l'abbiamo fatto o se non l'abbiamo fatto e non siamo più in grado di fare  $\varepsilon$ -mosse;

- **$F'$  insieme degli stati finali** formato dalle coppie che hanno come seconda componente la  $A$ , perché non sono passato da stati finali e mi sono fermato, quindi

$$F' = Q \times \{A\};$$

- **$q'_0$  stato iniziale** che dipende dallo stato iniziale vecchio, ovvero

$$q'_0 = \begin{cases} [q_0, n] & \text{se } q_0 \notin F \\ [q_0, y] & \text{se } q_0 \in F \end{cases};$$

- **$\delta'$  funzione di transizione** che dati  $q \in Q$  e  $X \in \Gamma$ :
  - se  $\delta(q, \varepsilon, X) = (p, \gamma)$  allora posso eseguire delle  $\varepsilon$ -mosse, quindi in base alla seconda componente degli stati definisco

$$\delta'([q, y], \varepsilon, X) = ([p, y], \gamma)$$

$$\delta'([q, n], \varepsilon, X) = \begin{cases} ([p, y], \gamma) & \text{se } p \in F \\ ([p, n], \gamma) & \text{altrimenti} \end{cases}$$

$$\delta'([q, A], \varepsilon, X) = \emptyset;$$

- se invece nella configurazione corrente ho finito le  $\varepsilon$ -mosse allora posso avere delle mosse che leggono simboli in input. Potendo fare questa operazione di lettura, dobbiamo dimenticarci dell'ultimo loop eseguito se contiene degli stati finali, quindi se  $\delta(q, a, X) = (p, \gamma) \mid a \in \Sigma$  allora



$$\delta'([q, y], a, X) = \begin{cases} ([p, y], \gamma) & \text{se } p \in F \\ ([p, n], \gamma) & \text{altrimenti} \end{cases}$$

Se invece nell'ultimo loop abbiamo terminato il giro senza passare da stati finali passiamo per  $A$ , ovvero

$$\begin{aligned} \delta'([q, n], \varepsilon, X) &= ([q, A], X) \\ \delta'([q, A], a, X) &= \begin{cases} ([p, y], \gamma) & \text{se } \varepsilon \in F \\ ([p, n], \gamma) & \text{altrimenti} \end{cases} \end{aligned}$$

**Finita**, finalmente, questa costruzione senza senso.

### 3.1.5. Riassunto

	CFL	DCFL
Unione	✓	✗
Intersezione	✗	✗
Intersezione con un regolare	✓	✓
Complemento	✗	✓

## 3.2. Operazioni regolari

Vediamo ora le **operazioni regolari**.

### 3.2.1. Prodotto

La prima operazione regolare, ovvero l'unione, l'abbiamo già analizzata. Vediamo ora il **prodotto**.

#### 3.2.1.1. CFL

Per i CFL è facile creare una grammatica  $G$  a partire da due grammatiche  $G'$  e  $G''$  CFG con un assioma  $S$  e una regola di produzione

$$S \rightarrow S' S''$$

che permetta di produrre le due stringhe separatamente a partire dai loro assiomi.

#### 3.2.1.2. DCFL

Purtroppo, i DCFL **non sono chiusi** rispetto al prodotto.

**Esempio 3.2.1.2.1:** Prendiamo di nuovo i linguaggi della Definizione 3.1, che sono entrambi deterministici, e creiamo il linguaggio

$$L_0 = L' \cup dL''$$

che è deterministico perché in base al primo carattere capisce subito che linguaggio riconoscere. Creiamo ora il linguaggio

$$L_1 = \{\varepsilon, d\}L_0.$$

In base al carattere che scegliamo di anteporre alle stringhe di  $L_0$  noi possiamo avere un numero di  $d$  iniziali differenti. Scriviamo quindi  $L_1$  come l'insieme

$$L_1 = \{d^s a^i b^j c^k \mid s \in \{0, 1, 2\}\}.$$

Analizziamo i vari casi:

- se  $s = 0$  allora stiamo scegliendo  $\varepsilon$  e  $L'$ , quindi  $i = j$ ;
- se  $s = 2$  allora stiamo scegliendo  $d$  e  $dL''$ , quindi  $j = k$ ;
- se  $s = 1$  allora:
  - ▶ stiamo scegliendo  $d$  e  $L'$ , quindi  $i = j$ ;
  - ▶ stiamo scegliendo  $\varepsilon$  e  $dL''$ , quindi  $j = k$ .

Filtriamo alcune stringhe di  $L_1$  calcolando

$$L_1 \cap da^*b^*c^* \stackrel{s=1}{=} \{da^i b^j c^k \mid i = j \vee j = k\}$$

che ovviamente non è DCFL. Ricordandoci che i DCFL sono chiusi rispetto all'intersezione con un regolare, se il linguaggio di destra non è DCFL allora non lo è nemmeno  $L_1$ , che era ottenuto però come concatenazione di due linguaggi DCFL.

Questa cosa è **tristissima**: non siamo chiusi nemmeno con un linguaggio finito, è drammatico.

Fatto stranissimo, se invece concateniamo con un linguaggio regolare a destra si ottiene un linguaggio DCFL, senza senso questa classe di linguaggi.

### 3.2.2. Star

Vediamo infine la **star** per finire le operazioni regolari.

#### 3.2.2.1. CFL

Nei CFL basta creare una nuova grammatica  $G$  a partire da  $G'$  CFG con le regole di produzione

$$S \longrightarrow S'S \mid \varepsilon$$

per iniziare a concatenare tante stringhe di  $G'$  a nostro piacere.

Un automa invece ogni volta che arriva in uno stato finale fa ripartire la computazione dall'inizio.

#### 3.2.2.2. DCFL

Sfigati come sono, i DCFL non sono chiusi nemmeno per la star di Kleene.

**Esempio 3.2.2.2.1:** Non ho ben capito l'esempio che ha scritto.

### 3.2.3. Riassunto

	CFL	DCFL

Prodotto	✓	✗
Star	✓	✗

## 4. Lezione 21 [21/05]

### 4.1. Riassunto chiusura operazioni

	CFL	DCFL
Unione	✓	✗
Intersezione	✗	✗
Intersezione con regolare	✓	✓
Complemento	✗	✓
Prodotto	✓	✗
Star	✓	✗

Come vediamo, i DCFL, tolti l'intersezione con regolari che non è proprio un'operazione interna, ha poche proprietà di chiusura. Molto molto male.

### 4.2. CFL vs DCFL

Per i CFL avevamo due criteri molto potenti per dire la **NON** appartenenza di un linguaggio  $L$  generico a questa classe. Abbiamo delle tecniche anche per i DCFL? **SI**, menomale.

Come per i CFL, anche i DCFL hanno il **pumping lemma**, o meglio, **i pumping lemma**: ce ne sono tanti, e di solito vanno bene solo su alcuni esempi, quindi sono molto tecnici e specifici.

Una seconda tecnica è dimostrare che  $L$  è **inerentemente ambiguo**, per far sì che ogni automa per  $L$  sia ambiguo e quindi che  $L$  è non deterministico.

**Esempio 4.2.1:** Avevamo visto, con questa tecnica, la dimostrazione di

$$L = \{a^p b^q c^r \mid p = q \vee q = r\} \in \text{CFL}.$$

Una terza tecnica è usare le **proprietà di chiusura** rispetto al complemento. Se facciamo vedere che  $L^C \notin \text{CFL}$  allora  $L$  non può essere DCFL perché questi ultimi sono chiusi rispetto al complemento, ed essendo  $L^C \notin \text{CFL}$  allora vale anche  $L^C \notin \text{DCFL}$ .

**Esempio 4.2.2:** Definiamo il linguaggio

$$L = \{x \in \{a, b\}^* \mid \nexists w \mid x = ww\}$$

formato dalle stringhe che non sono decomponibili come due stringhe uguali concatenate.

Calcoliamo il suo complemento

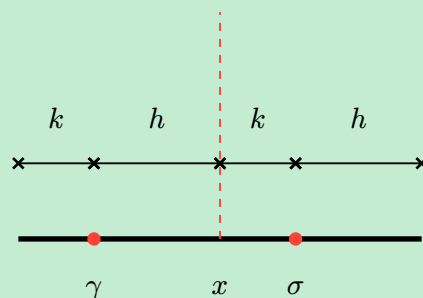
$$L^C = \{ww \mid w \in \{a, b\}^*\}.$$

Con il pumping abbiamo dimostrato che questo linguaggio non è CFL. Ma allora  $L$  non è DCFL, quindi sapendo che è CFL cerchiamo un PDA per esso.

Creiamo una sorta di automa prodotto che simula l'intersezione con un regolare:

- una prima componente è un **automa a stati finiti** che controlla la lunghezza della stringa. Se questa è dispari allora accettiamo, altrimenti guardiamo l'altra componente;
- la seconda componente è un **automa a pila**, e ora vediamo come è fatto.

Definita  $m$  la quantità che indica la metà della lunghezza della stringa in input, l'automa a pila deve trovare due simboli a distanza  $m$  che sono diversi.



Abbiamo quindi un simbolo  $\gamma$  a distanza  $k$  dall'inizio che deve essere diverso da un simbolo  $\sigma$  a distanza  $h + k = m$  da  $\gamma$ .

La prima idea per risolvere questo problema è quella di azzeccare dove sta la metà, ma questo è molto difficile quindi è un campanello che ci deve dire che non ci potrebbe servire. E infatti.

Facciamo una cosa più esotica: grazie alla bellissima **proprietà commutativa** della somma sappiamo che  $h + k = k + h$ . In particolare, proviamo a invertire la parte centrale della stringa, ovvero proviamo a pensare alla stringa  $x$  come se fosse formata da due pezzi lunghi  $k$  e da due pezzi lunghi  $h$ .

Vediamo la soluzione divisa in fasi:

#### 1. prima fase

- leggiamo l'input e carichiamo un simbolo sulla pila come contatore;
- ad un certo punto, non deterministicamente scegliamo il simbolo sospetto  $\gamma$  da controllare. A questo punto abbiamo caricato  $k$  simboli sulla pila;

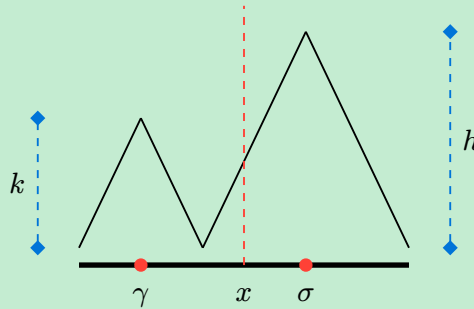
#### 2. seconda fase

- scarichiamo i  $k$  simboli sulla pila leggendo altri  $k$  simboli in input, arrivando fino al simbolo iniziale della pila. Con questa mossa abbiamo letto i primi due blocchi di  $k$  simboli;

#### 3. terza fase

- ripetiamo la prima fase, quindi iniziamo a caricare sulla pila dei caratteri leggendo l'input;

- ad un certo punto, sempre non deterministicamente, scegliamo il secondo simbolo sospetto  $\sigma$  tale che  $\gamma \neq \sigma$ . Questo controllo lo possiamo fare con il controllo a stati finiti. A questo punto abbiamo caricato  $h$  simboli sulla pila;
4. **quarta fase**
- come nella seconda fase, andiamo a scaricare gli  $h$  simboli che abbiamo sulla pila, sempre leggendo l'input.



Se abbiamo azzeccato bene il primo simbolo e bene il secondo simbolo arriviamo alla fine dell'input che abbiamo fatto una salita e una discesa di  $k$  e una salita e una discesa di  $h$ .

**Esempio 4.2.3:** Vediamo una grammatica per il linguaggio precedente.

Le regole di produzione sono:

$$\begin{aligned} S &\longrightarrow AB \mid BA \mid A \mid B \\ A &\longrightarrow aAa \mid aAb \mid bAa \mid bAb \mid a \\ B &\longrightarrow aBa \mid aBb \mid bBa \mid bBb \mid b. \end{aligned}$$

Se scegliamo solo una lettera generiamo stringhe dispari, che controlla l'automa a stati finiti. Se scegliamo invece una concatenazione di due lettere allora abbiamo che

$$\begin{aligned} A &\stackrel{*}{\Rightarrow} xAy \Leftrightarrow xay \quad | \quad |x| = |y| \\ B &\stackrel{*}{\Rightarrow} zBv \Leftrightarrow zbv \quad | \quad |z| = |v|. \end{aligned}$$

Ma allora stiamo generando della stringhe

$$S \Rightarrow AB \stackrel{*}{\Rightarrow} \underbrace{xayz} \quad | \quad |x| + |v| = |y| + |z|.$$

Stesso discorso lo possiamo fare per  $S \Rightarrow BA$ .

**Esempio 4.2.4:** Ora che abbiamo visto un automa a pila e anche una grammatica per  $L$ , possiamo usare il primo risultato per dire che  $L$  non può essere deterministico perché con le proprietà di chiusura  $L^C$  dovrebbe essere DCFL.

Vediamo ora un altro linguaggio con un esempio.

**Esempio 4.2.5:** Definiamo quindi il linguaggio

$$L = \{ww^R \mid w \in \{a,b\}^*\}$$

che ovviamente è CFL, ed è infatti molto facile definire un automa a pila per  $L$ .

Abbiamo visto che  $L^C$  è anch'esso CFL, la scorsa lezione, usando una costruzione con la pila come contatore o con la pila come «ricercatore» della prima occorrenza sbagliata.

Quindi in questo caso il criterio di chiusura dei DCFL non ci può aiutare. Inoltre, non ci può aiutare nemmeno il dimostrare  $L$  inerentemente ambiguo, perché questo linguaggio non è ambiguo, visto che la metà è una sola (se uso il contatore) o che mi sto ricordando quello che sto guardando (se nella pila butto i caratteri).

Ok possiamo usare il pumping lemma o il lemma di Ogden, però vediamo un quarto criterio.

Per introdurre questo nuovo criterio dobbiamo riprendere la **relazione di Myhill-Nerode** che abbiamo definito nei linguaggi regolari. Dato un linguaggio  $L \subseteq \Sigma^*$ , definiamo la relazione

$$R \subseteq \Sigma^* \times \Sigma^* \mid x R y \iff (\forall z \in \Sigma^* \quad (xz \in L \iff yz \in L)).$$

Avevamo visto che  $R$  era una **relazione di equivalenza** e le sue **classi di equivalenza** erano gli stati dell'**automa minimo**. Vediamo come useremo  $R$  per i DCFL.

**Teorema 4.2.1:** Se ogni classe di equivalenza di  $R$  ha cardinalità finita allora  $L$  non è DCFL.

La dimostrazione è combinatoria: preso il linguaggio  $L$ , si va ad assumere che esso sia DCFL e si dimostra che esiste almeno una classe di equivalenza con cardinalità infinita.

Applichiamolo subito all'ultimo esempio visto.

**Esempio 4.2.6:** Definiamo di nuovo il linguaggio

$$\text{PAL} = \{x \in \{a,b\}^* \mid x = x^R\}.$$

Facciamo vedere che

$$x, y \in \{a,b\}^* \mid x \neq y \implies (x, y) \notin R,$$

ovvero che ogni classe di equivalenza è formata da un solo elemento.

Prendiamo quindi due stringhe generiche

$$\begin{aligned} x &= x_1 \dots x_n \\ y &= y_1 \dots y_m \end{aligned}$$

e supponiamo di averle scritte in ordine di lunghezza, quindi  $n \leq m$ .

Per dimostrare che queste due stringhe non sono in relazione devo far vedere che esiste una stringa  $z$  che le distingue. Dividiamo in due casi l'analisi.

Se esiste un indice che pesca da  $x$  e da  $y$  due caratteri diversi, ovvero se

$$\exists i \in \{1, \dots, n\} \mid x_i \neq y_i$$

allora scegliamo la stringa  $z = x^R$  tale che

$$\begin{aligned}xz &= xx^R \in \text{PAL} \\yz &= yx^R = y_1 \dots y_m x_n \dots x_1 \notin \text{PAL}\end{aligned}$$

perché

- alla prima ho accodato proprio sé stessa ma rovesciata;
- alla seconda ho accodato  $x^R$  che però ha  $x_i \neq y_i$  alla stessa distanza dai bordi.

Se invece tutti i caratteri di  $x$  sono uguali ai primi  $n$  caratteri di  $y$ , ovvero se

$$\forall i \in \{1, \dots, n\} \quad x_i = y_i,$$

sapendo che  $x \neq y$  possiamo dire che  $m > n$ . Possiamo scrivere  $y$  come

$$y = x_1 \dots x_n y_{n+1} \dots y_m.$$

Come stringa  $z$  scegliamo  $z = cx^R$  dove

$$c = \begin{cases} a & \text{se } y_{n+1} = b \\ b & \text{altrimenti} \end{cases}.$$

Se applichiamo questa stringa alle due che abbiamo a disposizione otteniamo

$$\begin{aligned}xz &= xcx^R \in \text{PAL} \\yz &= xy_{n+1} \dots y_m cx^R \notin \text{PAL}\end{aligned}$$

perché

- alla prima ho accodato sé stessa ma rovesciata con in mezzo un carattere qualsiasi, che però essendo in mezzo non rompe;
- alla seconda ho accodato  $cx^R$ , quindi il pezzo fino a  $y_{n+1}$  è tutto uguale, e proprio in  $y_{n+1}$  e  $c$  abbiamo la diversità.

Ma allora ogni classe di equivalenza ha un'unica stringa, ma allora per il teorema precedente il linguaggio PAL non è deterministico.

### 4.3. Ricorsione

Visto che i linguaggi DCFL ci consentono l'uso della **ricorsione** essi sono utili per definire i **linguaggi di programmazione**. Come gerarchia abbiamo

$$\text{LR}(k) \subseteq \text{DCFL} \subseteq \text{CFL},$$

con la classe  $\text{LR}(k)$  che indica degli oggetti molto tecnici, poco naturali, che sono usati nei **parser**. Se  $k = 1$  allora stiamo considerando direttamente i DCFL.

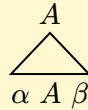
I PDA li possiamo immaginare come degli automi a stati finiti a cui abbiamo aggiunto una **pila**, ovvero una struttura dati che ci permette di implementare la **ricorsione**. Questo implica che i linguaggi CFL sono i linguaggi regolari a cui è stata aggiunta la ricorsione.



**Definizione 4.3.1** (Grammatiche self-embedding): Prendiamo una grammatica  $G = (V, \Sigma, P, S)$  context-free. Diciamo che  $G$  è **self-embedding** se

$$\exists A \in V \mid A \xRightarrow{*} \alpha A \beta \mid \alpha, \beta \in (\Sigma \cup V)^+.$$

In poche parole, esiste una variabile che ha un albero di derivazione in cui sulle foglie ho due stringhe diverse dalla parola vuota:



È importante che entrambe siano diverse dal vuoto:

- se è vuota  $\alpha$  abbiamo ricorsione all'inizio, che si può eliminare;
- se è vuota  $\beta$  abbiamo ricorsione in coda, che si può eliminare.

Se anche solo una è vuota non abbiamo più una **vera ricorsione**.

**Teorema 4.3.1:** Se  $G$  non è self-embedding allora  $L(G)$  è regolare.

Questo teorema ci dice che la  $G$  deve usare la ricorsione per generare un linguaggio CFL. Se non la utilizza e alcune cose possono essere eliminate allora collassiamo nei linguaggi regolari.

**Corollario 4.3.1.1:** Se  $L$  è un linguaggio CFL e non regolare allora ogni  $G$  per  $L$  è self-embedding.

## 4.4. Linguaggio di Dyck

Per finire, vediamo un risultato che secondo me è veramente fuori di testa.

**Definizione 4.4.1** (Linguaggio di Dyck): Definiamo l'alfabeto

$$\Omega_k = \{(1, (2, \dots (k, )_1, )_2, \dots, )_k\}$$

formato da  $k$  tipi di **parentesi**. Questo insieme contiene  $k$  parentesi aperte e le  $k$  parentesi chiuse corrispondenti, quindi  $|\Omega_k| = 2k$ .

Il **linguaggio di Dyck**

$$D_k \subseteq \Omega_k^*$$

è l'insieme delle parentesi bilanciate costruite sull'insieme  $\Omega_k$ .

Ora vediamo un teorema ideato dal nostro amico Chomsky e dal franco-tedesco Schutzenberger.

**Teorema 4.4.1** (Teorema di Chomsky-Schutzenberger): Dato  $L \subseteq \Sigma^*$  un CFL, allora:

- $\exists k > 0$  numero intero,
- $\exists$  morfismo  $h : \Omega_k \rightarrow \Sigma^*$ ,
- $\exists R \subseteq \Omega_k^*$  linguaggio regolare

tali che

$$L = h(D_k \cap R).$$

Questo è un **teorema di rappresentazione** ed è fuori di testa: scegliamo un insieme di parentesi, prendiamo il linguaggio di Dyck corrispondente, lo filtriamo con un linguaggio regolare definito sullo stesso linguaggio, applichiamo un morfismo che trasformi le parentesi in altri caratteri e otteniamo un CFL che abbiamo sotto mano.

**Esempio 4.4.1:** Definiamo il linguaggio

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

Possiamo considerare il blocco iniziale di  $a$  come se fosse un blocco di parentesi tonde aperte, mentre il blocco finale di  $b$  lo vediamo come se fosse un blocco di parentesi tonde chiuse.

Scegliamo quindi  $k = 1$  ottenendo l'insieme  $\Omega_k = \{(_1, )_1\}$  e definiamo il morfismo  $h$  tale che

$$(_1 \rightarrow a \qquad \qquad \qquad )_1 \rightarrow b$$

Tra tutte le stringhe di parentesi tonde bilanciate filtriamo le sequenze in cui le parentesi aperte si trovano prima delle parentesi chiuse, quindi scegliamo

$$R = (^*)^*.$$

**Esempio 4.4.2:** Se prendiamo  $L$  il linguaggio delle parentesi bilanciate, allora scegliamo l'identità come morfismo e come linguaggio regolare quello che fa passare tutto.

**Esempio 4.4.3:** Definiamo il linguaggio

$$L = \{ww^R \mid w \in \{a, b\}^*\}.$$

Possiamo vedere il fattore  $w$  come un blocco di parentesi aperte, che poi devono essere chiuse nella seconda metà con  $w^R$ . Scegliamo quindi  $k = 2$ , definendo un tipo di parentesi per le  $a$  e un tipo per le  $b$ . Il morfismo è tale che

$$(_1 \rightarrow a \qquad \qquad \qquad )_1 \rightarrow a \qquad \qquad \qquad (_2 \rightarrow b \qquad \qquad \qquad )_2 \rightarrow b$$

Come espressione regolare ci ispiriamo a quella di prima, quindi scegliamo

$$R = [(1 + (2)^*[1 + )_2]^*.$$

**Esempio 4.4.4:** Definiamo infine il linguaggio PAL delle stringhe palindrome di lunghezza anche dispari. Qua dobbiamo modificare leggermente la soluzione precedente

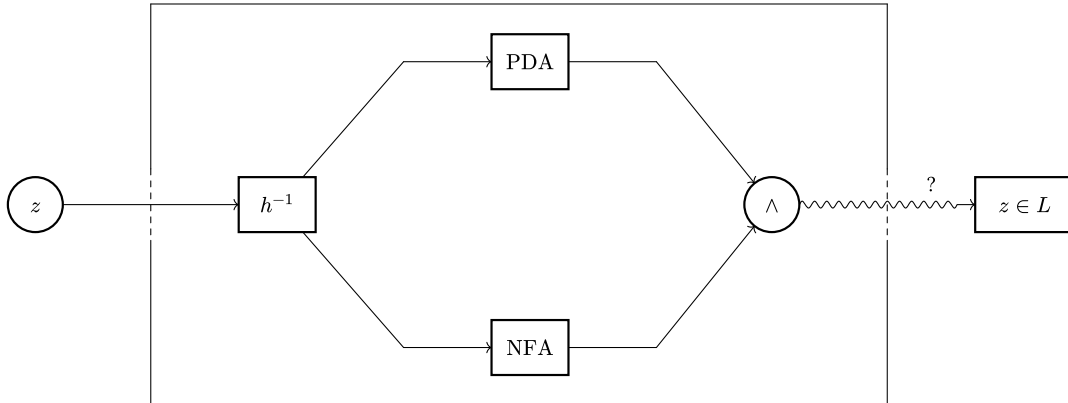
Scegliamo  $k = 4$  usando le parentesi definite prima, alle quali aggiungiamo due parentesi, che usiamo per codificare l'eventuale simbolo centrale, che può essere una  $a$  o una  $b$ , quindi:

$$(3 \rightarrow a \qquad )_3 \rightarrow \varepsilon \qquad (4 \rightarrow b \qquad )_4 \rightarrow \varepsilon$$

Come espressione regolare usiamo quella di prima, ma in mezzo possiamo avere una coppia di tipo 3, una coppia di tipo 4 oppure niente, quindi

$$R = [(1 + (2)^*[\varepsilon + (3)_3 + (4)_4][1 + )_2]^*.$$

Se non abbiamo a disposizione un riconoscitore per  $L$ , ma conosciamo tutto ciò che serve per costruirlo con il Teorema 4.4.1, ovvero conosciamo il morfismo  $h$ , il linguaggio di Dyck  $D_k$  e il linguaggio regolare  $R$ , possiamo **costruire un riconoscitore** per  $L$ .



Come vediamo, prima passiamo per il **morfismo inverso**  $h^{-1}$ , che viene anche detto **trasduttore**, ed è **non deterministico** perché il morfismo non è per forza iniettivo. Poi, l'input del trasduttore viene passato a due macchine:

- un **automa a pila** per  $D_k$ ;
- un **automa a stati finiti** per  $R$ .

Se entrambe le macchine rispondono **SI**, facendo un banale  $\wedge$ , allora  $z \in L$ .

Anche questo fatto è fuori di testa: mi danno un linguaggio  $L$  che non conosco, non solo lo posso definire come morfismo di un sottoinsieme di stringhe di parentesi bilanciate, ma posso anche costruire un riconoscitore per  $L$  usando gli stessi ingredienti che ho usato per definire il passaggio da parentesi a caratteri di  $L$ .

Possiamo quindi vedere i **riconoscitori dei CFL** come delle macchine di questo tipo.

## 5. Lezione 22 [23/05]

In questa lezione parleremo di troppe cose: toccheremo tutta la gerarchia di Chomsky, esclusi i linguaggi di tipo 0, considerando alfabeti particolari e macchine riconosctrici diverse dal solito. Andremo poi avanti anche con le grammatiche di tipo 2.

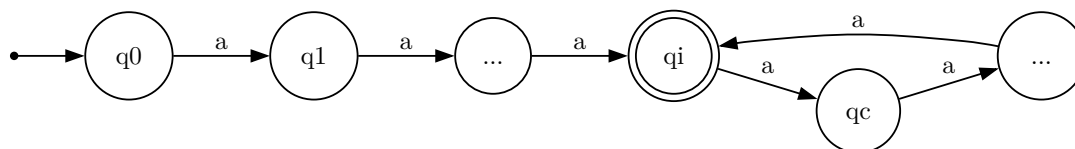
### 5.1. Alfabeti unari

Partiamo con gli **alfabeti unari**: questi sono alfabeti molto particolari formati da un solo carattere, ovvero sono nella forma

$$\Sigma = \{a\}.$$

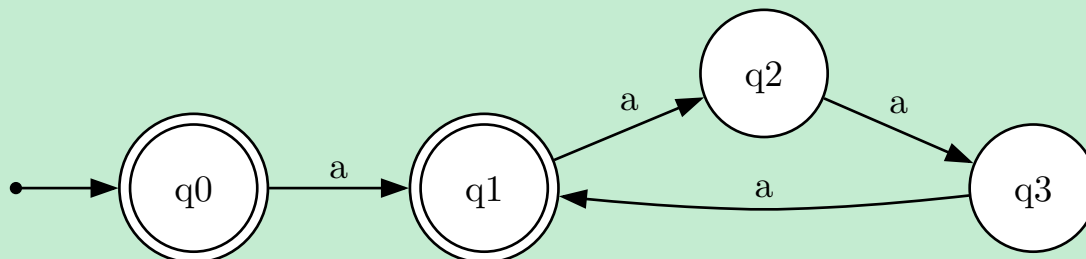
#### 5.1.1. Linguaggi regolari

Riprendiamo in mano, dopo tanto tempo, gli **automi a stati finiti**. Se rimaniamo nel caso deterministico, da ogni stato di un **DFA** può uscire un solo arco con una certa etichetta, ovvero non posso avere più di 2 archi uscenti con la stessa etichetta. Avendo ora un solo carattere in  $\Sigma$  quello che abbiamo è una sequenza (opzionale) di stati che prima o poi sfocia in un **ciclo** (opzionale).



Notiamo come l'informazione sulle parole diventa **informazione sulla lunghezza** di esse, visto che possiamo riconoscere delle stringhe che seguono un certo pattern di lunghezze.

**Esempio 5.1.1.1:** Vediamo un esempio di automa a stati finiti unario.



Con questo automa riconosciamo  $\varepsilon$ ,  $a$  e poi quest'ultima  $a$  cui aggiungiamo un numero di  $a$  uguali alla lunghezza del ciclo, ovvero

$$L = \{\varepsilon\} \cup \{a^{1+3k} \mid k \geq 0\} = \varepsilon + a(a^3)^*.$$

Dal punto di vista matematico, possiamo vedere questi automi come delle **successioni numeriche/aritmetiche**, ovvero delle successioni che hanno una parte iniziale e poi un periodo che viene ripetuto.

Nel caso di **NFA** invece abbiamo un grafo arbitrario, che per essere trasformato in DFA richiede meno dei  $2^n$  classici della **costruzione per sottoinsiemi**, ovvero ci costa

$$e^{n \ln(n)}.$$

Come vediamo, è una quantità **subesponenziale** ma comunque **superpolinomiale**. Inoltre, questo bound non può essere migliorato, è la soluzione ottimale.

**Esempio 5.1.1.2:** Definiamo tre linguaggi

$$L_1 = a^{28}(a^3)^*$$

$$L_2 = a^{11}(a^3)^*$$

$$L_3 = a^{37}(a^3)^*$$

Cosa aggiungono questi linguaggi al linguaggio  $L$  dell'Esempio 5.1.1.1?

Se consideriamo  $L_1$  notiamo che  $a^{28}$  può essere anche riconosciuto facendo uno step con una  $a$  e poi facendo 9 cicli da 3, quindi riusciamo a riconoscerlo anche con  $L$ . Possiamo fare un discorso praticamente simile con  $L_3$ . Ma allora questi due linguaggi non aggiungono niente.

Considerando invece  $L_2$  questo aggiunge qualcosa ad  $L$  perché riusciamo a riconoscere la stringa  $a^{10} = a(a^3)^3$  con  $L$  ma poi rimane una  $a$  fuori, che ci manda in  $q_2$  e quindi ci fa accettare di più, o comunque qualcosa di diverso rispetto a  $L$ .

Avremmo aggiunto altre informazioni considerando un linguaggio

$$L = a^k(a^3)^* \mid k \bmod 3 = 0.$$

Notiamo che, fissato un periodo, non possiamo unire tanti linguaggi, ma solo quelli che rimangono all'interno delle **classi di resto del periodo**.

### 5.1.2. Equivalenza tra linguaggi regolari e CFL

Vediamo ora come si comportano i CFL. Sia  $L$  un **CFL unario**, ovvero

$$L \subseteq a^*.$$

Applichiamo il **pumping lemma** a questo linguaggio. Prendiamo  $N$  la **costante del pumping lemma** per i CF per  $L$ . Questo ci dice che

$$\forall z \in L \mid |z| \geq N$$

noi possiamo decomporre  $z$  come  $z = uvwxy$  con:

1.  $|vwx| \leq N$ ;
2.  $vw \neq \varepsilon$ ;
3.  $\forall i \geq 0 \quad uv^iwx^iy \in L$ .

Le stringhe di  $L$  sono formate da sole  $a$ , quindi se scambiamo dei fattori nella stringa non lo notiamo. Modifichiamo l'ultima condizione del pumping lemma con

$$\forall i \geq 0 \quad uwy(vx)^i \in L.$$

Mettendo insieme le prime due condizioni possiamo dire che

$$1 \leq |vx| \leq N.$$

La stringa  $z$  la possiamo dividere in una **parte fissa** e in una **parte pompabile**, ovvero

$$|z| = |uwy| + |vx| = s_z + t_z.$$

Grazie alla terza condizione sappiamo che

$$\forall i \geq 0 \quad a^{s_z}(a^{t_z})^i \in L \implies a^{s_z}(a^{t_z})^* \in L.$$

Possiamo fare un'ulteriore divisione, stavolta sulle stringhe di  $L$ : infatti, possiamo scrivere  $L$  come unione di due insiemi  $L'$  e  $L''$  tali che

$$L = L' \cup L''.$$

Nell'insieme  $L'$  mettiamo tutte le stringhe che non fanno parte del pumping lemma, ovvero

$$L' = \{z \in L \mid |z| < N\}.$$

Nell'insieme  $L''$  mettiamo invece tutte le stringhe pompate, ovvero

$$L'' = \{z \in L \mid |z| \geq N\} \subseteq \bigcup_{z \in L''} a^{s_z}(a^{t_z})^*.$$

Analizziamo separatamente i due insiemi:

- $L'$  è un linguaggio **finito**, quindi lo possiamo riconoscere con un automa a stati finiti;
- $L''$  invece sembra un'unione infinita, ma abbiamo visto che il periodo  $t_z$  del pumping lemma è boundato con le classi di resto, ovvero

$$1 \leq t_z \leq N,$$

quindi questo linguaggio, che è unione finita di linguaggi regolari, è anch'esso **finito**.

Ma allora il linguaggio  $L$  è **regolare**.

**Teorema 5.1.2.1:** Sia  $L \subseteq a^*$  un CFL. Allora  $L$  è regolare.

Questo va d'accordo con quello che abbiamo fatto la lezione scorsa: i CFL hanno la **ricorsione**, ma se abbiamo un solo carattere non possiamo aprire e chiudere le parentesi, quindi collassiamo nei linguaggi regolari.

### 5.1.3. Teorema di Parikh

Vediamo, per finire, una serie di concetti un po' strani e che non dimostreremo.

**Definizione 5.1.3.1** (Immagine di Parikh sulle stringhe): Sia  $\Sigma = \{\sigma_1, \dots, \sigma_n\}$  un alfabeto. L'**immagine di Parikh** sulle stringhe è la funzione

$$\psi : \Sigma^* \longrightarrow \mathbb{N}^{|\Sigma|}$$

tale che

$$\psi(x) = (\#_{\sigma_1}(x), \dots, \#_{\sigma_n}(x)).$$

In poche parole, questa funzione conta le **occorrenze** di ogni lettera di  $\Sigma$  dentro la stringa  $x$ .

**Esempio 5.1.3.1:** Definiamo  $\Sigma = \{a, b\}$ . Data  $z = aababa$ , calcoliamo

$$\psi(z) = (4, 2).$$

Con l'immagine di Parikh sulle stringhe possiamo definire un insieme di queste immagini.

**Definizione 5.1.3.2** (Immagine di Parikh): Dato  $L$  un linguaggio generico, l'**immagine di Parikh** è l'insieme

$$\psi(L) = \{\psi(x) \mid x \in L\}.$$

In poche parole, l'immagine di Parikh è l'insieme di tutte le immagini di Parikh sulle stringhe di  $L$ .

**Esempio 5.1.3.2:** Vediamo tre linguaggi e le loro immagini di Parikh associate.

Linguaggio	Immagine di Parikh
$L = \{a^n b^n \mid n \geq 0\}$	$\{(n, n) \mid n \geq 0\}$
$L = a^* b^*$	$\{(i, j) \mid i, j \geq 0\}$
$L = (ab)^*$	$\{(n, n) \mid n \geq 0\}$

Notiamo come il primo e il terzo insieme sono uguali, anche se vengono generati da due linguaggi gerarchicamente diversi: il primo è un tipo 2, il terzo è un tipo 3.

L'ultima osservazione fatta genera quello che è il **teorema di Parikh**.

**Teorema 5.1.3.1** (Teorema di Parikh): Se  $L$  è un CFL allora  $\exists R$  regolare tale che

$$\psi(L) = \psi(R).$$

In poche parole, se non ci interessa l'**ordine** con cui scriviamo i caratteri di una stringa, allora i **linguaggi regolari** e i **CFL** sono la stessa cosa, collassano nella stessa classe.

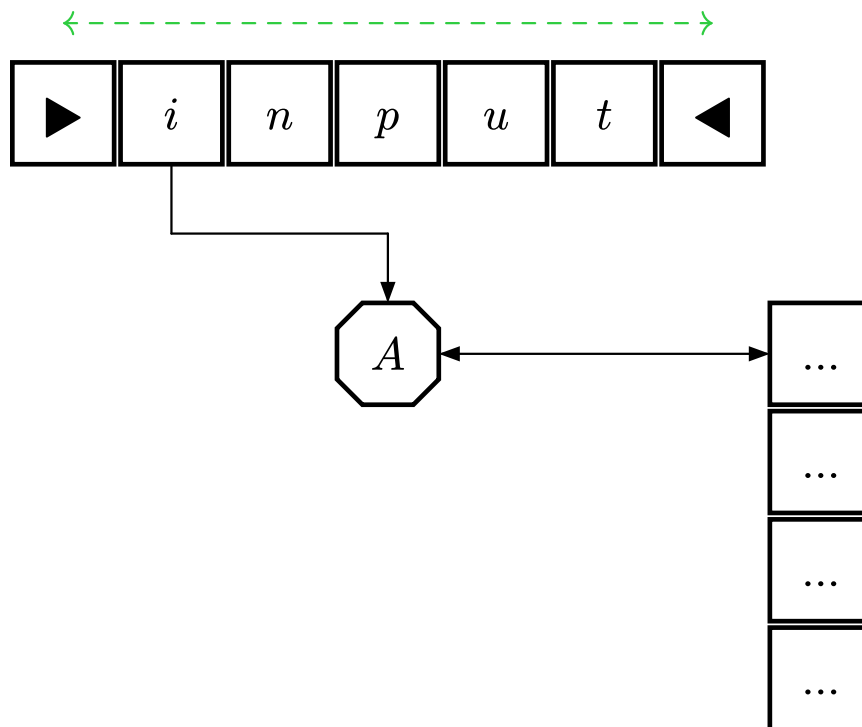
## 5.2. Automi a pila two-way

Modifichiamo un po' la macchina che stiamo usando da forse troppo tempo.

### 5.2.1. Definizione

Negli automi a stati finiti, il movimento **two-way** non aumentava la potenza computazionale del modello. Ma cosa succede negli **automi a pila two-way**?

Vediamo prima di tutto una rappresentazione del modello.



Come nei 2DFA, mettiamo degli **end marker** per marcare i bordi della stringa, perché ora la nostra testina di lettura può andare **avanti e indietro** sul nastro.

Con questo modello possiamo fare **molto di più** dei classici automi a pila.

### 5.2.2. Esempi

Vediamo una serie di linguaggi non CFL che riusciamo a riconoscere con questo modello.

**Esempio 5.2.2.1:** Definiamo il linguaggio

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

Questo linguaggio non è CFL perché una volta che controlliamo le  $b$  con le  $a$  perdiamo l'informazione su  $n$ . Con un **2DPDA** possiamo controllare le  $a$  con le  $b$ , poi tornare all'inizio delle  $b$  e controllare le  $b$  con le  $c$ .

**Esempio 5.2.2.2:** Definiamo il linguaggio

$$L = \{a^{2^n} \mid n \geq 0\}.$$

Con il **pumping lemma** avevamo mostrato che questo linguaggio non è CFL. Ora che abbiamo la definizione di sequenza algebrica, possiamo dire che questo linguaggio non è una **sequenza algebrica** perché le  $a$  si allontanano sempre di più tra loro.

Dobbiamo controllare se l'input è una potenza di 2: per fare ciò continuiamo a dividere per 2, verificando di avere sempre resto zero, salvo alla fine, dove abbiamo per forza resto 1.



Se  $k$  è la lunghezza dell'input, possiamo eseguire i seguenti passi:

1. leggiamo l'input per intero, e ogni due  $a$  carichiamo una lettera sulla pila, caricando in totale  $\frac{k}{2}$  caratteri. Con questa passata controlliamo se le  $a$  sono pari o dispari;
2. svuotiamo la pila, spostandoci di una posizione a sinistra ogni volta che togliamo un carattere. Con questa mezza passata ci troviamo, appunto, a metà della stringa, sul carattere in posizione  $\frac{k}{2}$ ;
3. ricominciamo dal primo punto fino a quando non rimaniamo con un carattere solo, che mi dà per forza resto 1.

Anche questo, come quello di prima, è un **2DPDA**.

**Esempio 5.2.2.3:** Definiamo il linguaggio

$$L = \{ww \mid w \in \{a, b\}^*\}.$$

Anche questo linguaggio non è CFL, e lo avevamo mostrato con uno dei quattro criteri della scorsa lezione, non mi ricordo quale in questo momento.

Come prima, carichiamo nella pila un carattere ogni due caratteri letti dell'input completo. Con questa prima passata controlliamo anche se il numero di caratteri è pari o dispari, e in quest'ultimo caso ci fermiamo e rifiutiamo. Spostiamoci poi in mezzo alla stringa scaricando la pila fino al carattere iniziale che avevamo anche prima.

Chiamiamo  $w$  la parte a sinistra della posizione nella quale ci troviamo ora. Carichiamo  $w$  dal centro verso l'inizio: stiamo leggendo  $w^R$ , che caricata sulla pila diventa  $w$ .

Spostiamoci di nuovo a metà della stringa, mettendo un separatore  $\#$  tra  $w$  e i caratteri che usiamo per spostarci. Ora che siamo a metà, togliamo  $\#$  dal congelatore e, con  $w$  sulla pila, possiamo controllare se la seconda parte è uguale a  $w$ .

Anche questo è un fantastico **2DPDA**.

**Esempio 5.2.2.4:** Non lo facciamo vedere, ma il linguaggio

$$L = \{ww^R \mid w \in \{a, b\}^*\}$$

ha un **2DPDA** che lo riconosce in maniera molto simile a quelle precedenti.

Come vediamo, questo modello è **molto potente**, talmente potente che nessuno sa quanto sia potente: infatti, tutti gli esempi visti sono stati risolti con un **2DPDA**, quindi anche da un **2NPDA** che fa partire una sola computazione alla volta, ma non sappiamo se

$$2NPDA \stackrel{?}{=} 2DPDA .$$

Inoltre, non si conosce la relazione che si ha con i linguaggi di tipo 1, che vediamo tra poco, ovvero non sappiamo se

$$2DPDA \stackrel{?}{=} CS .$$

### 5.3. Problemi di decisione dei CFL

Per finire questa lezione infinita, torniamo indietro ai linguaggi CFL e vediamo qualche **problema di decisione**. Per ora vedremo i problemi a cui sappiamo rispondere con quello che sappiamo, questo perché dei problemi di decisione richiedono conoscenze delle **macchine di Turing**, che per ora non abbiamo.

#### 5.3.1. Appartenenza

Dato  $L$  un CFL e una stringa  $x \in \Sigma^*$ , ci chiediamo se  $x \in L$ .

Questo è molto facile: sappiamo che i CFL sono **decidibili** perché lo avevamo mostrato per i linguaggi di tipo 1. Come complessità come siamo messi?

Sia  $n = |x|$ . Esistono algoritmi semplici che permettono di decidere in tempo

$$T(n) = O(n^3).$$

L'**algoritmo di Valiant**, quasi incomprensibile, riconduce il problema di riconoscimento a quello di prodotto tra matrici  $n \times n$ , che con l'algoritmo di Strassen possiamo risolvere in tempo

$$T(n) = O(n^{\log_2(7)}) = O(n^{2.81\dots}).$$

L'algoritmo di Strassen in realtà poi è stato superato da altri algoritmi ben più sofisticati, che impiegano tempo quasi quadratico, ovvero

$$T(n) = O(n^{2.3\dots}).$$

Una domanda aperta si chiede se riusciamo ad abbassare questo bound al livello quadratico, e questo sarebbe molto comodo: infatti, negli algoritmi di parsing avere degli algoritmi quadratici è apprezzabile, e infatti spesso di considerano sottoclassi per avvicinarsi a complessità lineari.

#### 5.3.2. Linguaggio vuoto e infinito

Sia  $L$  un CFL, ci chiediamo se  $L \neq \emptyset$  oppure se  $|L| = \infty$ .

Vediamo un teorema praticamente identico a uno che avevamo già visto.

**Teorema 5.3.2.1:** Sia  $L \subseteq \Sigma^*$  un CFL, e sia  $N$  la costante del pumping lemma per  $L$ . Allora:

1.  $L \neq \emptyset \iff \exists z \in L \mid |z| < N$ ;
2.  $|L| = \infty \iff \exists z \in L \mid N \leq |z| < 2N$ .

Gli algoritmi per verificare la non vuotezza o l'infinità non sono molto efficienti: infatti, prima di tutto bisogna trovare  $N$ , e se ho una grammatica è facile (basta passare in tempo lineare per la FN di Chomsky), ma se non ce l'abbiamo è un po' una palla. Poi dobbiamo provare tutte le stringhe fino alla costante, che sono  $2^N$ , e con questo rispondiamo alla non vuotezza. Per l'infinità è ancora peggio.

Si possono implementare delle tecniche che lavorano sul **grafo delle produzioni**, ma sono molto avanzate e (penso) difficili da utilizzare.

#### 5.3.3. Universalità

Dato  $L$  un CFL, vogliamo sapere se  $L = \Sigma^*$ , ovvero vogliamo sapere se siamo in grado di generare tutte le stringhe su un certo alfabeto.

Nei linguaggi regolari passavamo per il complemento per vedere se il linguaggio era vuoto, ma nei CFL **non abbiamo il complemento**, quindi non lo possiamo utilizzare.

Infatti, questo problema **non si può decidere**: non esistono algoritmi che stabiliscono se un PDA riesce a riconoscere tutte le stringhe, o se una grammatica riesce a generare tutte le stringhe.