

Dato in input qualcosa ad una grammatica, viene generato, per quel input, un albero sintattico (Esempio: DOM di un File HTML)

Grammatiche e Linguaggi Liberi dal Contesto

- Abbiamo visto che molti linguaggi non sono regolari. Consideriamo allora classi piu' grandi di linguaggi.
- *Linguaggi Liberi dal Contesto* (CFL = Context-Free Languages) sono stati usati nello studio dei linguaggi naturali dal 1950, e nello studio dei (generatori di) compilatori dal 1960.
- Le *grammatiche libere dal contesto* (CFG = Context-Free Grammars) sono la base della sintassi BNF (Backus-Naur-Form), usate per i linguaggi di programmazione.
- Oggi i CFL sono importanti anche per XML.

Questi linguaggi vengono espressi tramite grammatiche libere dal contesto ed automi a pila

Studieremo: CFG, i linguaggi che generano, gli alberi sintattici, gli automi a pila, e le proprieta' di chiusura dei CFL.

Le grammatiche sono simili a quella della grammatica italiana.
Esempio: parto da una espressione, dalla espressione ho un soggetto, verbo e complemento, etc..

Esempio informale di CFG

Consideriamo $\underline{L_{pal}} = \{w \in \Sigma^* : w = w^R\}$ Linguaggio delle stringhe palindrome

Per esempio: $\underline{\text{otto} \in L_{pal}, \text{ara} \in L_{pal}}$.

Sia $\Sigma = \{0, 1\}$ e supponiamo che $\underline{L_{pal}}$ sia regolare.

Sia n dato dal pumping lemma. Allora $0^n 1 0^n \in L_{pal}$. Nel leggere 0^n il FA deve passare per un loop. Se omettiamo il loop, contraddizione. (Essendo la lunghezza di $xy \leq n$, il FA legge 0^n e va in ciclo, quindi non memorizza le stesse quantità di 0^n a sinistra e destra (perché se $k=0$, di y^k , tolgo degli zeri a sinistra))

Definiamo $\underline{L_{pal}}$ induttivamente:

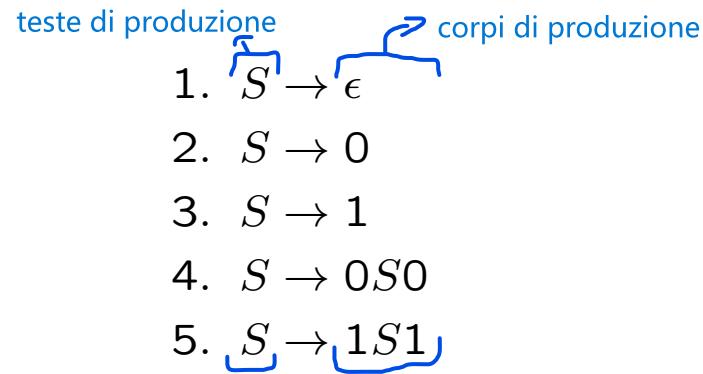
Base: $\epsilon, 0, 1$ sono palindromi.

Induzione: Se w è una palindrome, anche $0w0$ e $1w1$ lo sono.

Nessun'altra stringa è una palindrome.

L_{pal} non
regolare, dato
l'assurdo del
pumping
lemma

Le CFG sono un modo formale per definizioni come quella per L_{pal} .



0 e 1 sono *terminali*

S e' una *categoria sintattica* (o, piu' tecnicamente, *variabile*)

S e' in questa grammatica anche la categoria sintattica *iniziale*.

1–5 sono *produzioni* (o *regole*) (non esiste ordine nelle produzioni (é un insieme di produzioni))

Definizione formale di CFG

Una grammatica libera dal contesto e' una quadrupla

$$G = (V, T, P, S)$$

dove

V e' un insieme finito di *variabili* (o *non-terminali*).

T e' un insieme finito di *terminali*.

P e' un insieme finito di *produzioni* della forma $A \rightarrow \alpha$, dove A e' una variabile e $\alpha \in (V \cup T)^*$

A = testa di produz.
alfa = corpo di produz

S e' una variabile distinta chiamata *variabile iniziale*.

Esempio: $G_{pal} = (\{S\}, \{0, 1\}, P, S)$, dove $P =$
 $\{S \rightarrow \epsilon, S \rightarrow 0, S \rightarrow 1, S \rightarrow 0S0, S \rightarrow 1S1\}$.

A volte raggruppiamo le produzioni con la stessa testa: $P =$
 $\{S \rightarrow \epsilon|0|1|0S0|1S1\}$.

Esempio: espressioni (semplici) in un tipico linguaggio di programmazione. Gli operatori sono + e *, e gli operandi sono identificatori, cioe' stringhe in $L((a+b)(a+b+0+1)^*)$.

Le espressioni sono definite dalla grammatica

$$G = (\{E, I\}, T, P, E)$$

dove $T = \{+, *, (,), a, b, 0, 1\}$ e P e' il seguente insieme di produzioni:

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$

5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

Le derivazioni mi permettono di usare le grammatiche per generare le stringhe appartenenti al linguaggio della grammatica

Derivazioni usando le grammatiche

Sia $G = (V, T, P, S)$ una CFG, $A \in V$,
 $\{\alpha, \beta\} \subset (V \cup T)^*$, e $A \rightarrow \gamma \in P$.

Allora scriviamo

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

Data una grammatica, trasformo la variabile A in una produzione delle P disponibili

o, se e' ovvia la G ,

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

e diciamo che da $\alpha A \beta$ si deriva $\alpha \gamma \beta$.

Definiamo $\xrightarrow{*}$ la chiusura riflessiva e transitiva di \Rightarrow , cioe':

Base: Sia $\alpha \in (V \cup T)^*$. Allora $\alpha \xrightarrow{*} \alpha$.

Induzione: Se $\alpha \xrightarrow{*} \beta$, e $\beta \Rightarrow \gamma$, allora $\alpha \xrightarrow{*} \gamma$.

Applica tot
trasformazioni
per arrivare alla
stringa da
generare

Il Contesto (ciò che sta attorno alla variabile che sto sostituendo, cioè quella sottolineata) rimane invariato e sostituisco la variabile selezionata

Esempio: Derivazione di $a * (a + b00)$ da E nella grammatica delle espressioni:

$$\begin{aligned} E &\xrightarrow{3} \underline{E} * E \xrightarrow{1} \underline{I} * E \xrightarrow{5} a * \underline{E} \xrightarrow{4} a * (\underline{E}) \xrightarrow{2} \\ a * (\underline{E} + E) &\xrightarrow{1} a * (\underline{I} + E) \xrightarrow{5} a * (a + \underline{E}) \xrightarrow{7} a * (a + \underline{I}) \xrightarrow{9} \\ a * (a + \underline{I}0) &\xrightarrow{9} a * (a + \underline{I}00) \xrightarrow{6} a * (a + b00) \end{aligned}$$

Sostituisco la variabile, con una delle produzioni, con il corpo della produzione scelta. Sostituisco indipendentemente dal contesto.

Nota: ad ogni passo potremmo avere varie regole tra cui scegliere, ad esempio

- ① $I * E \Rightarrow a * E \Rightarrow a * (E)$, oppure
- ② $I * E \Rightarrow I * (E) \Rightarrow a * (E)$.

Nota: non tutte le scelte portano a derivazioni di una particolare stringa, per esempio

$$E \Rightarrow E + E$$

non ci fa derivare $a * (a + b00)$.

Le grammatiche libere dal contesto sono meno espansive di quelle dipendenti dal contesto.

8

Linguaggio di una grammatica: insieme delle stringhe che può generare.
Linguaggio di un automa: insieme delle stringhe che può riconoscere.

posso anche effettuare delle derivazioni miste (una volta prendo la variabile più a sinistra, una volta prendo la variabile più a destra)

Derivazioni a sinistra e a destra

Left Most Derivation

Derivazione a sinistra \xrightarrow{lm} : rimpiazza sempre la variabile piu' a sinistra con il corpo di una delle sue regole.

Right Most Derivation

Derivazione a destra \xrightarrow{rm} : rimpiazza sempre la variabile piu' a destra con il corpo di una delle sue regole.

Der. a sinistra: quella del lucido precedente.

A destra:

$$E \xrightarrow{rm} E * E \xrightarrow{rm}$$

$$E * (E) \xrightarrow{rm} E * (E + E) \xrightarrow{rm} E * (E + I) \xrightarrow{rm} E * (E + IO)$$

$$\xrightarrow{rm} E * (E + IO0) \xrightarrow{rm} E * (E + b00) \xrightarrow{rm} E * (I + b00)$$

$$\xrightarrow{rm} E * (a + b00) \xrightarrow{rm} I * (a + b00) \xrightarrow{rm} a * (a + b00)$$

Possiamo concludere che $E \xrightarrow[rm]{*} a * (a + b00)$

Il linguaggio di una grammatica

Se $G(V, T, P, S)$ e' una CFG, allora il *linguaggio di G* e'

$$L(G) = \{w \in T^* : S \xrightarrow[G]{*} w\}$$

cioe' l'insieme delle stringhe su T^* derivabili dal simbolo iniziale.

Se G e' una CFG, chiameremo $L(G)$ un
linguaggio libero dal contesto.

Esempio: $L(G_{pal})$ e' un linguaggio libero dal contesto.

Non significa che tante derivazioni = tanti alberi sintattici

Alberi sintattici

- Se $w \in L(G)$, per una CFG, allora w ha un *albero sintattico*, che ci dice la **struttura (sintattica)** di w .
- w potrebbe essere un programma, una query SQL, un documento XML, ...
- Gli alberi sintattici sono una **rappresentazione alternativa alle derivazioni**.
- Ci possono essere diversi alberi sintattici per la stessa stringa.
- Idealmente ci dovrebbe essere **solo un albero sintattico** (la "vera" struttura), cioè la CFG dovrebbe essere **non ambigua**.
- Sfortunatamente, non sempre possiamo rimuovere l'ambiguità.
(si rimuove l'ambiguità, trovando una grammatica equivalente a quella ambigua)

Costruzione di un albero sintattico

Sia $G = (V, T, P, S)$ una CFG. Un albero e' un *albero sintattico* per G se:

cioè nodi e radici (non foglie)

1. Ogni **nodo interno** è etichettato con una **variabile** in V .
2. Ogni foglia è etichettata con un simbolo in $V \cup T \cup \{\epsilon\}$. Ogni foglia etichettata con ϵ è l'unico figlio del suo genitore.
3. Se un nodo interno è etichettato A , e i suoi figli (da sinistra a destra) sono etichettati



Significa che se un nodo interno ha dei figli, i suoi figli insieme rappresentano il corpo di una produzione

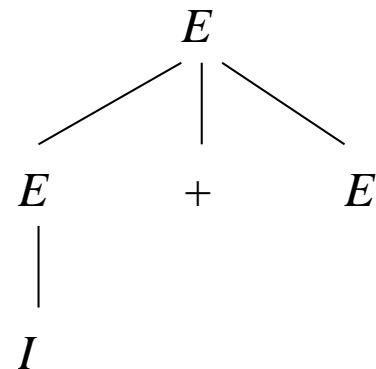
$$X_1, X_2, \dots, X_k,$$

allora $A \rightarrow X_1 X_2 \dots X_k \in P$.

Esempio: nella grammatica

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
- ⋮

il seguente e' un albero sintattico:

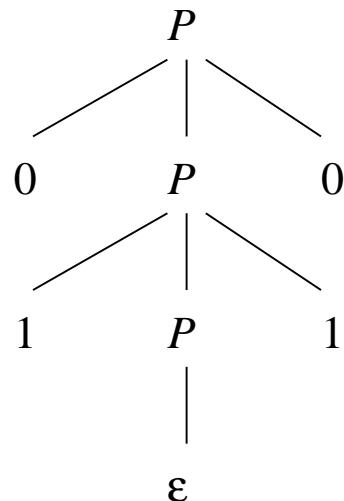


Questo albero sintattico mostra la derivazione $E \xrightarrow{*} I + E$

Esempio: nella grammatica

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

il seguente e' un albero sintattico:



Mostra la derivazione $P \xrightarrow{*} 0110$.

Il prodotto di un albero sintattico

(cioè ciò che produce un albero)

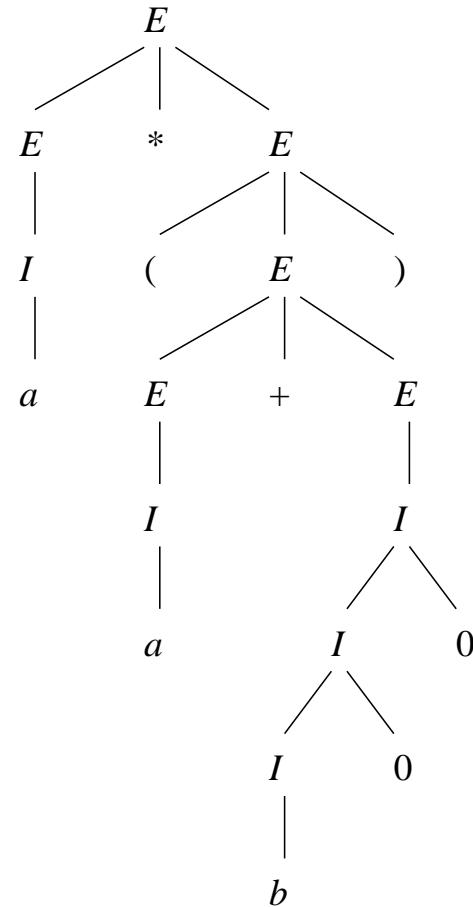
Il **prodotto** di un albero sintattico e' la **stringa di foglie da sinistra a destra**.

Importanti sono quegli alberi sintattici dove:

1. Il prodotto e' **una stringa terminale**.
2. La radice e' etichettata dal **simbolo iniziale**.

L'insieme dei **prodotti di questi alberi sintattici** e' il **linguaggio della grammatica**.

Esempio:



Il prodotto e' $a * (a + b00)$.

Devo verificare che 1,2,3 sono equivalenti (cioè 1 implica 2 che implica 3 che implica 1)

Sia $G = (V, T, P, S)$ una CFG, e $A \in V$. I seguenti sono equivalenti:

1. $A \xrightarrow{*} w$ (derivazione qualsiasi per w)

2. $A \xrightarrow{lm} w$, e $A \xrightarrow{rm} w$ (derivazione a sinistra e destra di w)

3. C'è un albero sintattico di G con radice A e prodotto w .

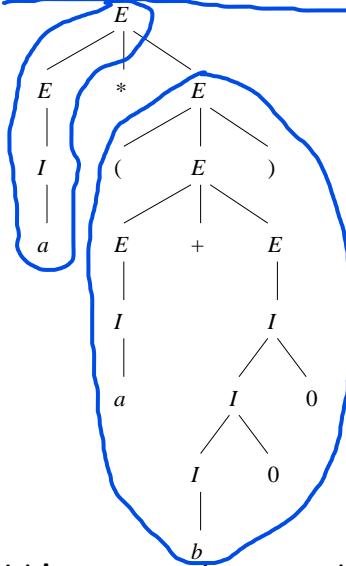
Per provare l'equivalenza, usiamo il seguente piano:

1. • Dagli alberi alle derivazioni a sinistra (destra): visito l'albero da sinistra a destra (da destra a sinistra) $\textcircled{3} \Rightarrow \textcircled{2}$
2. • Una derivazione sinistra (o destra) è anche una derivazione $\textcircled{2} \Rightarrow \textcircled{1}$
3. • Leggendo la derivazione costruisco l'albero $\textcircled{1} \Rightarrow \textcircled{3}$



Dato l'albero, ho un'unica derivazione canonica sinistra e ho un'unica derivazione canonica destra

Esempio: Costruiamo la derivazione a sinistra per l'albero (Da albero costruisco derivazione sx)



Supponiamo di aver induttivamente costruito la deriv. a sinistra

$$E \xrightarrow{lm} I \xrightarrow{lm} a$$

corrispondente al sottoalbero piu' a sinistra, e la deriv. a sinistra

$$E \xrightarrow{lm} (E) \xrightarrow{lm} (E + E) \xrightarrow{lm} (I + E) \xrightarrow{lm} (a + E) \xrightarrow{lm}$$

$$(a + I) \xrightarrow{lm} (a + I0) \xrightarrow{lm} (a + I00) \xrightarrow{lm} (a + b00)$$

corrispondente al sottoalbero piu' a destra.

Per la derivazione corrispondente all'intero albero, iniziamo con
 $E \Rightarrow E * E$ e espandiamo la prima E con la prima derivazione e la
seconda E con la seconda derivazione:

$$E \xrightarrow{lm} E * E \xrightarrow{lm}$$

$$I * E \xrightarrow{lm}$$

$$a * E \xrightarrow{lm}$$

$$a * (E) \xrightarrow{lm}$$

$$a * (E + E) \xrightarrow{lm}$$

$$a * (I + E) \xrightarrow{lm}$$

$$a * (a + E) \xrightarrow{lm}$$

$$a * (a + I) \xrightarrow{lm}$$

$$a * (a + I0) \xrightarrow{lm}$$

$$a * (a + I00) \xrightarrow{lm}$$

$$a * (a + b00)$$

Ambiguità in Grammatiche e Linguaggi

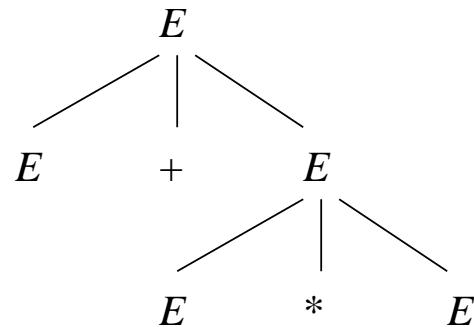
Nella grammatica

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
- ...

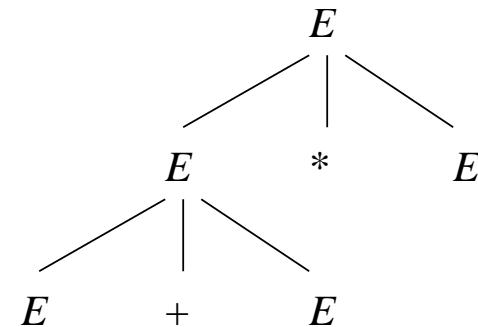
$E + E * E$ ha due derivazioni:

$$E \Rightarrow E + E \Rightarrow E + E * E \quad \text{e} \quad E \Rightarrow E * E \Rightarrow E + E * E$$

Questo ci da' due alberi sintattici:



(a)



(b)

L'esistenza di varie *derivazioni* di per se non e' pericolosa, e'
l'esistenza di vari alberi sintattici che rovina la grammatica.

(é un problema perché il parser,
ad esempio nella creazione del
DOM di un file HTML, non sa
quale albero scegliere e l'albero
che sceglie ogni volta non deve
cambiare ogni volta)

Esempio: Nella stessa grammatica

5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

la stringa $a + b$ ha varie derivazioni:

$$E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$$

e

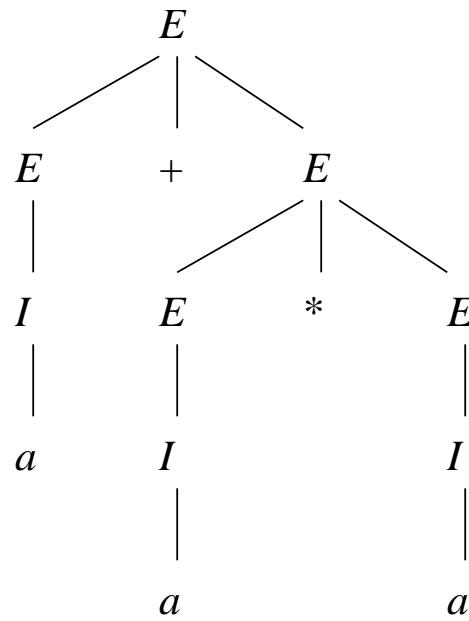
$$E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$$

Pero' il loro albero sintattico e' lo stesso (anche per le altre possibili derivazioni di $a + b$): la struttura di $a + b$ e' quindi non ambigua.

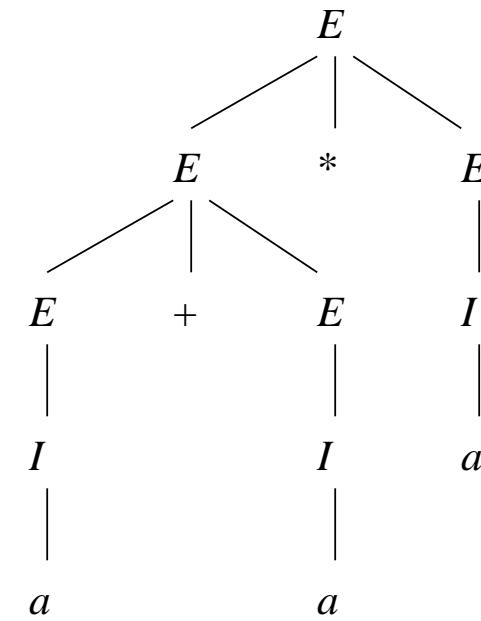
Definizione: Sia $G = (V, T, P, S)$ una CFG. Diciamo che G e' *ambigua* se esiste una stringa in T^* che ha piu' di un albero sintattico.

Se ogni stringa in $L(G)$ ha un unico albero sintattico, G e' detta *non-ambigua*.

Esempio: La stringa terminale $a + a * a$ ha due alberi sintattici:



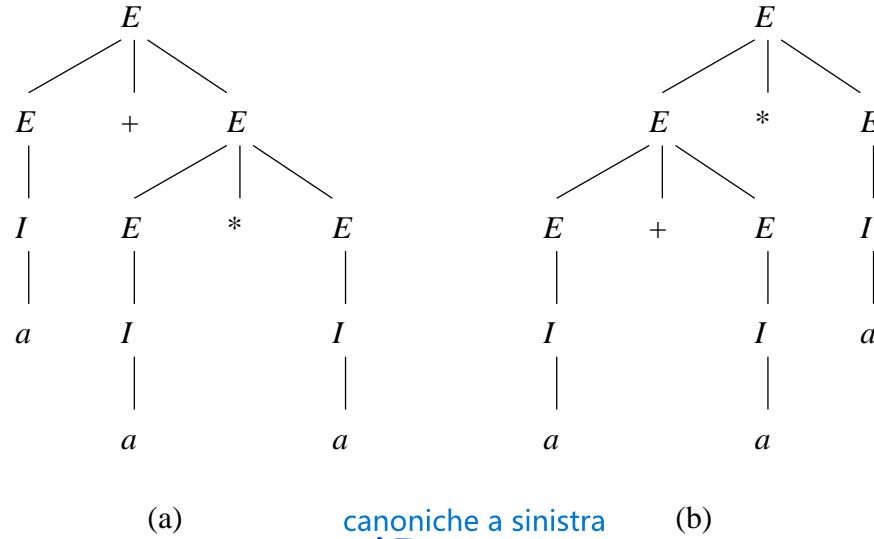
(a)



(b)

Derivazioni a sinistra e ambiguità'

I due alberi sintattici per $a + a * a$



danno luogo a due derivazioni:

$$\begin{aligned}
 E &\xrightarrow{lm} E + E \xrightarrow{lm} I + E \xrightarrow{lm} a + E \xrightarrow{lm} a + E * E \\
 &\xrightarrow{lm} a + I * E \xrightarrow{lm} a + a * E \xrightarrow{lm} a + a * I \xrightarrow{lm} a + a * a
 \end{aligned}$$

e

$$\begin{aligned}
 E &\xrightarrow{lm} E * E \xrightarrow{lm} E + E * E \xrightarrow{lm} I + E * E \xrightarrow{lm} a + E * E \\
 &\xrightarrow{lm} a + I * E \xrightarrow{lm} a + a * E \xrightarrow{lm} a + a * I \xrightarrow{lm} a + a * a
 \end{aligned}$$

In generale:

- Ad un albero sintattico corrispondono molte derivazioni, ma
- ad ogni (diverso) albero sintattico corrisponde un'unica (diversa) derivazione *a sinistra*.
- ad ogni (diverso) albero sintattico corrisponde un'unica (diversa) derivazione *a destra*.

Teorema 5.29: Data una CFG G , una stringa terminale w ha due distinti alberi sintattici se e solo se w ha due distinte derivazioni a sinistra dal simbolo iniziale.

Rimuovere l'ambiguità dalle grammatiche

Buone notizie: a volte possiamo rimuovere l'ambiguità

Cattive notizie: non c'e' nessun algoritmo per farlo in modo sistematico

Ancora cattive notizie: alcuni CFL hanno solo CFG ambigue

Studiamo la grammatica

$$\begin{aligned} E &\rightarrow I \mid E + E \mid E * E \mid (E) \\ I &\rightarrow a \mid b \mid Ia \mid Ib \mid IO \mid I1 \end{aligned}$$

Bisogna modificarla in modo da stabilire:

1. Chi ha precedenza tra * e +

2. Come si raggruppano sequenze di uno stesso operatore:

$E + E + E$ e' inteso come $E + (E + E)$ o come $(E + E) + E$?

Il problema nasce quando nella stringa non ci sono parentesi e quindi bisogna definire:
- chi ha precedenza tra i vari operatori
- chi ha precedenza tra stessi operatori

Associatività degli operatori: porto associatività a sinistra, quindi posso avere lo stesso operatore a sinistra ma non a destra
(Esempio: posso avere trasformazioni con il + solo a sinistra)

25

forzo la variabile di destra a cambiare variabile per andare in un livello intermedio (es: T) per evitare di avere stesso operatore a destra

Si risolve creando dei livelli intermedi nella grammatica per dare ordine di priorità.
Ad esempio: forzo ad avere + nella parte alta dell'albero e * nella parte in basso

Cioè vado in profondità sul * anziché sul +

Soluzione: Introduciamo una gerarchia di variabili che stabilisca un ordine di precedenza tra gli operatori

1. *espressioni E*: composizioni di uno o più *termini T* tramite $+$
2. *termini T*: composizioni di uno o più *fattori F* tramite $*$
3. *fattori F*:
 - (a) *identificatori I*
 - (b) *espressioni E* racchiuse tra parentesi



I termini *T* non possono generare $+$ che siano fuori da parentesi: questa gerarchia stabilisce che $*$ **ha precedenza rispetto a $+$** .

Esempio: l'unico modo di generare $a+a*a$, visto in precedenza, e' considerando $a*a$ come un termine *T* (albero sintattico di sinistra)

Formalmente:

1. $E \rightarrow T \mid E + E$
2. $T \rightarrow F \mid T * T$
3. $F \rightarrow I \mid (E)$
4. $I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

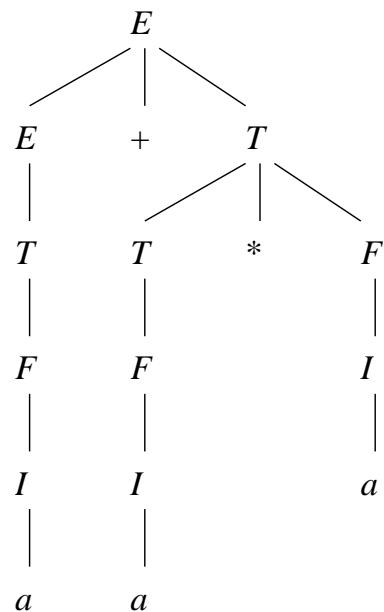
Questa grammatica e' non ambigua? **NO**

Posso ancora generare, sia a sx sia a dx, molti operatori dello stesso tipo

No! Dobbiamo anche imporre un ordine per raggruppamento operatori allo stesso livello. Es con associativita' a sinistra:

1. $E \rightarrow T \mid E + T$
2. $T \rightarrow F \mid T * F$
3. $F \rightarrow I \mid (E)$
4. $I \rightarrow a \mid b \mid Ia \mid Ib \mid IO \mid I1$

Grammatica non ambigua, es. unico albero sintattico di $a + a * a$ è



L'Ambiguità può essere
caratteristica del linguaggio

Ambiguità' inerente

Quindi, se cambio grammatica
con una equivalente ho sempre
un linguaggio ambiguo

Un CFL L è *inerentemente ambiguo* se tutte le grammatiche per L sono ambigue.

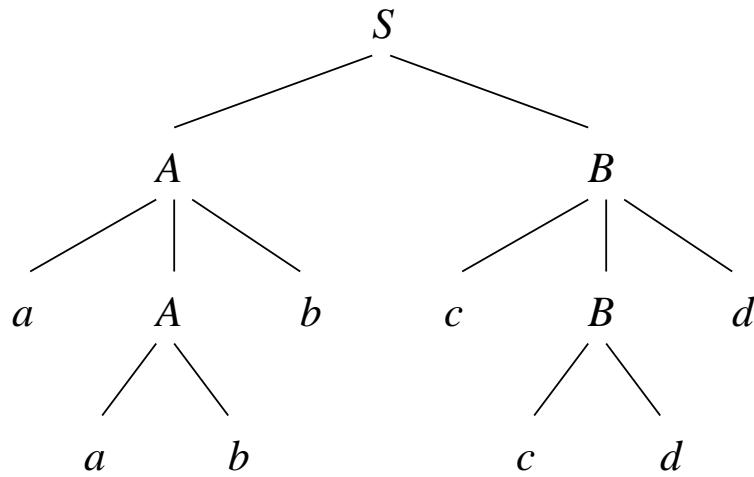
Esempio: Consideriamo $L =$

$$\{a^n b^n c^m d^m : n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n : n \geq 1, m \geq 1\}.$$

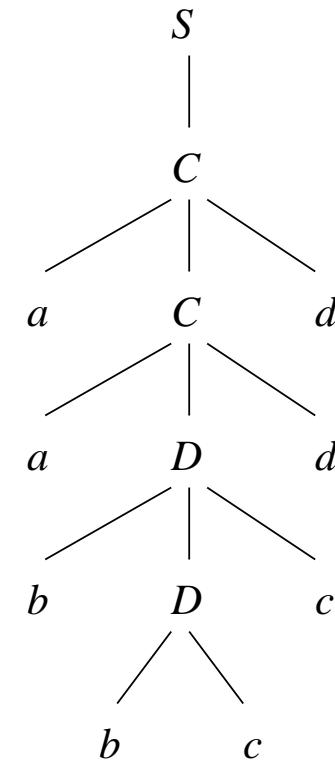
Una grammatica per L è'

$$\begin{aligned} S &\rightarrow AB \mid C \\ A &\rightarrow aAb \mid ab \\ B &\rightarrow cBd \mid cd \\ C &\rightarrow aCd \mid aDd \\ D &\rightarrow bDc \mid bc \end{aligned}$$

Guardiamo la struttura sintattica della stringa $aabbccdd$.



(a)



(b)

Vediamo che ci sono due derivazioni a sinistra:

$$S \xrightarrow{lm} AB \xrightarrow{lm} aAbB \xrightarrow{lm} aabbB \xrightarrow{lm} aabbcBd \xrightarrow{lm} aabbccdd$$

e

$$S \xrightarrow{lm} C \xrightarrow{lm} aCd \xrightarrow{lm} aaDdd \xrightarrow{lm} aabDcdd \xrightarrow{lm} aabbccdd$$

Puo' essere provato che ogni grammatica per L si comporta come questa. Il linguaggio L e' quindi inherentemente ambiguo.

PDA = Push Down Automaton

Automi a pila

Viene scelta la pila a causa dei suoi metodi di inserimento/rimozione. Il primo che aggiungo, é il primo che rimuovo (FIFO)

Un automa a pila (PDA) e' in pratica un ϵ -NFA con una pila.

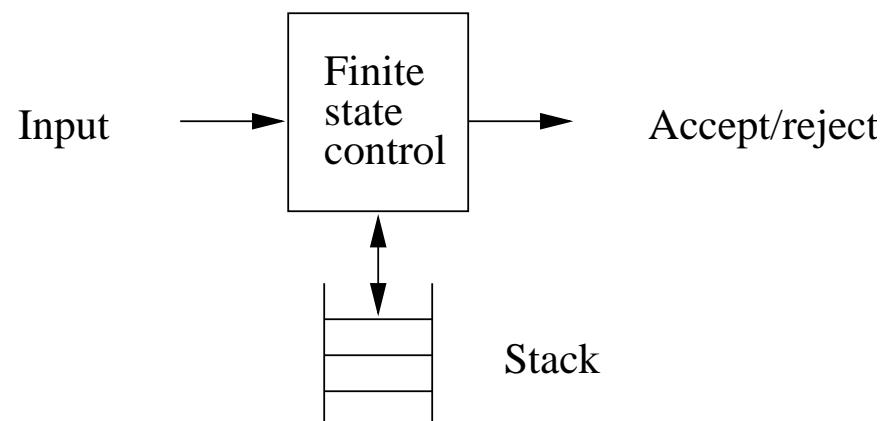
In una transizione un PDA:

Come prima
in epsilon NFA

1. Consuma un simbolo di input o esegue una transizione ϵ .
2. Va in un nuovo stato (o rimane dove e').
3. Rimpiazza il top della pila con una stringa
(consuma il carattere in cima, e mette al suo posto una stringa, eventualmente vuota o uguale al carattere consumato lasciando quindi la pila inalterata)

Dipende da cosa c'è in cima allo stack (perché ciò verrà rimpiazzato con una nuova stringa):

- Aggiungo stringa vuota: Faccio POP e basta
- Aggiungo una stessa stringa: Faccio PUSH e basta



Gli automi a stati finiti senza pila non riconoscono le stringhe palindrome perché non hanno memoria (non viene rispettato il pumping lemma), non ricordano ciò che è passato di w nell'automa.

Gli automi a pila usano il non determinismo per percorrere qualsiasi strada, anche quelle bloccanti.

Esempio: Consideriamo

Linguaggio di stringhe palindrome con lunghezza pari

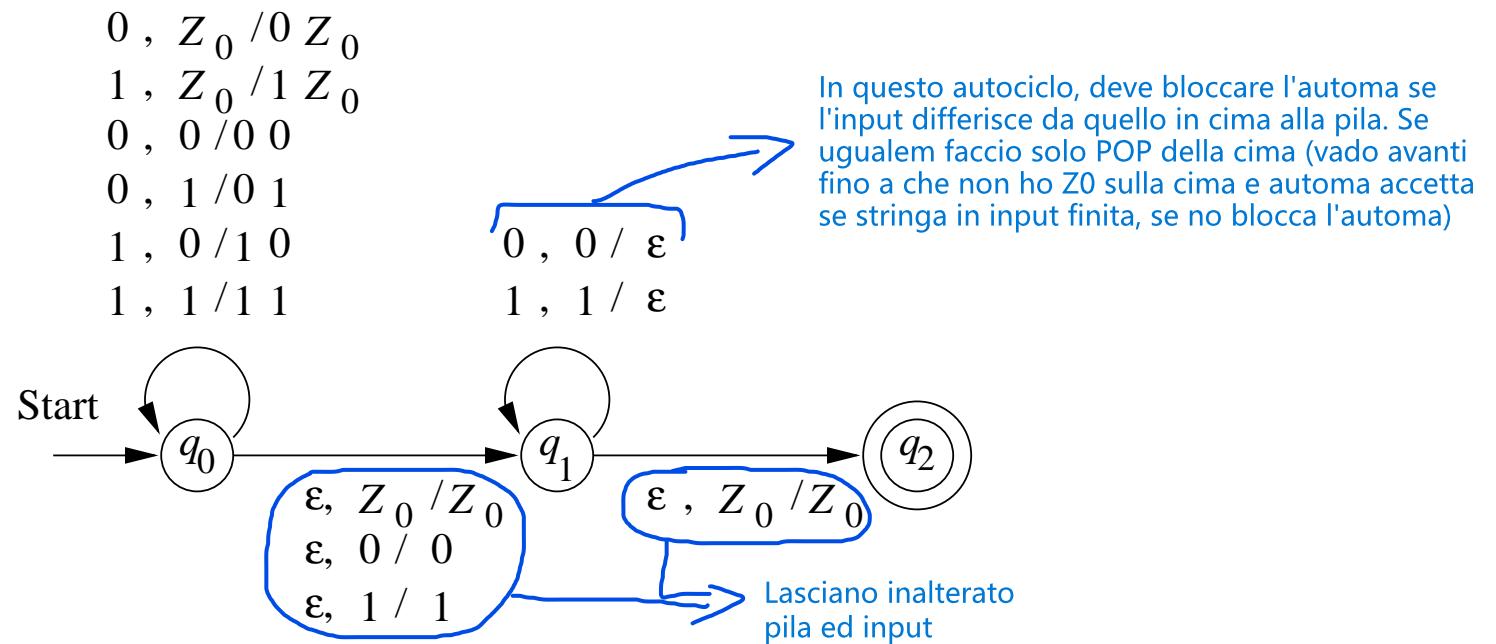
$$L_{wwr} = \{ww^R : w \in \{0, 1\}^*\},$$

con "grammatica" $P \rightarrow 0P0, P \rightarrow 1P1, P \rightarrow \epsilon$. Un PDA per L_{wwr} ha tre stati, e funziona come segue:

1. Legge w un simbolo alla volta, rimanendo nello stato q_0 , e aggiungendo il simbolo di input alla pila.
2. Decide non deterministicamente che sta nel mezzo di ww^R e va nello stato q_1 .
3. Legge w^R un simbolo alla volta e lo paragona col simbolo al top della pila: Se sono uguali, fa un pop della pila, e rimane nello stato q_1 . Se non sono uguali, si blocca.
4. Se la pila non ha più simboli (0 o 1), va nello stato q_2 e accetta.

In generale, le transizioni riscrivono la pila:
eliminando il primo elemento in cima sulla pila e
riscrivendo l'elemento eliminato + input

Il PDA per L_{wwr} come diagramma di transizione:



Come viene riconosciuta una stringa palindroma? Appena arrivo a metà della stringa, vado in q_1 , leggo la seconda parte della stringa ed alla fine vado in q_2 .

Definizione formale di PDA

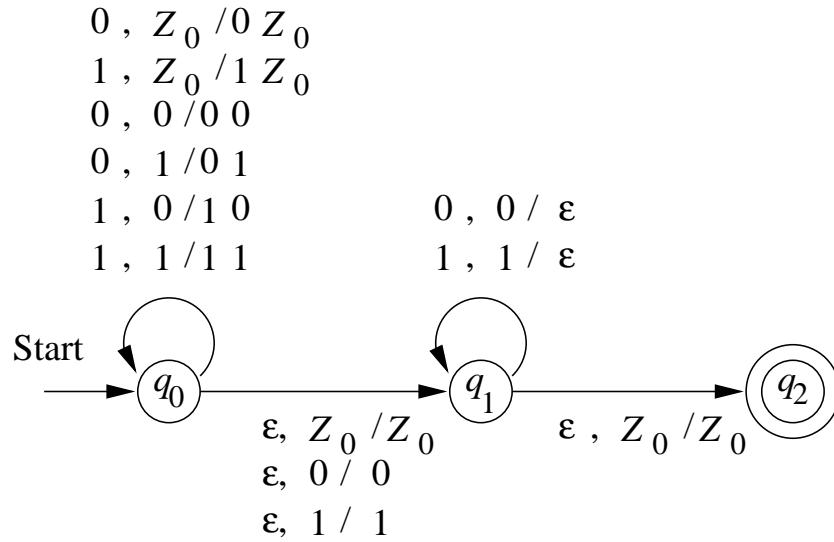
Un PDA e' una tupla di 7 elementi:

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

dove

- Q e' un insieme finito di stati,
- Σ e' un alfabeto finito di input,
- Γ e' un alfabeto finito di pila,
- δ e' una funzione di transizione da $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ a sottinsiemi di $Q \times \Gamma^*$,
- q_0 e' lo stato iniziale,
- $Z_0 \in \Gamma$ e' il simbolo iniziale per la pila, e
- $F \subseteq Q$ e' l'insieme di stati di accettazione.

Esempio: Il PDA



e' la 7-tupla

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\}),$$

dove δ e' data dalla tabella seguente:

S

	0, Z_0	1, Z_0	0, 0	0, 1	1, 0	1, 1	ϵ, Z_0	$\epsilon, 0$	$\epsilon, 1$
$\rightarrow q_0$	$\{(q_0, 0Z_0)\}$	$\{(q_0, 1Z_0)\}$	$\{(q_0, 00)\}$	$\{(q_0, 01)\}$	$\{(q_0, 10)\}$	$\{(q_0, 11)\}$	$\{(q_1, Z_0)\}$	$\{(q_1, 0)\}$	$\{(q_1, 1)\}$
q_1	\emptyset	\emptyset	$\{(q_1, \epsilon)\}$	\emptyset	\emptyset	$\{(q_1, \epsilon)\}$	$\{(q_2, Z_0)\}$	\emptyset	\emptyset
$*q_2$	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset

Descrizioni istantanee

(Configurazione)

Un PDA passa da una configurazione ad un'altra configurazione:

- consumando un simbolo di input (o tramite transizione ϵ),
- consumando la cima dello stack sostituendolo con una stringa (eventualmente vuota).

Per ragionare sulle computazioni dei PDA, usiamo delle *descrizioni istantanee* (ID) del PDA. Una ID e' una tripla

$$(q, w, \gamma)$$

dove q e' lo stato, w l'input rimanente, e γ il contenuto della pila.

Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA. Allora $\forall w \in \Sigma^*, \beta \in \Gamma^*$:

transizione $\xleftarrow{(p, \alpha)} (p, \alpha) \in \delta(q, a, X) \Rightarrow (q, aw, X\beta) \vdash (p, w, \alpha\beta)$.
primo simbolo input \downarrow primo elemento in cima alla pila

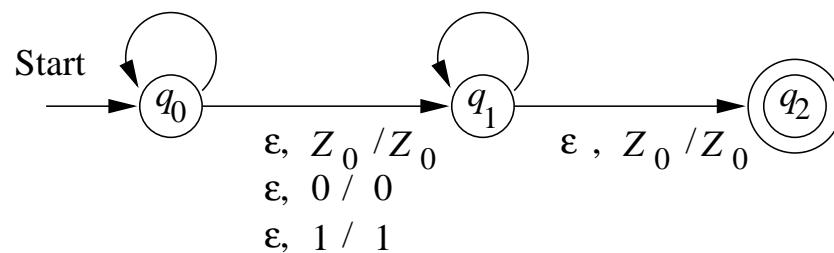
Definiamo \vdash^* la chiusura riflessiva e transitiva di \vdash .

0 o più relazioni di passaggio di configurazione
(quindi 0 o più transizioni)

37
relazione di passaggio di configurazione

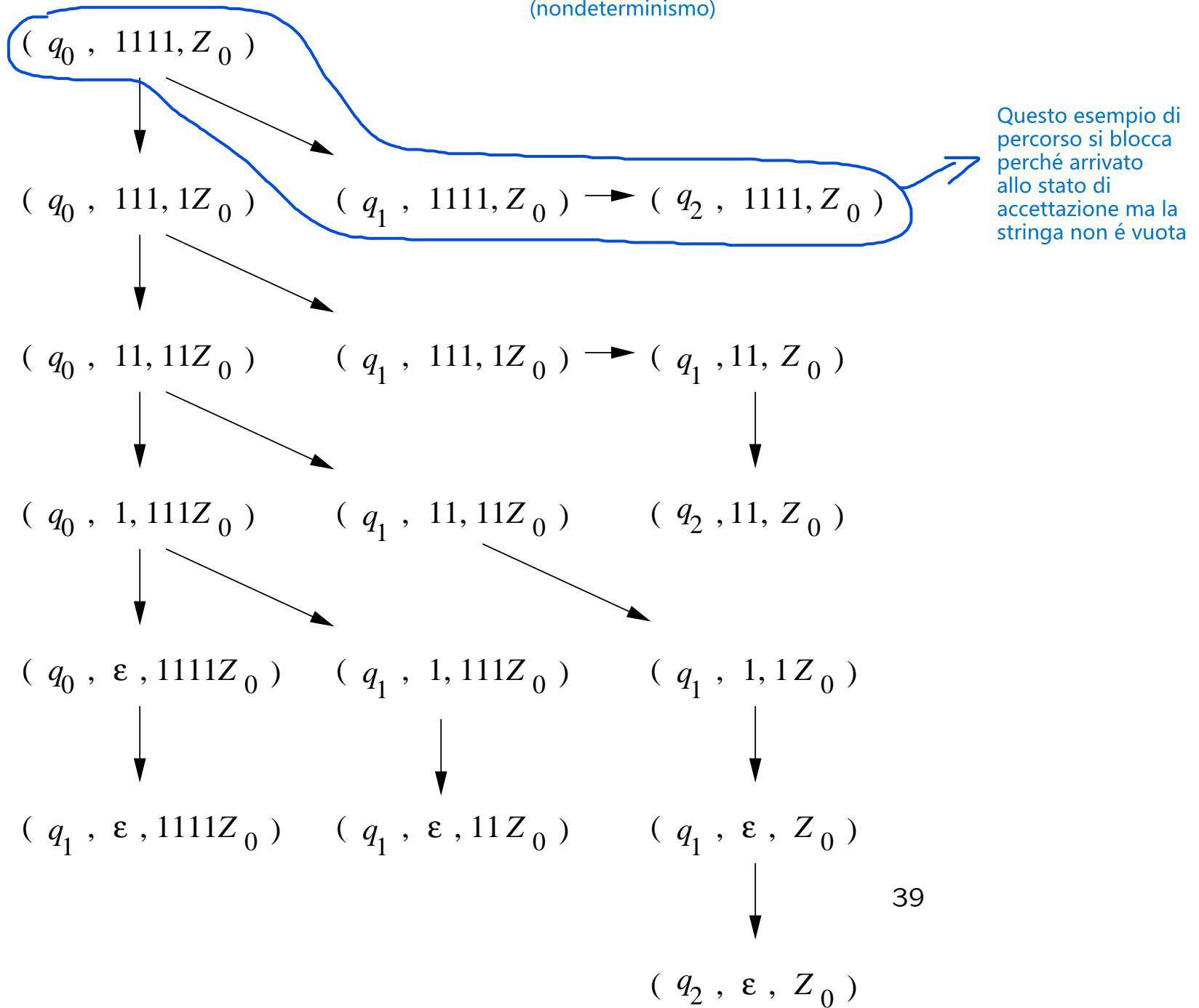
Esempio: Su input 1111 il PDA

0 , $Z_0 / 0 Z_0$
1 , $Z_0 / 1 Z_0$
0 , 0 / 0 0
0 , 1 / 0 1
1 , 0 / 1 0 0 , 0 / ϵ
1 , 1 / 1 1 1 , 1 / ϵ



ha le seguenti sequenze di computazioni:

L'automa fa tutti i percorsi possibili per ogni stato (nondeterminismo)



Affinché un PDA accetti la stringa, ci deve essere almeno un percorso che mi porta alla fine, data la stringa (Certificato)

Un PDA accetta la stringa se arrivato ad uno degli stati di accettazione la stringa è vuota (qualsiasi sia il contenuto della pila che può essere vuota o meno)

Accettazione per stato finale

Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA. Il *linguaggio accettato da P per stato finale* è'

$$L(P) = \{w : (q_0, w, Z_0) \xrightarrow{*} (q, \epsilon, \alpha), q \in F\}.$$

Esempio: Il PDA di prima accetta esattamente L_{www} .

Per riconoscere una stringa come parte del linguaggio, devo arrivare in una configurazione che ha pila vuota e la stringa in input è vuota, senza considerare il tipo di stato (posso terminare anche in uno stato che non è di accettazione)

Accettazione per pila vuota

Questo posso rimuoverlo perché non mi interessano gli stati di accettazione

Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA. Il *linguaggio accettato da P per pila vuota* è'

$$N(P) = \{w : (q_0, w, Z_0) \xrightarrow{*} (q, \epsilon, \epsilon)\}.$$

Nota: q può essere uno stato qualunque.

Domanda: come modificare il PDA per ww^R per accettare lo stesso linguaggio per pila vuota?

①

Da pila vuota a stato finale

linguaggio riconosciuto da PDA per pila vuota

Teorema 6.9: Se $L = N(P_N)$ per un PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, allora \exists PDA P_F , tale che $L = L(P_F)$.

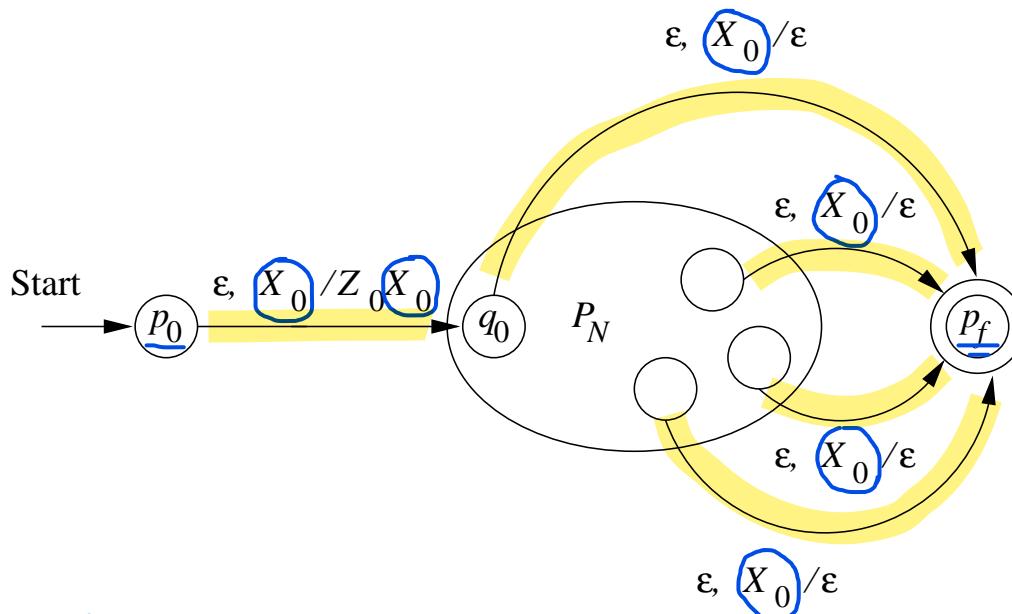
linguaggio riconosciuto da PDA per stato finale

Prova: Sia

F: insieme degli stati di accettazione

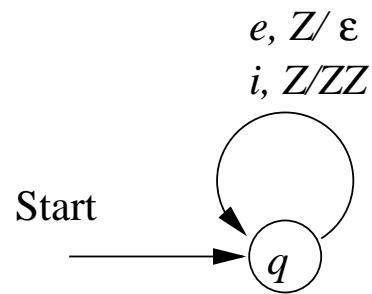
$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, \{p_f\})$$

dove $\delta_F(p_0, \epsilon, X_0) = \{(q_0, Z_0 X_0)\}$, e per ogni $q \in Q, a \in \Sigma \cup \{\epsilon\}, Y \in \Gamma : \delta_F(q, a, Y) = \delta_N(q, a, Y)$, e inoltre $(p_f, \epsilon) \in \delta_F(q, \epsilon, X_0)$.



Aggiungo X_0 (simbolo nuovo) all'alfabeto della pila. All'inizio dell'automa, metto nella pila X_0 e con una transizione da p_0 aggiungo, sopra X_0 , il simbolo iniziale dell'automa a pila vuoto Z_0 . Aggiungo per ogni stato di PDA per pila vuota le transizioni per arrivare allo stato di accettazione creato p_f .

Consideriamo il seguente automa a pila:



Formalmente,

$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z),$$

dove $\delta_N(q, i, Z) = \{(q, ZZ)\}$, e $\delta_N(q, e, Z) = \{(q, \epsilon)\}$.

Da P_N possiamo costruire

$$\underline{P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\})},$$

dove

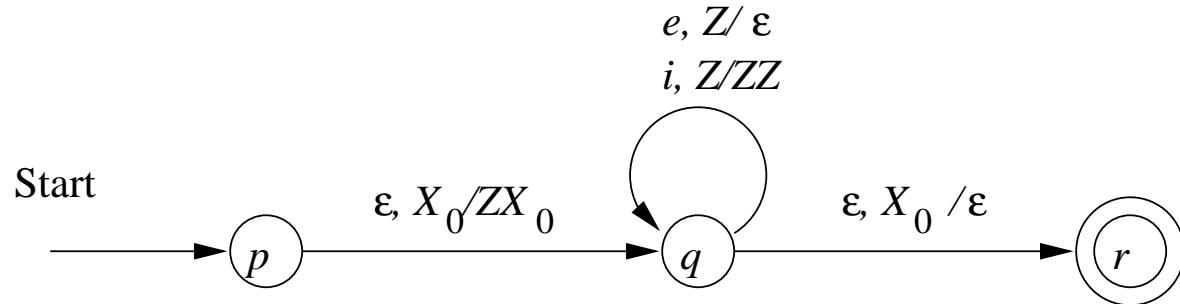
$$\delta_F(p, \epsilon, X_0) = \{(q, ZX_0)\},$$

$$\delta_F(q, i, Z) = \delta_N(q, i, Z) = \{(q, ZZ)\},$$

$$\delta_F(q, e, Z) = \delta_N(q, e, Z) = \{(q, \epsilon)\}, \text{ and}$$

$$\delta_F(q, \epsilon, X_0) = \{(r, \epsilon)\}$$

Il diagramma per P_F e'



2

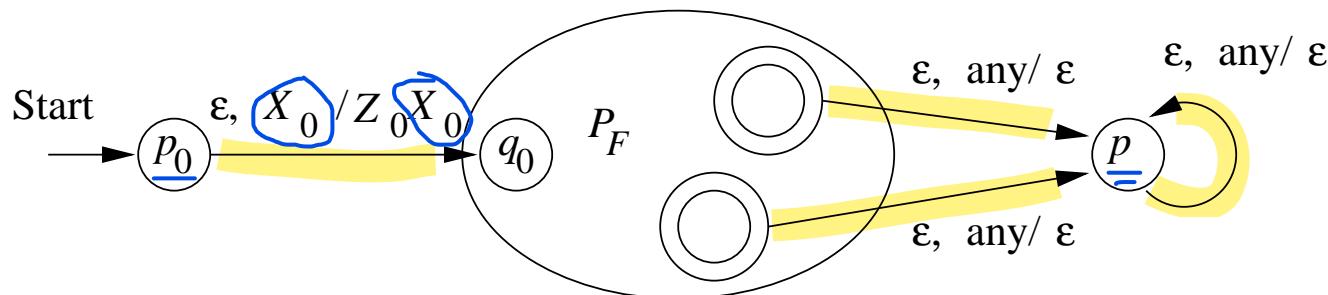
Da stato finale a pila vuota

Teorema 6.11: Sia $L = L(P_F)$, per un PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$. Allora \exists PDA P_N , tale che $L = N(P_N)$.

Prova: Sia

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

dove $\delta_N(p_0, \epsilon, X_0) = \{(q_0, Z_0 X_0)\}$, $\delta_N(p, \epsilon, Y) = \{(p, \epsilon)\}$, per $Y \in \Gamma \cup \{X_0\}$, e per tutti i $q \in Q$, $a \in \Sigma \cup \{\epsilon\}$, $Y \in \Gamma$: $\delta_N(q, a, Y) = \delta_F(q, a, Y)$, e inoltre $\forall q \in F$, e $Y \in \Gamma \cup \{X_0\}$: $(p, \epsilon) \in \delta_N(q, \epsilon, Y)$.



Aggiungo, all'inizio dell'automa, X_0 come primo elemento in cima alla pila. Aggiungo una transizione che mi porta ad aggiungere Z_0 nella pila, sopra ad X_0 . Aggiungo tante transizioni quanti sono gli stati di accettazione * numero di simboli dell'alfabeto della pila ed Aggiungo un autociclo sullo stato p creato per svuotare il restante della pila (any indica che ci saranno n transizioni per quella freccia dato dal numero di simboli dell'alfabeto della pila)

45

Con questa trasformazione, ho l'equivalenza di riconoscimento di linguaggio ($L(P_F) = N(P_N)$)

Equivalenza di PDA e CFG

Un linguaggio e'

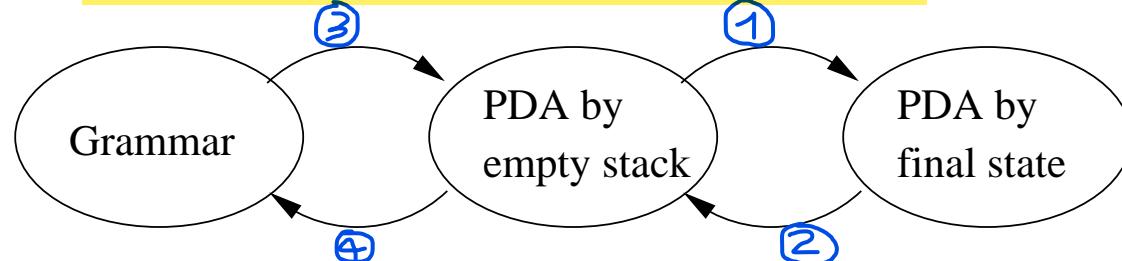
generato da una CFG

se e solo se e'

accettato da un PDA per pila vuota

se e solo se e'

accettato da un PDA per stato finale



Sappiamo già andare da pila vuota a stato finale.

3

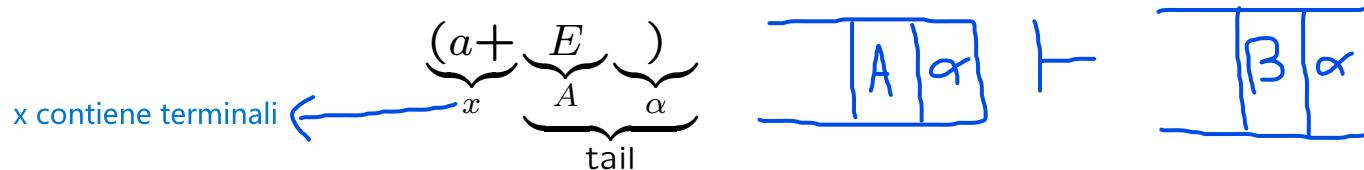
Da CFG a PDA

Idea: data G , costruiamo un PDA che simula \xrightarrow{lm}^* .

Scriviamo le stringhe ottenute lungo una *derivazione sinistra* come

$xA\alpha$

dove A e' la variabile *piu' a sinistra*. Ad esempio,



Sia \xrightarrow{lm}^* sia $xA\alpha \Rightarrow x\beta\alpha$ (a causa di una produzione $A \rightarrow \beta$ della CFG).

Questo corrisponde al PDA che, dopo aver consumato input x , e essersi ritrovato con $A\alpha$ sulla pila, ora esegue una transizione ϵ che elimina A e mette al suo posto β sulla pila.

Piu' formalmente, sia w la stringa data in *input* al PDA e y tale che $w = xy$. Allora il PDA va non deterministicamente dalla configurazione $(q, y, A\alpha)$ alla configurazione $(q, y, \beta\alpha)$.

Alla configurazione $(q, y, \beta\alpha)$ il PDA si comporta come prima, a meno che ci siano *terminali* nel prefisso di β . In questo caso, il PDA li elimina, se li legge nell'input (se fanno match con l'input).

Se tutte le scommesse sono giuste (consentono di matchare l'input), il PDA finisce l'input con la *pila vuota*.

Quindi **la trasformazione è la seguente.**

Sia $G = (V, T, Q, S)$ una CFG. Definiamo P_G come

$$(\{q\}, T, V \cup T, \delta, q, S),$$

dove

$$\delta(q, \epsilon, A) = \{(q, \beta) : A \rightarrow \beta \in Q\},$$

per $A \in V$, e

$$\delta(q, a, a) = \{(q, \epsilon)\},$$

per $a \in T$.

Preso la variabile la sostituisco con il corpo della produzione (letto l'input, metto in cima alla pila il corpo)

Letto l'input che è un terminale, se sulla pila ho lo stesso terminale (match) allora lo rimuovo dalla pila

Ovviamente, per capire se quella stringa in input viene riconosciuta correttamente, devo utilizzare la grammatica per definire la derivazione canonica sinistra e poi utilizzare tale derivazione insieme all'automa creato (che contiene un unico stato con un autociclo contenente le transizioni per le variabili e le transizioni per il match dei terminali) per verificare che la stringa venga riconosciuta correttamente dall'automa

Esempio:

Consideriamo la grammatica

$$S \rightarrow \epsilon | SS | iS | iSe.$$

Il PDA corrispondente e'

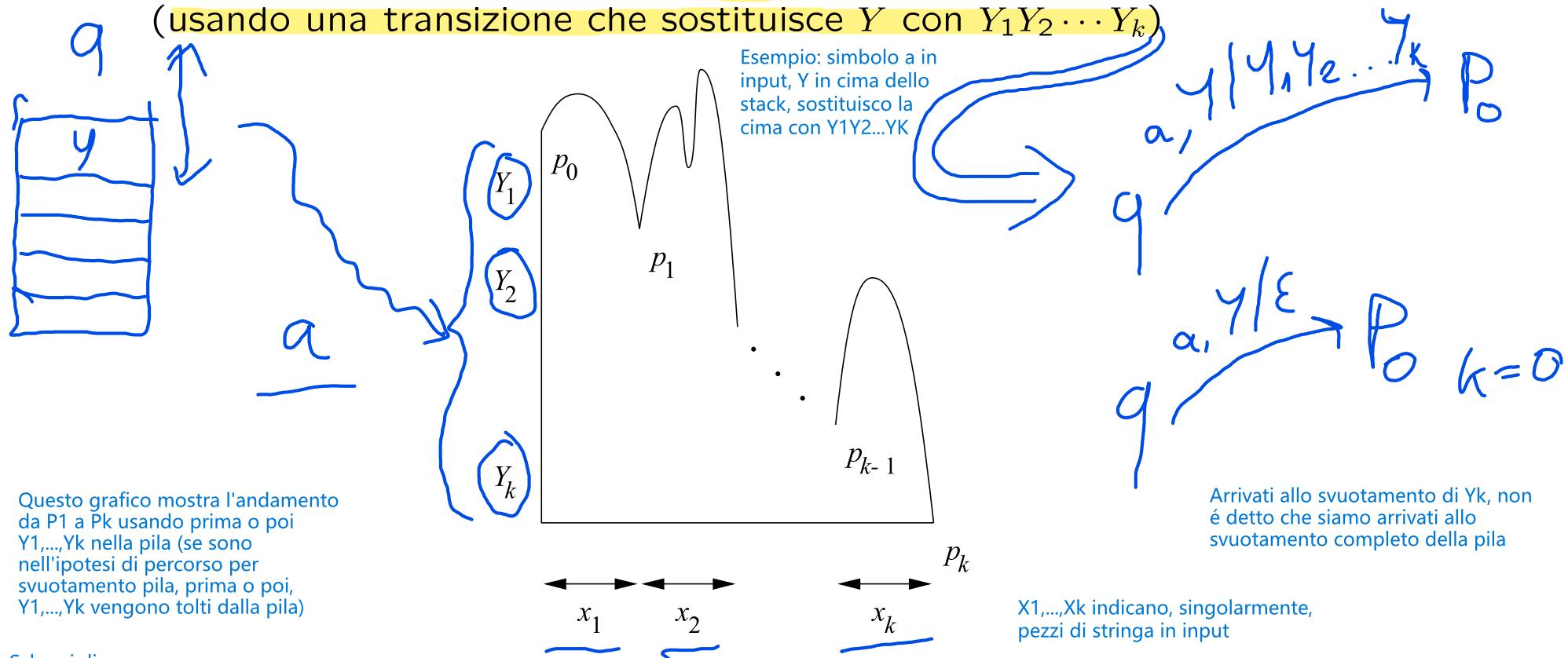
$$P = (\{q\}, \{i, e\}, \{S, i, e\}, \delta, q, S),$$

dove $\delta(q, \epsilon, S) = \{(q, \epsilon), (q, SS), (q, iS), (q, iSe)\}$, $\delta(q, i, i) = \{(q, \epsilon)\}$,
e $\delta(q, e, e) = \{(q, \epsilon)\}$

4

Da PDA a CFG

Idea: comportamento dei PDA per rimuovere simbolo Y dalla pila
(usando una transizione che sostituisce Y con $Y_1 Y_2 \dots Y_k$)



Schemi di produzione

Definiremo una grammatica con variabili della forma $[p_{i-1} Y_i p_i]$ che rappresentano il passaggio da p_{i-1} a p_i con l'effetto di eliminare Y_i .

Ciò che é in Blu lo conosco, ciò che é in Nero non lo conosco

$$\begin{aligned}
 [q \underline{Y} p_k] &\rightarrow a [p_0 \underline{Y_1} p_1] [p_1 \underline{Y_2} p_2] \dots [p_{k-1} \underline{Y_k} p_k]^{51} \quad \forall p_i \in Q \quad 1 \leq i \leq k \\
 [q \underline{Y} p_0] &\rightarrow a \quad k=0 \quad S \rightarrow [q_0 \underline{Z_0} p] \quad \forall p \in Q
 \end{aligned}$$

Produzione Iniziale: questa variabile permette di passare da q_0 a un qualsiasi stato p , rimuovendo il simbolo iniziale della pila Z_0 (e quindi svuotando l'intera pila).

Questa variabile rappresenta tutte le stringhe terminali w che il PDA può leggere (consumare dall'input) mentre si sposta dallo stato iniziale p allo stato finale q , e allo stesso tempo rimuove esattamente il simbolo X dalla cima della pila, senza toccare nient'altro sotto X .

Quindi stringa terminale generata da variabile $[pXq]$ rappresenta:
input letto da PDA andando da p a q e rimuovendo X da pila

alfabeto dei terminali

Formalmente, sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ un PDA. Definiamo $G = (V, \Sigma, R, S)$, con

$$V = \{[pXq] : \{p, q\} \subseteq Q, X \in \Gamma\} \cup \{S\}$$

insieme delle variabili della grammatica

$$R = \{S \rightarrow [q_0 Z_0 p] : p \in Q\} \cup$$

r_1, \dots, r_k sarebbero P_1, \dots, P_k

$$\{[qXr_k] \rightarrow a[rY_1r_1] \cdots [r_{k-1}Y_kr_k] : a \in \Sigma \cup \{\epsilon\}, \{r_1, \dots, r_k\} \subseteq Q, (r, Y_1Y_2 \cdots Y_k) \in \delta(q, a, X)\}$$

insieme delle produzioni

dove, in caso $k = 0$ si ha: $Y_1Y_2 \cdots Y_k = \epsilon$ e $r_k = r$ (Pop Senza Push)

Se $k=0$, $[q_0 X r_0] = a$

Se $k=1$, $[q_0 X r_1] = a [p X r_1]$

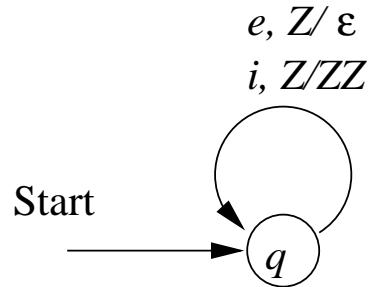
Se $k>1$, $[q_0 X r_k] = a [p Y_1 r_1] \cdots [r_{k-1} Y_k r_k]$

Sequenza di rimozioni (derivazioni) necessarie per eliminare i simboli Y_i (che sono stati inseriti al posto di X).

La produzione simula questa sequenza di azioni:

La variabile $[q X r_k]$ sulla sinistra indica che stiamo cercando la stringa w che porta da q a r_k rimuovendo X . La stringa w deve iniziare con il terminale a letto nella transizione (o epsilon se la mossa è epsilon-transizione). Dopo la transizione (pop X e push $Y_1 \dots Y_k$), il PDA è nello stato r e in cima alla pila c'è Y_1 . Per completare l'eliminazione effettiva di X , bisogna eliminare tutti i simboli Y_1, \dots, Y_k che sono stati appena inseriti. Ogni nuova variabile $[r_i Y_{i+1} r_{i+1}]$ sulla destra gestisce la rimozione di un singolo Y_i . $[r Y_1 r_1]$ genera la parte di input che va dallo stato r a uno stato intermedio r_1 , rimuovendo Y_1 . Questo continua fino a $[r_{k-1} Y_k r_k]$, che rimuove l'ultimo simbolo Y_k , portando allo stato finale r_k (che è lo stato finale della variabile originale $[q X r_k]$).

Esempio: Convertiamo



$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z),$$

dove $\delta_N(q, i, Z) = \{(q, ZZ)\}$,
e $\delta_N(q, e, Z) = \{(q, \epsilon)\}$
in una grammatica

$$G = (V, \{i, e\}, R, S),$$

dove $V = \{[qZq], S\}$ e
 $R = \{S \rightarrow [qZq], [qZq] \rightarrow i[qZq][qZq], [qZq] \rightarrow e\}.$

Se rimpiazziamo $[qZq]$ con A otteniamo le produzioni $S \rightarrow A$ e $A \rightarrow iAA|e$.

Le triple servono per la costruzione delle produzioni, poi posso sostituire le triple con dei nomi più "normali" (ad ogni tripla diversa, un nome diverso)

Esempio: Convertiamo $P = (\{p, q\}, \{0, 1\}, \{X, Z_0\}, \delta, q, Z_0)$, dove δ e' data da

1. $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$

$$\xrightarrow{q} 1, Z_0 / XZ_0$$

2. $\delta(q, 1, X) = \{(q, XX)\}$

$$\xrightarrow{q} 1, X / XX$$

3. $\delta(q, 0, X) = \{(p, X)\}$

$$q \xrightarrow{0, X / X} p$$

4. $\delta(q, \epsilon, X) = \{(q, \epsilon)\}$

$$\xrightarrow{q} \epsilon, X / \epsilon$$

5. $\delta(p, 1, X) = \{(p, \epsilon)\}$

$$\xrightarrow{p} 1, X / \epsilon$$

6. $\delta(p, 0, Z_0) = \{(q, Z_0)\}$

$$q \xleftarrow{p, Z_0 / Z_0} p$$

in una CFG.

Otteniamo $G = (V, \{0, 1\}, R, S)$, dove

Insieme delle variabili

$$V = \{[qZ_0q], [pZ_0q], [qZ_0p], [pZ_0p], [qXq], [pXq], [qXp], [pXp], S\}$$

e le produzioni in R sono

$$S \rightarrow [qZ_0q] | [qZ_0p]$$

4 produzioni

Dalla transizione (1) $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$ si ha:

- 1 $[qZ_0q] \rightarrow 1[qXq][qZ_0q]$
- 2 $[qZ_0q] \rightarrow 1[qXp][pZ_0q]$
- 3 $[qZ_0p] \rightarrow 1[qXq][qZ_0p]$
- 4 $[qZ_0p] \rightarrow 1[qXp][pZ_0p]$

Dalla transizione (2) $\delta(q, 1, X) = \{(q, XX)\}$ si ha:

4 produzioni

- 1 $[qXq] \rightarrow 1[qXq][qXq]$
- 2 $[qXq] \rightarrow 1[qXp][pXq]$
- 3 $[qXp] \rightarrow 1[qXq][qXp]$
- 4 $[qXp] \rightarrow 1[qXp][pXp]$

Dalla transizione (3) $\delta(q, 0, X) = \{(p, X)\}$ si ha:

- 1 $[qXq] \rightarrow 0[pXq]$
- 2 $[qXp] \rightarrow 0[pXp]$

2 produzioni

Dalla transizione (4) $\delta(q, \epsilon, X) = \{(q, \epsilon)\}$ si ha:

$$[qXq] \rightarrow \epsilon$$

1 produzione

Dalla transizione (5) $\delta(p, 1, X) = \{(p, \epsilon)\}$ si ha:

$$[pXp] \rightarrow 1$$

1 produzione

Dalla transizione (6) $\delta(p, 0, Z_0) = \{(q, Z_0)\}$ si ha:

$$\begin{aligned} [pZ_0q] &\rightarrow 0[qZ_0q] \\ [pZ_0p] &\rightarrow 0[qZ_0p] \end{aligned}$$

2 produzioni

$d(q, a, X) = \{ (q_1, \alpha); (q_2, \beta) \} \rightarrow$ non deterministico (ha più di un percorso da uno stato)

Per renderlo deterministico, per ogni stato con stessa stringa in input e cima dello stack, devo avere un percorso



PDA deterministici

Un PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ è deterministico se e solo se:

1. ogni $\delta(q, a, X)$, con $a \in \Sigma \cup \{\epsilon\}$, contiene al piu' un elemento
2. se $\delta(q, a, X)$ non vuoto per un $a \in \Sigma$, allora $\delta(q, \epsilon, X)$ vuoto.

delta deve avere massimo un elemento (cioè un unico percorso per input-cima stack)

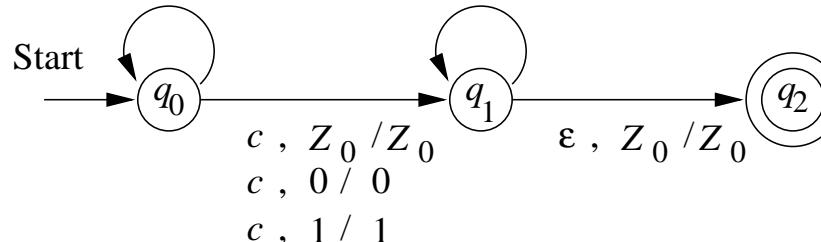
Esempio: Definiamo

$$L_{wcwr} = \{wcw^R : w \in \{0, 1\}^*\}$$

Allora L_{wcwr} è riconosciuto dal seguente DPDA

0 , $Z_0 / 0 Z_0$
1 , $Z_0 / 1 Z_0$
0 , $0 / 0$
0 , $1 / 0$
1 , $0 / 1$
1 , $1 / 1$

0 , $0 / \epsilon$
1 , $1 / \epsilon$

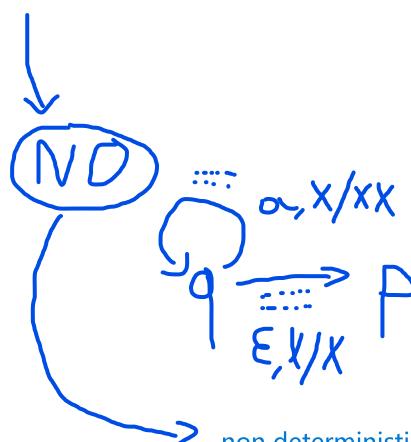


ww^R (stringhe palindrome) non vengono riconosciute da DPDA perché è grazie al nondeterminismo che le stringhe palindrome vengono riconosciute (il nondet. mi serve per capire quando sono a metà della stringa)

per mantenere espressività, ma con determinismo, aggiungo a metà un simbolo che mi indica che siamo a metà della stringa (wcw^R)

Con Determinismo, riduciamo l'espressività dell'automa e linguaggi che esso riconosce

il non determinismo può avvenire anche con transizioni con stesso elemento in cima allo stack, ma con qualsiasi simbolo in input 'a' ed epsilon (perché posso scegliere uno dei due percorsi). Quindi, se ho qualche transizione con (a,X), non posso avere transizioni con (epsilon,X), partendo dallo stesso stato



non deterministico: ho la possibilità di scegliere due percorsi

Tutti i linguaggi Regolari sono anche linguaggi Context-Free (CFL) e possono essere riconosciuti da un particolare tipo di Automa a Pila Deterministico (DPDA).

DPDA che accettano per stato finale

Classe dei linguaggi

Mostreremo che $\text{Regolari} \subset L(\text{DPDA}) \subset \text{CFL}$

Conseguenza: Poiché i linguaggi regolari sono accettati da un DPDA, e i linguaggi accettati da un DPDA sono context-free, ne consegue che tutti i linguaggi regolari sono context-free.

Ogni linguaggio regolare può essere accettato da un DPDA.

Teorema 6.17: Se L è regolare, allora $L = L(P)$ per qualche DPDA P .

Prova: Dato che L è regolare, esiste un DFA A tale che $L = L(A)$.
Sia

$$A = (Q, \Sigma, \delta_A, q_0, F)$$

definiamo il DPDA

$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F),$$

dove

$$\delta_P(q, a, Z_0) = \{(\delta_A(q, a), Z_0)\},$$

per tutti i $p, q \in Q$ e $a \in \Sigma$.

In pratica, il DPDA ignora completamente la pila e usa le sue transizioni per ricalcare fedelmente gli spostamenti del DFA. L'unica differenza è che l'accettazione avviene per stato finale, proprio come nel DFA.

Un'induzione su $|w|$ ci da'

$$(q_0, w, Z_0) \xrightarrow{*} (p, \epsilon, Z_0) \Leftrightarrow \hat{\delta}_A(q_0, w) = p$$

Leggere l'intera stringa w nel PDA (partendo da q_0 con Z_0 nella pila) porta a uno stato finale p , con la pila ancora a Z_0 .

Poiché l'equivalenza \Leftrightarrow è dimostrata, il linguaggio $L(A)$ accettato dal DFA è esattamente lo stesso del linguaggio $L(P)$ accettato dal DPDA.

58

Leggere l'intera stringa w nel DFA (partendo da q_0) porta allo stato p .



Poiché le transizioni sono definite in modo identico e la pila non ha effetto, l'automa P si comporta esattamente come l'automa A .

esempio di un linguaggio
context-free deterministico
che non è regolare.



- Abbiamo visto che $\text{Regolari} \subseteq L(\text{DPDA})$.
- $L_{wcwr} \in L(\text{DPDA}) \setminus \text{Regolari}$ (Linguaggi riconosciuti da Automa a Pila Deterministico, ma non riconosciuti da DFA, quindi non regolari)
- Ci sono linguaggi in $\text{CFL} \setminus L(\text{DPDA})$. (Linguaggi riconosciuti da Automa a Pila nondeterministico, ma non riconosciuti da Automa a Pila deterministico)

Si, per esempio L_{wwr} .



esempio di un linguaggio context-free non deterministico

DPDA che accettano per pila vuota

E i DPDA che accettano per pila vuota?

Possono riconoscere solo linguaggi con la **proprieta' del prefisso**.

Un linguaggio L ha la *proprieta' del prefisso* se **non** esistono due stringhe distinte in L , tali che una e' un prefisso dell'altra.

Esempio: L_{wcwr} ha la proprieta' del prefisso.

Esempio: $\{0\}^*$ non ha la proprieta' del prefisso.

Teorema 6.19: L e' $N(P)$ per qualche DPDA P se e solo se L ha la proprieta' del prefisso e L e' $L(P')$ per qualche DPDA P' .

l'insieme dei linguaggi accettati da DPDA non coincide con l'insieme dei linguaggi che ammettono una grammatica non ambigua (chiamati anche linguaggi Non Inerentemente Ambigui).

DPDA e non ambiguità

L(DPDA) coincide con i CFL aventi grammatiche **non ambigue** (cioè' non inerentemente ambigui)? **No**. Per esempio:

Esempio: L_{wwr} ha una grammatica non ambigua $S \rightarrow 0S0|1S1|\epsilon$ ma non è $L(\text{DPDA})$.

L'inverso invece vale! Abbiamo, preliminarmente:

Linguaggio riconosciuto per pila vuota

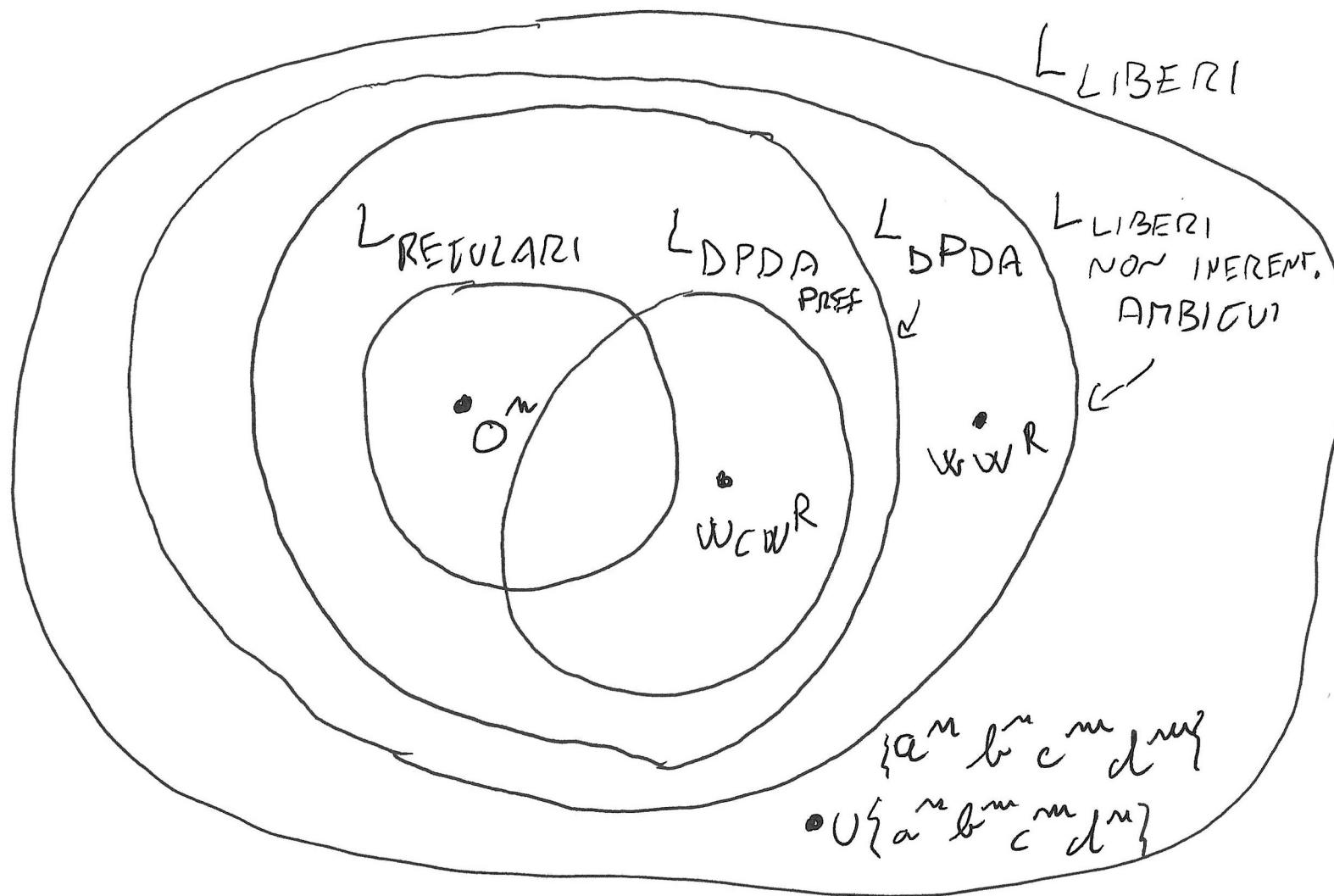
Teorema 6.20: Se $L = N(P)$ per qualche DPDA P , allora L ha una CFG non ambigua.

Prova: Applicando la costruzione vista da PDA a CFG, se la costruzione è applicata ad un DPDA, il risultato è una CFG con derivazioni a sinistra uniche per ogni stringa.

Perché derivazione sinistra unica implica non ambiguità?

La variabile $[p \ X \ q]$ è definita in modo che la sua derivazione corrisponda a una sequenza unica di mosse del DPDA che rimuove X e consuma una stringa w . Poiché il DPDA è deterministico, per una data stringa w ed una configurazione di partenza, esiste una sola sequenza di mosse. Questa sequenza unica di mosse si traduce in una unica sequenza di applicazioni di regole nella CFG risultante. Se esiste una sola sequenza di applicazioni di regole (derivazione sinistra unica) per ogni stringa, allora la grammatica è per definizione non ambigua.

$L(\text{DPDA})$ è un sottoinsieme proprio dei linguaggi con grammatiche non ambigue.



Per estendere il teorema, dobbiamo portare la proprietà del prefisso anche ai DPDA per stato finale



Quindi, verifichiamo che i linguaggi accettati da DPDA sono sottoinsieme dei linguaggi liberi da contesto non inerentemente ambigui



Teorema 6.20 puo' essere rafforzato: Verifichiamo che non vale solo per DPDA che accettano per pila vuota, ma anche per DPDA che accettano per stato finale
Linguaggio riconosciuto per stato finale

Teorema 6.21: Se $L = \overline{L(P)}$ per qualche DPDA P , allora L ha una CFG non ambigua.

Prova: Sia $\$$ un simbolo fuori dell'alfabeto di L , e sia $L' = L\{\$\}$. E' facile modificare P per riconoscere L' (PDA ancora deterministico); inoltre L' ha la proprietà del prefisso.

Per ogni stringa di L , si concatena con $\$$ alla fine di ogni stringa

Per il teorema 6.19 abbiamo $L' = N(P')$ per qualche DPDA P' .

Linguaggio riconosciuto per pila vuota

Per il teorema 6.20 L' puo' essere generato da una CFG G' non ambigua

Modifichiamo G' in G , tale che $L(G) = L$, aggiungendo la produzione

$$\$ \rightarrow \epsilon$$

(e considerando $\$$ una variabile anziche' un terminale)

Dato che G' ha derivazioni a sinistra uniche, anche G le avra' uniche, dato che l'unica cosa nuova e' l'aggiunta di derivazioni

$$w\$ \xrightarrow{lm} w$$

alla fine.

Proprieta' dei CFL

- 1 • *Semplificazione di una CFG.* Se un linguaggio e' un CFL, ha una grammatica in una possibile forma speciale.
- 2 • *Pumping Lemma per CFL.* Simile ai linguaggi regolari. (ma piú complesso)
- 3 • *Proprieta' di chiusura.* Solo alcune delle proprietà di chiusura dei linguaggi regolari valgono anche per i CFL.
- 4 • *Proprieta' di decisione.* Possiamo controllare l'appartenenza e l'essere vuoto, ma, per esempio, l'equivalenza di CFL e' non verificabile tramite un algoritmo (indecidibile).

Con Forma Normale di Chomsky, da qualsiasi grammatica a grammatica con queste due forme ($A \rightarrow BC$, $A \rightarrow a$) si viene a creare sempre un albero binario.

Consiglio: pulire la grammatica prima di rinominare le triple dopo la trasformazione PDA -> CFG

1

Forma normale di Chomsky

Ogni CFL (senza ϵ) e' generato da una CFG dove tutte le produzioni sono della forma

$$A \rightarrow BC, \text{ o } A \rightarrow a$$

dove A, B , e C sono variabili, e a e' un simbolo terminale. Questa e' detta forma normale di Chomsky (CNF), e per ottenerla dobbiamo innanzitutto "pulire" la grammatica:

- A • Eliminare i *simboli inutili*, quelli che non appaiono in nessuna derivazione $S \xrightarrow{*} w$, per simbolo iniziale S e terminale w .
- B • Eliminare le produzioni ϵ , della forma $A \rightarrow \epsilon$.
- C • Eliminare le *produzioni unita'*, cioe' produzioni della forma $A \rightarrow B$, dove A e B sono variabili.

- X generante: testa di produzione che genera una stringa di terminali (le variabili non generanti le tolgo). Possiamo dire che X generante se genera un terminale (X testa di una produzione che genera un terminale)
- X raggiungibile: se fa parte di una derivazione

A

Eliminazione simboli inutili

- Un simbolo X è *utile* per una grammatica $G = (V, T, P, S)$, se esiste una derivazione

$$S \xrightarrow[G]{*} \alpha X \beta \xrightarrow[G]{*} w$$

per una stringa di terminali w . Simboli che non sono utili sono detti *inutili*.

- Un simbolo X è *generante* se $X \xrightarrow[G]{*} w$, per qualche $w \in T^*$
- Un simbolo X è *raggiungibile* se $S \xrightarrow[G]{*} \alpha X \beta$, per qualche $\{\alpha, \beta\} \subseteq (V \cup T)^*$

\Rightarrow Se in G (con $L(G) \neq \emptyset$) eliminiamo prima i simboli non generanti, e poi quelli non raggiungibili, rimarranno solamente simboli utili.

- 1 - Vedo se S,A,B sono generanti (in questo primo passo, vedo che S e A sono generanti mentre B non lo è perché non genera nessun terminale, cioè partendo da B non si deriva terminali).
- 2 - Vedo se S e A raggiungibili (essendo S la variabile d'inizio, è raggiungibile, mentre A non lo è perché non si raggiunge a partendo dalla variabile d'inizio)
- 3 - Rifaccio la stessa cosa, controllo se ci sono generanti e raggiungibili: non ci sono delle modifiche da fare, quindi ho una grammatica equivalente con sole variabili utili.

Esempio: Sia G la grammatica

$$S \rightarrow AB|a, A \rightarrow b$$

- 1 S e A sono generanti, B non lo è. Se eliminiamo B dobbiamo eliminare $S \rightarrow AB$, riducendo la grammatica

$$S \rightarrow a, A \rightarrow b$$

- 2 Ora, solo la variabile S è raggiungibile. Eliminando A rimane solo

$$S \rightarrow a$$

con linguaggio $\{a\}$.

Nota Se eliminiamo prima i simboli non raggiungibili, si ha che tutti i simboli sono raggiungibili. Da

$$S \rightarrow AB|a, A \rightarrow b$$

eliminiamo B in quanto non generante, e rimane la grammatica

$$S \rightarrow a, A \rightarrow b$$

che contiene ancora simboli inutili

Questo mostra che se elimino prima i non raggiungibili e poi i non generanti, l'algoritmo non funziona (gli inutili rimangono)

Nell'eliminazione delle produzioni epsilon, perdo dal linguaggio la possibilità di generare/riconoscere la stringa vuota

B

Eliminazione produzioni ϵ

Si ha che se L è un CFL, allora $L \setminus \{\epsilon\}$ ha una grammatica priva di produzioni ϵ (cioè mostra anche che $L \setminus \{\epsilon\}$ è CFL).

La variabile A è *annullabile* se $A \xrightarrow{*} \epsilon$.

Sia A annullabile. Rimpiazzeremo una regola del tipo

$$B \rightarrow \alpha A \beta$$

con

$$B \rightarrow \alpha A \beta, \quad B \rightarrow \alpha \beta$$

Per rimuoverla, creo due produzione dove A viene considerata e non viene considerata

(rimpiazzando in tal modo anche le nuove regole via via ottenute) e cancelleremo tutte le regole con corpo ϵ .

↗ insieme delle variabili nullable di G

Indichiamo con $n(G)$, l'insieme dei simboli annullabili di una grammatica $G = (V, T, P, S)$

Perché queste 3 variabili, applicando le derivazioni,
arrivano ad una produzione con epsilon

A e B nullable direttamente (hanno
una produzione con epsilon),
mentre S nullable indirettamente
(dopo tot passi nella derivazione
arrivano ad una produzione epsilon)

Esempio: Sia G la grammatica

$$S \rightarrow AB, A \rightarrow aAA|\epsilon, B \rightarrow bBB|\epsilon$$

Abbiamo $n(G) = \{A, B, S\}$. La prima regola diventa

$$S \rightarrow AB|A|B \quad \text{dove considero A e B insieme, solo A e solo B}$$

la seconda

$$A \rightarrow aAA|aA|aA|a \quad \text{dove considero A e non}$$

e la terza

$$B \rightarrow bBB|bB|bB|b \quad \text{dove considero B e non}$$

Eliminiamo le regole con corpo ϵ , ed otteniamo la grammatica G_1 :

$$S \rightarrow AB|A|B, A \rightarrow aAA|aA|a, B \rightarrow bBB|bB|b$$

sono state eliminate anche le produzioni di A e B
che si ripetevano (es: $A \rightarrow aA|aA|... \Rightarrow A \rightarrow aA|...$)

Procedimento ad espansione:

Per rimuovere le produzioni unità, es: $E \rightarrow T$, espando la variabile in cui vado della produzione unità fino a che non ho più una produzione unità nel corpo della produzione unità iniziale. Alla fine della rimozione della produzione unità, unisco le nuove produzioni create con le produzioni non unità della testa di produzione

C

Eliminazione produzioni unità'

$$A \rightarrow B$$

e' una produzione *unita'*, nel caso in cui A e B siano variabili.

Produzioni unita' possono essere eliminate.

Si consideri la grammatica

$$\begin{cases} E \rightarrow T \mid E + T \\ T \rightarrow F \mid T * F \\ F \rightarrow I \mid (E) \\ I \rightarrow a \mid b \mid Ia \mid Ib \mid IO \mid I1 \end{cases}$$

ha le produzioni unita' $E \rightarrow T$, $T \rightarrow F$, e $F \rightarrow I$

Si consideri la produzione unità $E \rightarrow T$. Si trasforma tale produzione con il seguente procedimento a **espansione**.

Si espande $E \rightarrow T$ ottenendo le produzioni:

$$E \rightarrow F, E \rightarrow T * F$$

Poi, espandendo $E \rightarrow F$, si ottiene:

$$E \rightarrow I | (E) | T * F$$

Infine, espandendo $E \rightarrow I$, si ottiene:

$$E \rightarrow a | b | Ia | Ib | I0 | I1 | (E) | T * F$$

non ho più produzioni unità nel corpo della produzione iniziale

Si considerano poi le altre produzioni unità $T \rightarrow F$ e $F \rightarrow I$ della grammatica e, per ciascuna, si applica analogo procedimento.

La grammatica iniziale

$$\begin{array}{l} E \rightarrow T \mid \underline{\underline{E + T}} \\ T \rightarrow F \mid T * F \\ F \rightarrow I \mid (E) \\ I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \end{array}$$

viene quindi modificata trasformando:

$E \rightarrow T$ in

$$E \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \mid T * F$$

$T \rightarrow F$ in

$$T \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E)$$

$F \rightarrow I$ in

$$F \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

Quindi, eliminando le produzioni unità, la grammatica diviene

$$\begin{array}{l} E \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \mid T * F \mid \underline{\underline{E + T}} \\ T \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \mid T * F \mid \underline{\underline{-}} \\ F \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \\ I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \end{array}$$

ho unito le produzioni non unità con le produzioni unità trasformate tramite procedimento ad espansione

Consideriamo ancora il **procedimento a espansione usato per trasformare le produzioni unità**. Ad esempio per $E \rightarrow T$:

Si espande $E \rightarrow T$ ottenendo le produzioni:

$$E \rightarrow F, E \rightarrow T * F$$

Poi, espandendo $E \rightarrow F$, si ottiene:

$$E \rightarrow I|(E)|T * F$$

Infine, espandendo $E \rightarrow I$, si ottiene:

$$E \rightarrow a | b | Ia | Ib | IO | I1 | (E) | T * F$$



Questo procedimento funziona per qualsiasi grammatica?

Questo procedimento non funziona per tutte le grammatiche
(il problema nasce quando c'è un ciclo (es: A->B, B->C, C->A))

No! Se ci sono **cicli** il procedimento visto non consente di giungere alla rimozione delle produzioni unità che via via si generano!

Si consideri per esempio la grammatica: $A \rightarrow B, B \rightarrow C, C \rightarrow A$

Soluzione:

Se durante il procedimento sopra **si genera una produzione unità che si ha già espanso** la si può semplicemente **rimuovere** (espandendola si otterrebbero di nuovo produzioni già generate)

Esempio: si consideri la grammatica

$$\begin{array}{l} A \rightarrow B \mid a \\ B \rightarrow C \mid b \\ C \rightarrow A \mid c \end{array}$$

Comincio trasformando la produzione unità $A \rightarrow B$.

Si espande $A \rightarrow B$ ottenendo le produzioni:

$$A \rightarrow C \mid b$$

Poi, espandendo $A \rightarrow C$, si ottiene:

$$A \rightarrow A \mid c \mid b$$

Infine, espandendo $A \rightarrow A$, si ottiene:

$$A \rightarrow B \mid a \mid c \mid b$$

Andando avanti ottengo ovviamente sempre produzioni che ho già, quindi mi posso fermare ed eliminare $A \rightarrow B$.

La produzione unità $A \rightarrow B$ si trasforma quindi in:

$$A \rightarrow a \mid c \mid b$$

La grammatica iniziale

$$\begin{array}{l} A \rightarrow B \mid a \\ B \rightarrow C \mid b \\ C \rightarrow A \mid c \end{array}$$

viene quindi modificata trasformando:

$A \rightarrow B$ in

$$\underline{A \rightarrow a \mid c \mid b}$$

$B \rightarrow C$ in

$$\underline{B \rightarrow b \mid a \mid c}$$

$C \rightarrow A$ in

$$\underline{C \rightarrow c \mid b \mid a}$$

Quindi, eliminando le produzioni unità, la grammatica diviene

$$\begin{array}{l} A \rightarrow a \mid b \mid c \\ B \rightarrow a \mid b \mid c \\ C \rightarrow a \mid b \mid c \end{array}$$

La grammatica generata partendo da A, all'eliminazione delle produzioni unità, B e C non sono raggiungibili, quindi possibile pulirla rendendola con una sola variabile (A) e tre produzioni: A->a|b|c

Sommario

Per “pulire” una grammatica si deve

1. Eliminare le produzioni ϵ
2. Eliminare le produzioni unita'
3. Eliminare i simboli inutili

in questo ordine. Seguendo così i passaggi, sono sicuro di ottenere una grammatica pulita.

Esercizio. Trovare una grammatica in cui cambiando l'ordine non si ottiene una versione “pulita”

Le produzioni con 2 o più simboli (terminali e variabili), le trasformo in variabili (creando variabili sia per i terminali sia per gli operatori). Così ho produzioni che vanno in produzioni con 2 o più variabili e produzioni con un solo terminale. Quindi, le regole in CNF non possono mischiare terminali e variabili (come $S \rightarrow aS$). Dobbiamo fare in modo che i terminali stiano da soli. L'obiettivo è trasformare regole come $A \rightarrow aB$ o $A \rightarrow a'$ in $A \rightarrow XB$ e $X \rightarrow a'$.

Le produzioni che ho dopo aver effettuato il passo 2, non sono quelle finali (devo ridurre il corpo delle produzioni con 3 o più variabili, $>=3$)

In una grammatica CNF, non posso generare la stringa vuota

Forma Normale di Chomsky, CNF

Il suo scopo principale non è essere leggibile per un umano, ma essere estremamente semplice e prevedibile per un computer.

Ogni CFL non vuoto, che non contiene ϵ , ha una grammatica G priva di simboli inutili, con produzioni nella forma

- $A \rightarrow BC$, dove $\{A, B, C\} \subseteq V$, o
- $A \rightarrow a$, dove $A \in V$, e $a \in T$.

Per ottenerla, si effettuano le seguenti trasformazioni su una qualsiasi grammatica per il CFL

1. "Pulire" la grammatica
2. Modificare le produzioni con 2 o piu' simboli in modo tale che siano tutte variabili
3. Ridurre il corpo delle regole di lunghezza superiore a 2 in cascate di produzioni con corpi da 2 variabili.

Passo 3 da svolgere dopo passo 2

Le produzioni con 3 o più variabili ($>=3$), li trasformo in produzioni con corpi da 2 variabili. Così ho produzioni con corpi con 1 o 2 variabili e produzioni con un solo terminale. L'ultimo passo è sistemare le regole che hanno più di due variabili a destra, come $S \rightarrow XSB$. Dobbiamo "spezzarle" in una cascata di regole con solo due variabili. La regola $S \rightarrow XSB$ è troppo lunga. La spezziamo introducendo una nuova variabile "di supporto", ad esempio $S \rightarrow XY$ e $Y \rightarrow SB$

- Devo avere corpi di produzione con terminali isolati e variabili insieme. Per questo, creo delle variabili che contengono solo terminali, in modo da avere nelle produzioni con piú di un simbolo solo variabili
- Per il passo 2, per ogni terminale a che compare in un corpo di lunghezza ≥ 2 , creare una nuova variabile, ad esempio A , e sostituire a con A in tutti i corpi, e aggiungere la nuova regola $A \rightarrow a$.

- Per il passo 3, per ogni regola nella forma

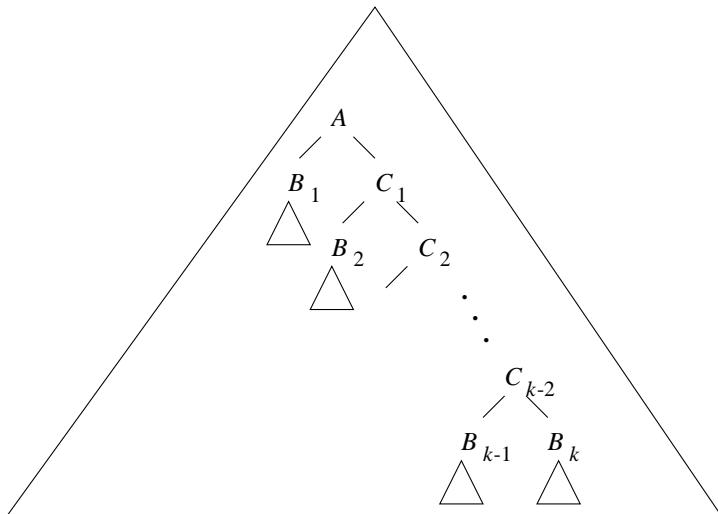
$$A \rightarrow B_1 B_2 \cdots B_k,$$

$k \geq 3$, introdurre le nuove variabili C_1, C_2, \dots, C_{k-2} , e sostituire la regola con

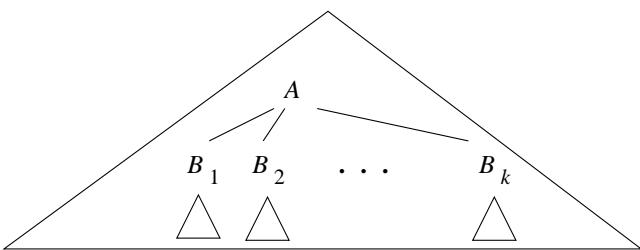
$$\begin{array}{ll} A & \rightarrow B_1 C_1 \\ C_1 & \rightarrow B_2 C_2 \\ & \vdots \\ C_{k-3} & \rightarrow B_{k-2} C_{k-2} \\ C_{k-2} & \rightarrow B_{k-1} B_k \end{array}$$

L'obiettivo è ridurre la lunghezza dei corpi di produzione delle variabili (dopo aver effettuato passo 2) in soli corpi di produzione di lunghezza massima 2

Illustrazione dell'effetto del passo 3.



(a) Dopo il passo 3



(b) Prima del Passo 3

Esempio

Iniziamo dalla grammatica

$$\begin{aligned} E &\rightarrow E + T \mid T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid IO \mid I1 \\ T &\rightarrow T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid IO \mid I1 \\ F &\rightarrow (E) \mid a \mid b \mid Ia \mid Ib \mid IO \mid I1 \\ I &\rightarrow a \mid b \mid Ia \mid Ib \mid IO \mid I1 \end{aligned}$$

Per il passo 2 usiamo le regole

$$A \rightarrow a, B \rightarrow b, Z \rightarrow 0, O \rightarrow 1$$

$$P \rightarrow +, M \rightarrow *, L \rightarrow (, R \rightarrow)$$

e otteniamo la grammatica

$$\begin{aligned} E &\rightarrow EPT \mid TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ T &\rightarrow TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ F &\rightarrow LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ I &\rightarrow a \mid b \mid IA \mid IB \mid IZ \mid IO \\ A &\rightarrow a, B \rightarrow b, Z \rightarrow 0, O \rightarrow 1 \\ P &\rightarrow +, M \rightarrow *, L \rightarrow (, R \rightarrow) \end{aligned}$$

Per il passo 3, rimpiazziamo

$E \rightarrow EPT$ con $E \rightarrow EC_1, C_1 \rightarrow PT$

$E \rightarrow TMF, T \rightarrow TMF$ con
 $E \rightarrow TC_2, T \rightarrow TC_2, C_2 \rightarrow MF$

$E \rightarrow LER, T \rightarrow LER, F \rightarrow LER$ con
 $E \rightarrow LC_3, T \rightarrow LC_3, F \rightarrow LC_3, C_3 \rightarrow ER$

La grammatica in CNF finale e'

$E \rightarrow EC_1 \mid TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$
 $T \rightarrow TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$
 $F \rightarrow LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$
 $I \rightarrow a \mid b \mid IA \mid IB \mid IZ \mid IO$
 $C_1 \rightarrow PT, C_2 \rightarrow MF, C_3 \rightarrow ER$
 $A \rightarrow a, B \rightarrow b, Z \rightarrow 0, O \rightarrow 1$
 $P \rightarrow +, M \rightarrow *, L \rightarrow (, R \rightarrow)$

Questa ripetizione crea un ciclo. La sottostringa y che fa percorrere quel ciclo può essere "pompata": Possiamo saltare il ciclo ($i=0$) e la stringa è ancora accettata. Possiamo percorrere il ciclo una volta ($i=1$, la stringa originale). Possiamo percorrere il ciclo 2, 3, 4... volte ($i \geq 2$) e la stringa sarà sempre accettata.

Forma: $w = xyz$. Pompiamo la parte centrale y . xy^iz appartiene al Linguaggio Regolare



Se prendi una stringa w più lunga del numero di stati (n), l'automa deve per forza ripetere uno stato mentre la legge.

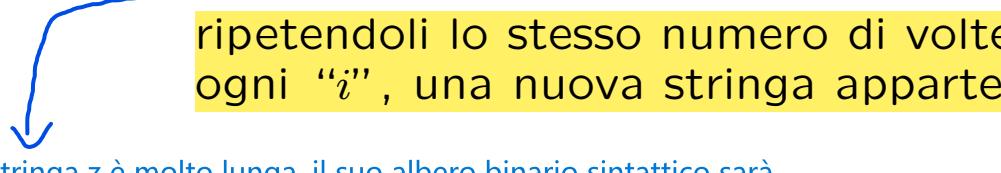
Pumping Lemma per CFL

Pumping Lemma per linguaggi regolari: per una stringa del linguaggio abbastanza lunga da causare un ciclo nel relativo DFA si può "ripetere" il ciclo e scoprire una infinita' di stringhe che appartengono al linguaggio

con n stati

Pumping Lemma per CFL (un po' piu' complicato): per una stringa del linguaggio sufficientemente lunga e' sempre possibile trovare due pezzi distinti da ripetere "in tandem":

ripetendoli lo stesso numero di volte " i ", otteniamo, per ogni " i ", una nuova stringa appartenente al linguaggio



Se una stringa z è molto lunga, il suo albero binario sintattico sarà molto profondo (alto) (binario perché qualsiasi grammatica è in CNF).

Se è abbastanza alto (più alto del numero di variabili), un qualche percorso dalla radice a una foglia dovrà ripetere almeno una variabile.



Questo crea una struttura ricorsiva nell'albero. La Pompa "in Tandem": Questa ripetizione ci permette di dividere la stringa z in 5 parti: $z = uvwxy$. A (la variabile ripetuta, in alto) genera $uvwxy$. A (in basso) genera la parte centrale w . La parte v è ciò che viene generato "scendendo" dal primo A al secondo A (sul lato sinistro). La parte x è ciò che viene generato "scendendo" dal primo A al secondo A (sul lato destro).⁸²

Cosa succede quando "pompiamo"? $i=0$ (non pompiamo): Possiamo "tagliare via" la parte ricorsiva. In pratica, sostituiamo la derivazione $A =^*> vwx$ con la derivazione più corta $A =^*> w$. La stringa che otteniamo è uw . E deve appartenere al linguaggio. $i=1$ (stringa originale): $uvwxy$. $i=2$ (pompiamo una volta): Incolliamo la parte ricorsiva dentro se stessa. Il primo A genera vAx , e quel A interno genera a sua volta vAx , che poi genera w . Il risultato è $\$uvv\ w\ xx\ y\$$, cioè uv^2wx^2y . E deve appartenere al linguaggio.

In generale: $\$uv^iwx^iy\$$ deve appartenere al linguaggio per ogni $i \geq 0$.

Perché "in Tandem"? Si chiama "in tandem" perché v e x sono generati insieme dalla stessa "cornice" ricorsiva ($A =^* vAx$). Non puoi pompare v senza pompare anche x (e viceversa). Devono essere ripetuti lo stesso numero di volte.

Condizioni Importanti:

- $|vx| > 0$ (o $|v| + |x| > 0$): Almeno una delle due sottostringhe v o x non deve essere vuota (altrimenti non staremmo pompando nulla).
- $|vwx| \leq n$: La parte che contiene la "ricorsione" (vwx) non può essere più lunga della costante di pumping lemma n . Questo ci assicura che v e x siano "vicini" tra loro.

Enunciato del Pumping Lemma per CFL

Pumping Lemma:

Sia L un CFL. Allora $\exists n \geq 1$ che soddisfa:

ogni $z \in L : |z| \geq n$ è scomponibile in 5 stringhe $z = uvwxy$ tali che:

1. $|vwx| \leq n$ la barchetta è più piccola della stringa z
2. $|vx| > 0$ uno dei due è non vuota
3. per ogni $i \geq 0$, $uv^iwx^i y \in L$ i mi serve per decidere se tagliare (i=0) o tornare all'originale (i=1) o pompare (i>=2)

Dimostrazione Pumping Lemma per CFL

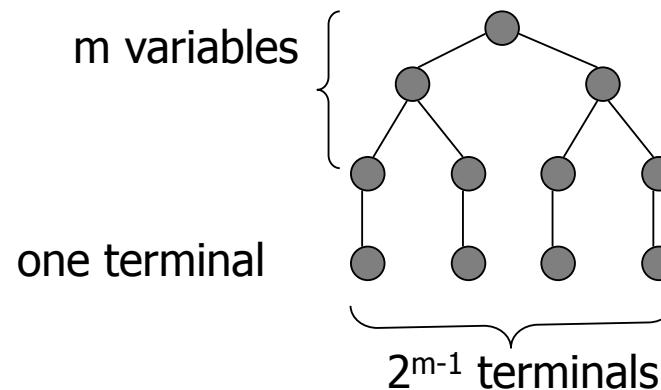
Prova:

- Si consideri una grammatica per $L \setminus \{\epsilon\}$ in CNF
- Assumiamo che la grammatica abbia m variabili. Sia $n = 2^m$
- Sia $z \in L$ una qualsiasi stringa tale che $|z| \geq n = 2^m$. Si ha che ogni albero sintattico di z contiene un cammino di lunghezza $\geq m + 1$ (Cammino da radice a foglia)

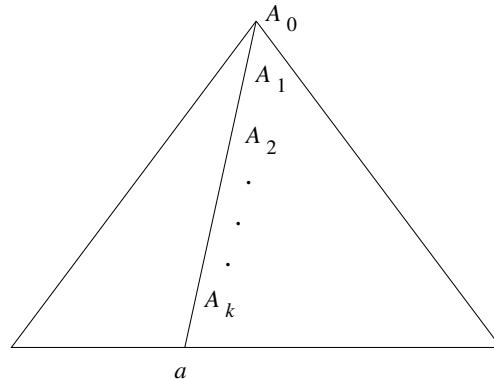
Lemma 1: Se tutti i cammini dell'albero sintattico hanno lunghezza $\leq m$, allora la stringa generata ha lunghezza $\leq 2^{m-1}$

quindi $z > n$ e quindi almeno una variabile si ripete nell'albero binario sintattico

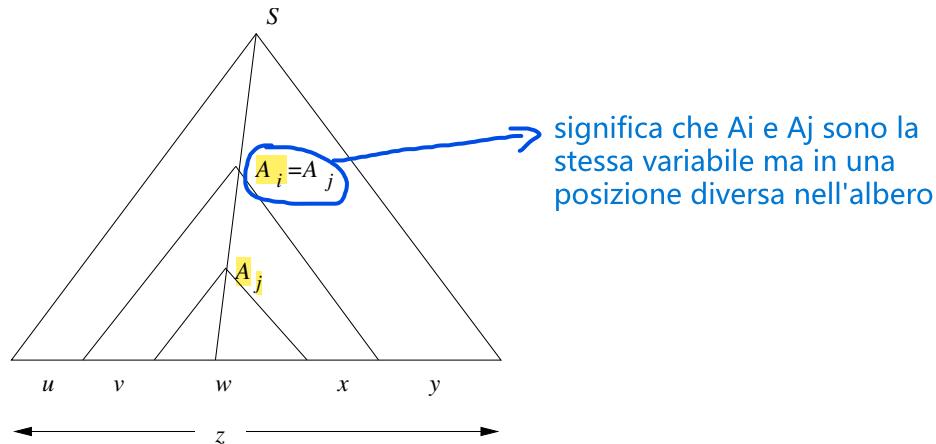
Prova:



- Consideriamo un cammino $\sqrt{A_0 A_1 \dots A_k a}$ di lunghezza massima:
ha lunghezza $\geq m + 1$.



Esistono $i \neq j$ tali che $A_i = A_j$ (assumiamo che i, j siano fra le **ultime** $m + 1$ variabili del cammino)



stringa con lunghezza $2^{(m-1)}$ allora cammino è lungo m.
 stringa con lunghezza 2^m allora cammino è lungo m+1

$n = 2^m$, così sono sicuri
 che c'è la ripetizione di
 almeno una variabile.

Se il cammino è lungo quanto il numero delle variabili,
 allora numero delle variabili nel cammino è m+1

Ai è un sottoalbero dell'albero binario sintattico o l'albero binario
 sintattico (che ha altezza m+1), quindi Ai ha altezza $\leq m+1$

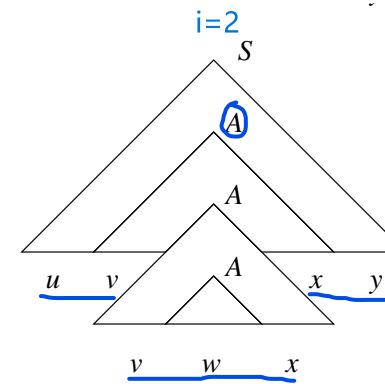
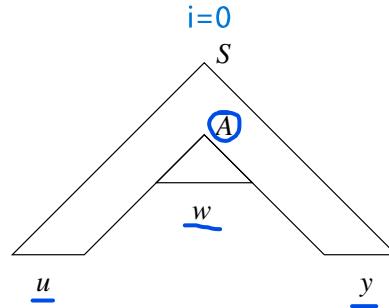
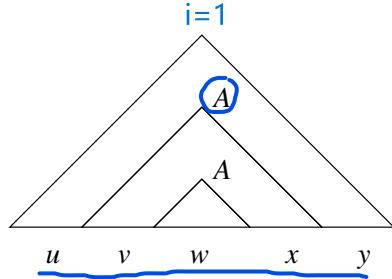
• **Osservazione 1:** l'albero radicato in A_i ha altezza $\leq m+1$,
 quindi la stringa corrispondente ha lunghezza $\leq 2^m = n$ (cioè'
 $|vwx| \leq n$)

• **Osservazione 2:** le stringhe v e x non possono essere entrambe
 vuote in quanto A_i (essendo la grammatica in CNF) genera due
 variabili entrambe non annullabili (quindi $|vx| > 0$)

• **Osservazione 3:** l'albero sintattico ottenuto ripetendo un nu-
 mero arbitrario di volte (possibilmente anche 0 volte) la parte di al-
 bero radicato in A_i meno l'albero radicato in A_j , continua ad essere
 un albero sintattico corretto (quindi per ogni $i \geq 0$, $uv^iwx^iy \in L$)

Questo perché
 non ci sono
 produzioni
 epsilon, quindi se
 ho 2 figli o uno
 solo (non posso
 non averli)

Ai - Aj ho ancora un albero sintattico



Se si pensa che il Linguaggio non sia libero, dimostrarlo con Pumping Lemma
Se si pensa che il Linguaggio sia libero, creare un PDA o Grammatica del linguaggio

Applicazioni del Pumping Lemma per CFL

Come per i linguaggi regolari, il Pumping Lemma per CFL puo' essere usato per dimostrare che un dato linguaggio non e' libero

Esempio 1: Si consideri $L = \{0^m 1 0^m 1 0^m : m \geq 1\}$. Dimostrare che L non e' un CFL.

Prova: Assumiamo, per assurdo, che L sia CFL. Sia n la costante del Pumping Lemma. Si consideri la stringa $z = 0^n 1 0^n 1 0^n$. Si ha che $z \in L$ e $|z| \geq n$. Allora, per Pumping Lemma, $z = uvwxy$ con $|vwx| \leq n$, $|vx| > 0$ e $uv^iwx^i y \in L$ per ogni $i \geq 0$. Consideriamo ora i due seguenti casi:

- vx contiene almeno un 1.

massimo

In questo caso $uwy \notin L$ visto che ha al piu' un solo 1 (caso in cui taglio, cioè $i=0$)

- vx contiene solo 0.

Ci sono solo due casi. O vx include 0 tutti appartenenti ad uno stesso gruppo di 0 oppure v appartiene ad un gruppo ed x ad un altro.

In entrambi i casi $uwy \notin L$ in quanto almeno un gruppo di 0 mantiene lunghezza n ed un'altro si riduce a lunghezza $< n$ (caso in cui taglio, cioè $i=0$)

Esempio 2: Si consideri $L = \{0^{k^2} : k \geq 1\}$. Dimostrare che L non e' un CFL.

Prova: Assumiamo, per assurdo, che L sia CFL. Sia n la costante del Pumping Lemma. Si consideri la stringa $z = 0^{n^2}$. Si ha che $z \in L$ e $|z| \geq n$. Allora, per Pumping Lemma, $z = uvwxy$ con $|vwx| \leq n$, $|vx| > 0$ e $uv^iwx^i y \in L$ per ogni $i \geq 0$. Consideriamo ora il seguente caso:

- $uv^2wx^2y \in L$ per Pumping Lemma;
 $\rightarrow |uvwxy| + |vx| \leq n^2 + n$ (perché la lunghezza di vx è minore o uguale della lunghezza della barchetta vwx)
- $n^2 < |uv^2wx^2y| \leq n^2 + n < n^2 + 2n + 1 = (n+1)^2$. Non essendoci quadrati perfetti strettamente inclusi fra n^2 e $(n+1)^2$ allora $uv^2wx^2y \notin L$, contraddicendo il punto precedente.
 \rightarrow non appartiene al linguaggio, essendo che tra n^2 e $(n+1)^2$ non ci sono quadrati

- Unione: da G1 e G2, trovare L1uL2. Creo una nuova variabile S (nuova variabile iniziale) che ha come Produzioni S1 e S2 delle due grammatiche ($S \rightarrow S_1|S_2; S_1 \rightarrow \dots; S_2 \rightarrow \dots$).
- Concatenazione: da G1 e G2, trovare L1L2. Creo una nuova variabile S (nuova variabile iniziale) che ha come Produzione S1S2 delle due grammatiche ($S \rightarrow S_1S_2; S_1 \rightarrow \dots; S_2 \rightarrow \dots$).
- Stella di Clini: da G1, trovare L1*. Creo una nuova variabile S (nuova variabile iniziale) che ha come Produzioni S1S ed epsilon della grammatica ($S \rightarrow S_1S|\epsilon; S_1 \rightarrow \dots$).
- Chiusura positiva: da G1, trovare L1+. Creo una nuova variabile S (nuova variabile iniziale) che ha come Produzioni S1S ed S1 della grammatica ($S \rightarrow S_1S|S_1; S_1 \rightarrow \dots$).

Chiusura Operazione dei CFL: i risultati delle operazioni sono ancora dei Linguaggi Liberi dal Contesto

Proprieta' di chiusura dei CFL

Teorema 7.24: I CFL sono chiusi rispetto ai seguenti operatori
 (i) : unione, (ii) : concatenazione e (iii) : chiusura di Kleene e
 chiusura positiva +

Prova: Per esercizio.

Teorema: Se L è CFL, allora lo è anche L^R .

Prova: Supponiamo che L sia generato da $G = (V, T, P, S)$. Costruiamo $G^R = (V, T, P^R, S)$, dove

$$P^R = \{A \rightarrow \alpha^R : A \rightarrow \alpha \in P\} \quad \text{Per ogni produzione, il corpo viene invertito (P^R)}$$

Si mostra per induzione sulla lunghezza delle derivazioni in G e in G^R che $(L(G))^R = L(G^R)$. cioè le grammatiche sono equivalenti e quindi L^R è un CFL

I CFL non sono chiusi rispetto all'intersezione

Sia $L_1 = \{0^n 1^n 2^i : n \geq 1, i \geq 1\}$. Allora L_1 e' CFL con grammatica

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow 0A1|01 \\ B &\rightarrow 2B|2 \end{aligned}$$

Inoltre, $L_2 = \{0^i 1^n 2^n : n \geq 1, i \geq 1\}$ e' CFL con grammatica

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow 0A|0 \\ B &\rightarrow 1B2|12 \end{aligned}$$

Invece, $L_1 \cap L_2 = \{0^n 1^n 2^n : n \geq 1\}$ non e' CFL (dimostrazione tramite Pumping Lemma per esercizio).

L'intersezione di un Libero con un Regolare, mi da un Libero

Operazioni su liberi e regolari

Teorema 7.27: Se L e' CFL, e R e' regolare, allora $L \cap R$ e' CFL.

Tramite Simulazione Parallelia (accetto quando accettano entrambi)

Prova: Sia L accettato dal PDA

Simulazione Parallelia: l'input viene fatto passare contemporaneamente sia al FA sia al PDA

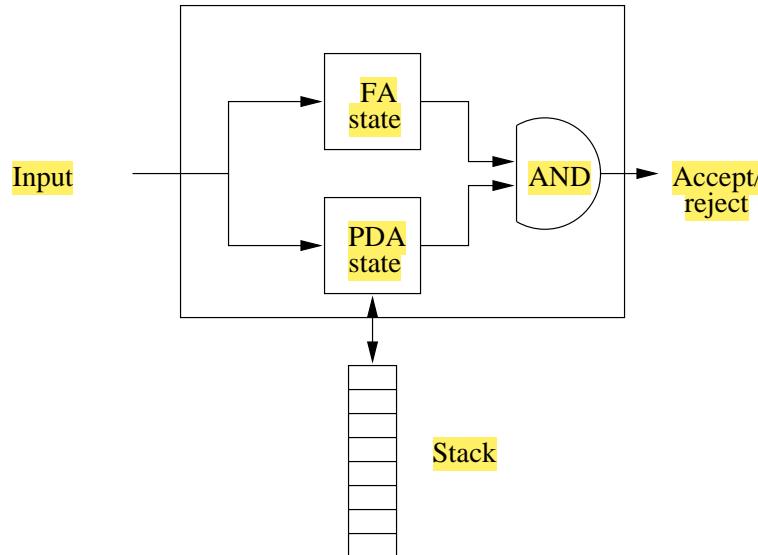
$$P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$$

per stato finale, e sia R accettato dal DFA

$$A = (Q_A, \Sigma, \delta_A, q_A, F_A)$$

Costruiremo un PDA per $L \cap R$ secondo la figura

Ciò che metto nella pila è lo stesso che metto quando ho solo il PDA



Formalmente, definiamo

Prodotto cartesiano dei due insieme di stati

$$P' = (\overline{Q_P \times Q_A}, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, \overline{F_P \times F_A})$$

Prodotto cartesiano dei
due insieme di stati finali

dove

Funzione di
transizione per
i singoli simboli

$$\delta((q, p), a, X) = \{((r, \widehat{\delta}_A(p, a)), \gamma) : (r, \gamma) \in \delta_P(q, a, X)\}$$

indica, per DFA, sia transizione
epsilon sia transizione con simbolo

il PDA P "comanda" la computazione e
le operazioni sulla stack, mentre il DFA
A "segue" passivamente, aggiornando il
suo stato ad ogni passo.

Possiamo provare per induzione su \vdash^* che

$$(q_P, w, Z_0) \stackrel{*}{\vdash} (q, \epsilon, \gamma) \text{ in } P$$

se e solo se

$$((q_P, q_A), w, Z_0) \stackrel{*}{\vdash} ((q, \widehat{\delta}(q_A, w)), \epsilon, \gamma) \text{ in } P'$$

Teorema 7.29: Siano L, L_1, L_2 CFL e R regolare. Allora

1. $L \setminus R$ e' CFL Differenza tra libero e regolare, é un libero
2. \bar{L} non e' necessariamente CFL Il complementare di un Libero, non é necessariamente un libero
3. $L_1 \setminus L_2$ non e' necessariamente CFL Differenza tra due liberi, non é necessariamente un libero

Prova:

1. \bar{R} e' regolare, $L \cap \bar{R}$ e' CFL, e $L \cap \bar{R} = L \setminus R$.
2. Se \bar{L} fosse sempre CFL, seguirebbe che

É sia Libero, sia Regolare

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Se L complementare fosse un libero, allora anche intersezione é libera (ASSURDO)

- sarebbe sempre CFL.
3. Notare che Σ^* e' CFL, quindi se $L_1 \setminus L_2$ fosse sempre CFL, allora lo sarebbe sempre anche $\Sigma^* \setminus L = \bar{L}$. (ASSURDO)

Conseguenza diretta della complementarietà.

Proprieta' di decisione per CFL

Analizzeremo i seguenti problemi decidibili:

- 1 • Verificare se $L(G) \neq \emptyset$, per una CFG G
- 2 • Verificare se $w \in L(G)$, per una stringa w ed una CFG G
senza questo, non saremmo in grado di fare parsing

E elencheremo alcuni **problemi indecidibili**

Ad ogni passata, valuto le n produzioni con variabili generanti e quindi rivaluto tutte le n produzioni con variabili generanti.

1

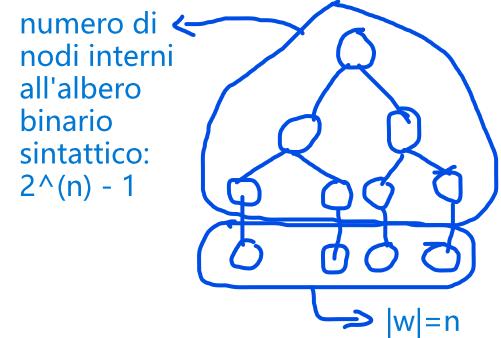
Verificare se un CFL e' vuoto

$L(G)$ e' non-vuoto se il simbolo iniziale S e' generante

senza ottimizzazioni

Una implementazione naïve del calcolo dei simboli generanti di G richiede tempo $O(n^2)$

Ottimizzando le strutture dati di appoggio può essere calcolato in tempo $O(n)$.



la stringa è generabile dalla grammatica?

2

$w \in L(G) ?$

Tecnica inefficiente:

Supponiamo che G sia in CNF, e che la stringa w abbia lunghezza $|w| = n$. Visto che il suo albero sintattico è binario, ci sono $2n - 1$ nodi interni

Basta quindi generare tutti gli alberi sintattici di G con $2n - 1$ nodi interni, e poi controllare se almeno uno genera w

Numero degli alberi/etichettature possibili (complessità algoritmo) quindi esponenziale in n .

Esiste una tecnica che lo fa in tempo polinomiale

Problemi indecidibili per CFL

I seguenti problemi sono indecidibili:

1. Una data **CFG** G e' ambigua?



Non è possibile creare un algoritmo che, data una grammatica G qualsiasi, determini se essa è ambigua. Questo è problematico perché l'ambiguità rende difficile la corretta interpretazione delle istruzioni da parte di un compilatore

2. Un dato **CFL** L e' inherentemente ambiguo?



Se è difficile determinare se una singola grammatica è ambigua (1), è ancora più difficile determinare se tutte le infinite possibili grammatiche per quel linguaggio sono ambigue. È un problema di indecidibilità di ordine superiore.

3. L'intersezione di due CFL e' vuota?

4. Due CFL sono uguali? → cioè grammatiche equivalenti (non esiste un algoritmo generale che lo risolve)

5. Un CFL e' universale (cioe' uguale a Σ^*)?

Verificare se un PDA accetta tutto ciò che può essere digitato è impossibile per un algoritmo generale. Se fosse decidibile, potremmo usarlo per risolvere il punto 3 e 4.

