GRAMMATICHE LIBERE DAL CONTESTO:

Dimostrazione che Lpal NON è regolare col pumping lemma:

n è la costante del pumping lemma.

Consideriamo la stringa

w = 0^n 1 0^n

Se Lpal fosse regolare, possiamo dividere w = xyz, con y che ha uno o più zeri del primo gruppo.

Il pumping lemma dice che, se Lpal è regolare, allora anche xz appartiene a Lpal, ma xz NON appartiene a Lpal perché ci sono meno zeri a sinistra dell' 1 che a destra.

Lpal può essere definito induttivamente:

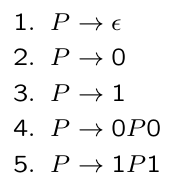
**Base:**

epsilon, o e 1 sono palindromi

**Induzione:**

se w è palindroma, allora anche 0w0 e 1w1 è palindroma.

Una CFG è una notazione formale per definire le definizioni ricorsive dei linguaggi.



Una CFG è definita da una quadrupla:

G = (V, T, P, S)

Dove:

V è l'insieme delle variabili (es. A,E,P,...)

T è l'insieme dei terminali (es. 0,1,...)

P è l'insieme delle produzioni della forma A → α, dove A e’

una variabile e α ∈ (V ∪ T )∗

S è una variabile chiamata simbolo iniziale

------------------

Esempio: rappresentare le espressioni di un linguaggio di programmazione, che permette solo il + e il \*, e i nomi delle variabili iniziano per a|b e sono seguiti da tanti a|b|0|1.

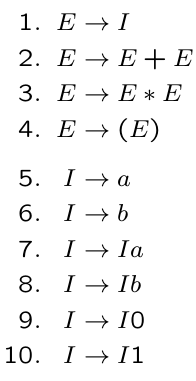
Per questa grammatica abbiamo bisogno di 2 variabili: E, I.

E rappresenta le espressioni (+, \*)

I rappresenta il linguaggio degli identificatore, che è un linguaggio regolare:

(a + b)(a + b + 0 + 1)\*

Però nelle grammatiche non si usano le espressioni regolari, ma un insieme di produzioni che dicono la stessa cosa:



La definizione formale di questa grammatica è:

V = {E, I}

T = {a,b,0,1,+,\*,(,)}

P = le produzioni di prima, da 1 a 10

S = E

DERIVAZIONI

Sia G = (V, T, P, S) una CFG, A ∈ V ,

{α, β} ⊂ (V ∪ T )∗ , e A → γ ∈ P .

αAβ ⇒ αγβ

Diciamo che αAβ si deriva αγβ.

Definiamo ⇒\* la chiusura riflessiva e transitiva di ⇒, cioe’:

**Base:**

Sia α ∈ (V ∪ T )∗ . Allora α ⇒\* α.

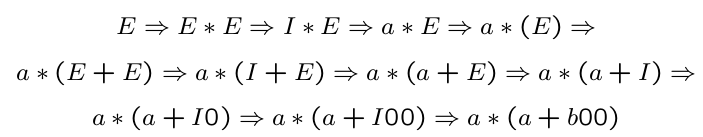
(produce se stessa)

**Induzione:**

Se α ⇒\* β, e β ⇒ γ, allora α ⇒\* γ.

Esempio di una derivazione dalla grammatica delle espressioni definita prima:

a ∗ (a + b00)



Ad ogni passo possiamo avere più regole tra cui scegliere, es:



Però non tutte le regole ci portano alla derivazione di prima, es . La regola:



DERIVAZIONI A SINISTRA (LEFT-MOST) O A DESTRA (RIGHT-MOST):

Derivazione a sinistra ⇒lm :

rimpiazza sempre la variabile piu’ a sinistra con il corpo di una delle sue regole.

Derivazione a destra ⇒rm :

rimpiazza sempre la variabile piu’ a destra con il corpo di una delle sue regole.

IL LINGUAGGIO DI UNA GRAMMATICA

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

L(G) = {w ∈ T ∗ : S ⇒ w}

cioe’ l’insieme delle stringhe su T∗ (composte da soli terminali) derivabili dal simbolo iniziale.

Se G è una CFG, allora L(G) è un linguaggio libero dal contesto.

FORME SENTENZIALI:

Sia G = (V, T, P, S) una CFG, e α ∈ (V ∪ T )∗ .

Se S ⇒ α, α e’ una forma sentenziale (è una derivazione formata sia da variabili che da terminali).

L(G) contiene le forme sentenziali che sono in T∗

Le forme sentenziali possono essere anche derivazioni NE' a sinistra NE' a destra, ad esempio se si deriva una variabile nel mezzo.

ALBERI SINTATTICI:

Se w ∈ L(G), per una CFG, allora w ha un albero sintattico, che

ci dice la struttura (sintattica) di w.

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), cioe’ il linguaggio dovrebbe essere non ambiguo.

Sfortunatamente, non sempre possiamo rimuovere l’ambiguita’.

COSTRUZIONE DI UN ALBERO SINTATTICO:

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

1. Ogni nodo interno e’ etichettato con una variabile in V .

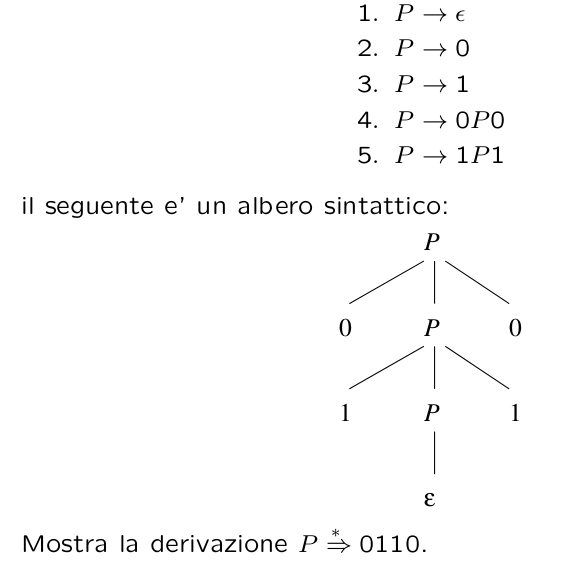
2. Ogni foglia e’ etichettata con un simbolo in V ∪ T ∪ { } (epsilon). Ogni foglia etichettata con epsilon e’ l’unico figlio del suo genitore.

3. Se un nodo interno e’ etichettato A, e i suoi figli (da sinistra a destra) sono etichettati

X 1 , X 2 , . . . , X k ,

allora A → X 1 X 2 . . . X k ∈ P . (è una produzione)

ESEMPI ALBERI SINTATTICI:



PRODOTTO DI UN ALBERO SINTATTICO:

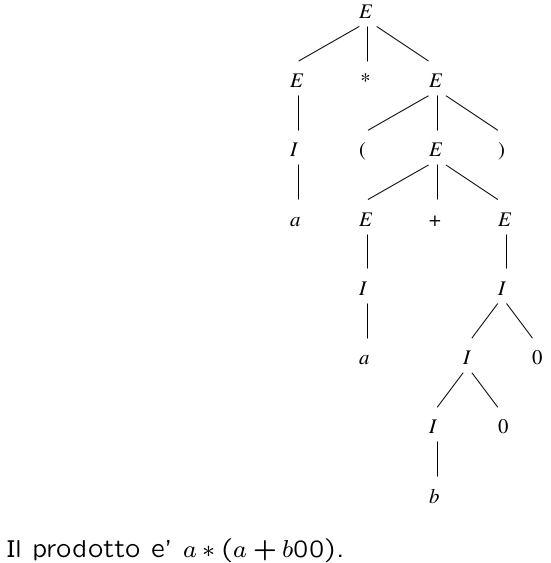
Il prodotto di un albero sintattico e’ la stringa di foglie da sinistra a destra.

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.



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

1. A ⇒ w

2. A ⇒lm w, e A ⇒rm w

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

Per provare l’equivalenza, usiamo il seguente piano:

• Dagli alberi alle derivazioni a sinistra (destra): visito l’albero da sinistra a destra (da destra a sinistra)

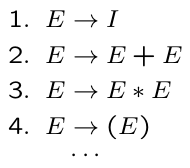
• Una derivazione sinistra (o destra) è anche una derivazione

• Leggendo la derivazione costruisco l’albero

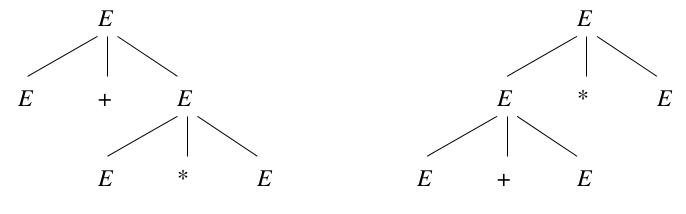
(guarda esempi pag 19-20 parte2.pdf)

AMBIGUITÀ IN GRAMMATICHE E LINGUAGGI

Ad esempio, nella grammatica che ha queste produzioni:

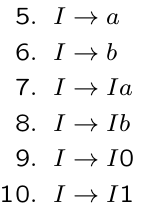


La forma sentenziale E + E \* E ha 2 derivazioni diverse, quindi ci sono 2 alberi sintattici diversi:

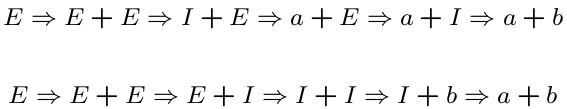


Per una stessa forma sentenziale, non è pericolosa l'esistenza di più derivazioni, è pericolosa l'esistenza di più alberi sintattici, che cambiano il significato dell'espresione.

Ad esempio, nella grammatica delle operazioni di un linguaggio di programmazione:



La stringa a+b può avere molte derivazioni, ma l'albero sintattico è lo stesso:



(Prova a disegnare i 2 alberi sintattici)

**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 al piu’ un albero sintattico, G e’ detta non-ambigua.

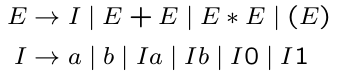
RIMUOVERE L'AMBIGUITÀ DALLE GRAMMATICHE:

Buone notizie: a volte possiamo rimuovere l’ambiguita’

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

Ancora cattive notizie: alcuni CFL hanno solo CFG ambigue

Ad esempio, sempre nella grammatica di prima:



Non c'è la precedenza tra gli operatori, in questo caso il \* e il +.

Come si raggruppano le espressioni?

E + E + E e’ inteso come E + (E + E) o come (E + E) + E?

SOLUZIONE:

Si introducono nuove variabili, che rappresentano le stesse espressioni con lo stesso grado di coesione.

1) Un **fattore** è un'espressione che NON può essere spezzata ne da un + ne da un \*.

Nella grammatica sopra i nostri fattori sono:

- gli identificatori (i nomi delle variabili)

- un'espressione racchiusa tra parentesi

2) un **termine** è un espressione che NON può essere spezzata da un + (ne precedente ne seguente), ma solo se preceduta da un \* (con associatività a sinistra).

Esempio:

Il termine a\*b può essere spezzato se preceduto da a1\*, in questo caso per l'associatività a sinistra diventa:

(a1\*a)\*b

a\*b \*a1 NON veniva spezzato, diventava (a\*b)\*a1

Se invece era preceduto da un + non veniva spezzato:

a1 + (a\*b), (a\*b) + a1

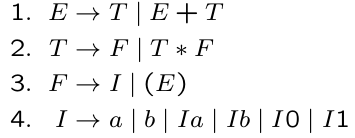
3) il resto sono espressioni che possono essere spezzate se un + precede l'espressione, o da un \* (sia precedente che seguente):

a + b può essere spezzato se:

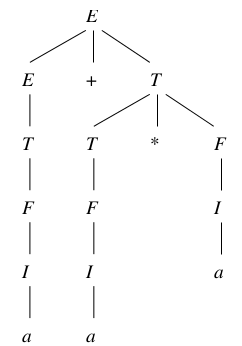
(c + a) + b (precede un +)

a + (b \* c) (segue un \*)

Le nuove produzioni diventano quindi queste:



Con queste nuove regole, la derivazione a + a ∗ a ha solo un albero sintattico:

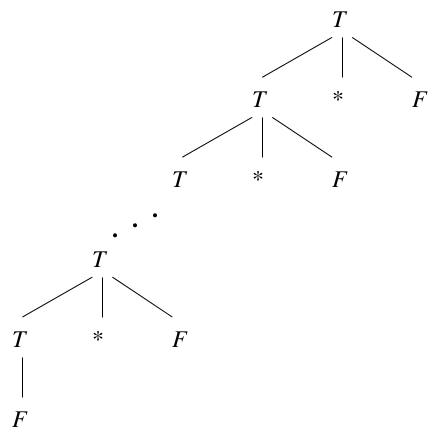


L'albero sarebbe:

a + T, il valore in T è calcolato in modo univoco come a \* a.

PERCHÈ LA GRAMMATICA NON È AMBIGUA?

Perchè ha associatività a sinistra, ad esempio per i termini l'albero sintattico diventa:



In generale:

• Un albero sintattico, ha molte derivazioni (esempio di a+b visto prima)

• Molte derivazioni a sinistra implica molti alberi sintattici.

• Molte derivazioni a destra implica molti alberi sintattici.

(vedi teorema sotto)

**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.

AMBIGUITÀ INERENTE

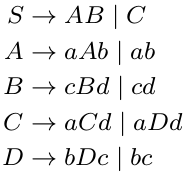
Un CFL L e’ inerentemente ambiguo se tutte le grammatiche per

L sono ambigue. Esempio:

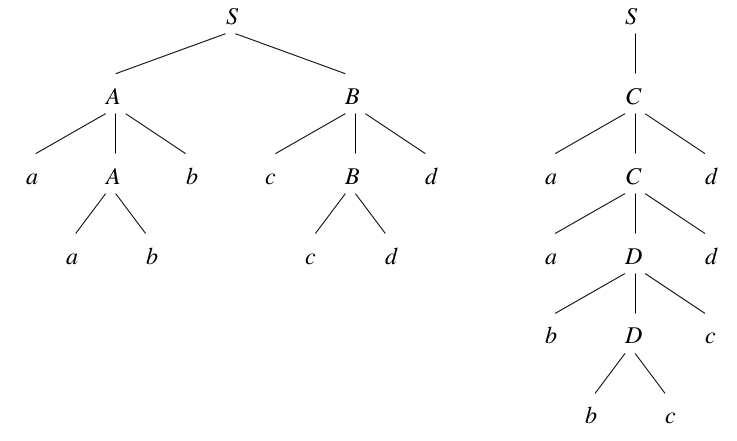
L=



Una grammatica per L è:



Ad esempio, la stringa aabbccdd ha 2 alberi sintattici:



Può essere provato che qualsiasi grammatica per questo linguaggio è ambigua, quindi il linguaggio è **inerentemente ambiguo.**

**AUTOMI A PILA:**

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

In una transizione un PDA:

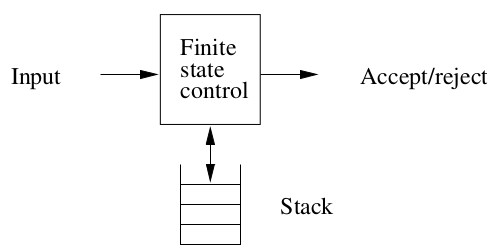
1. Consuma un simbolo di input.

2. Va in un nuovo stato (o rimane dove e’).

3. Rimpiazza il top della pila con una stringa

(non fa niente, o elimina il top della pila, o mette una stringa

in cima alla pila)



ESEMPIO:

Consideriamo il linguaggio dei palindromi nell'alfabeto {0,1}, con questa grammatica:

P → 0P0,

P → 1P1,

P → epsilon

Un PDA per questa grammatica ha 3 stati e funziona come segue:

1. Scommette che sta leggendo w. Rimane nello stato 0, e mette

il simbolo di input sulla pila.

2. Scommette che sta nel mezzo di wwR . Va spontaneamente

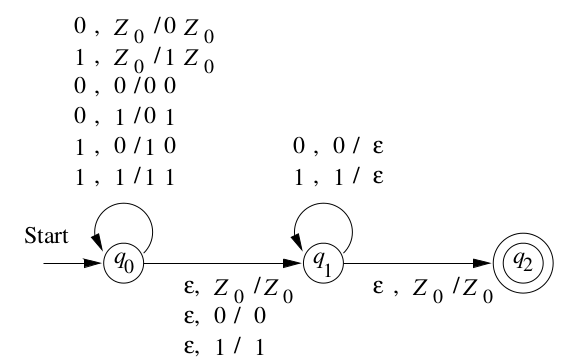
nello stato 1.

3. Sta leggendo la testa di wR . La paragona al top della pila. Se

sono uguali, fa un pop della pila, e rimane nello stato 1. Se

non sono uguali, si ferma.

4. Se la pila e’ vuota, va nello stato 2 e accetta la stringa.



DEFINIZIONE FORMALE DI PDA:

P = (Q, Σ, Γ, δ, 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 : Q × Σ ∪ { } × Γ → 2 Q×Γ e’ la funzione di transizione,

• q 0 e’ lo stato iniziale,

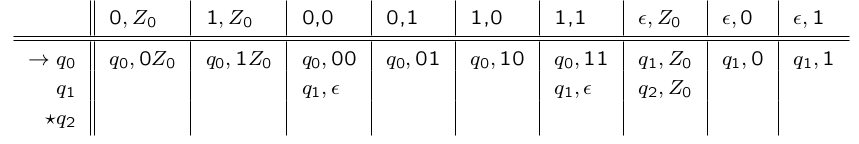
• Z 0 ∈ Γ e’ il simbolo iniziale per la pila, e

• F ⊆ Q e’ l’insieme di stati di accettazione.

Ad esempio, il PDA di prima è la 7-upla:

P = ({q0 , q1 , q2 }, {0, 1}, {0, 1, Z0 }, δ, q0 , Z0 , {q2})

dove δ e’ data dalla tavola seguente:



La funzione di transizione prende in input 3 parametri:

- lo stato attuale del PDA

- il prossimo carattere della stringa in input

- la cima dello stack

In output restituisce:

un insieme di coppie (può essere non deterministico, è come un E\_NFA), formate da:

- lo stato di destinazione del PDA

- Quello che scrive al posto dello stack letto (epsilon, la stessa cosa, o un'altra cosa)

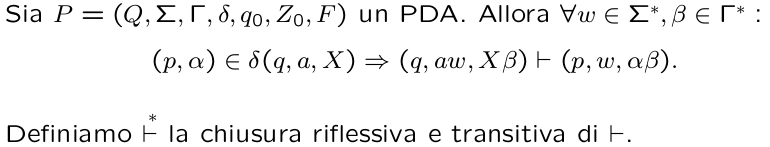
DESCRIZIONI ISTANTANEE

Negli automi a stati finiti, una descrizione istantanea era semplicemente il suo stato, quindi la funzione di transizione applicata alle stringhe definiva una sequenza di descrizioni istantanee, atraverso le quali il DFA si muoveva consumando la stringa in input.

Per un PDA, una descrizione istantanea (ID) è una tripla:

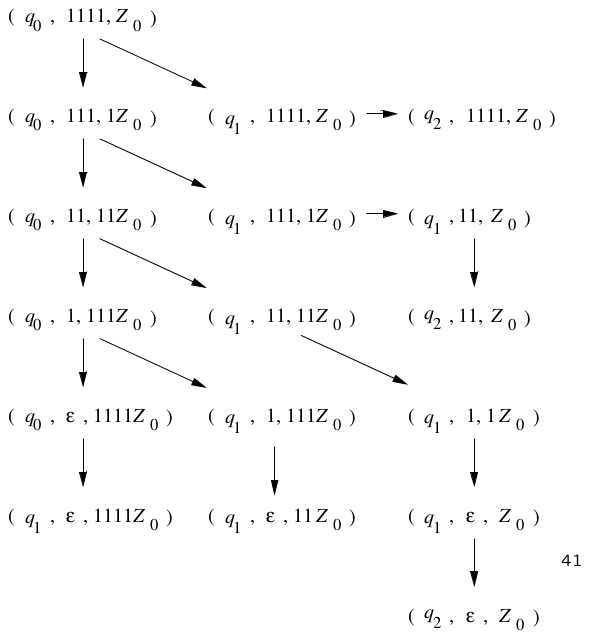
(q, w, γ)

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



Con lo \* si indicano zero o più mosse del PDA con quel particolare input e quel particolare stack.

Ad esempio, l'automa di prima, su input 1111 ha le seguenti computazioni:



PROPRIETÀ DELLE DESCRIZIONI ISTANTANEE:

(Sono computazioni legali, non significa che vadano in uno stato accettante, ma che è possibile fare la tale sequenza di ID).

1) Se una sequenza di ID e’ una computazione legale per un

PDA, allora lo e’ anche la sequenza ottenuta aggiungendo

una stringa alla fine della seconda componente (input).

2) Se una sequenza di ID e’ una computazione legale per un

PDA, allora lo e’ anche la sequenza ottenuta aggiungendo

una stringa alla fine della terza componente (alla fine dello stack).

3)Se una sequenza di ID e’ una computazione legale per un PDA,

e la coda di un input non e’ consumata, allora rimuovendola

da tutte le ID si ottiene una computazione lecita.

**Teorema 6.5:**





Se w = epsilon, allora siamo nella proprietà (2)

Se gamma = epsilon, allora siamo nella proprietà (1)

ACCETTAZIONE PER STATO FINALE:

Non è necessario che anche lo stack sia vuoto, basta solo che tutto l'input sia stato consumato dal PDA e che il Pda si trovi in uno stato finale.

Il linguaggio accettato da P per stato finale è:



ACCETTAZIONE PER PILA VUOTA:

NON è necessario che il PDA si trovi in uno stato finale, basta che tutto l'input sia consumato e che lo stack sia vuoto.

Il linguaggio accettato da P per pila vuota è:



Per rendere il PDA di prima accettante per pila vuota, basta cambiare l'ultima transazione, quella che passa da q1 a q2, senza riscrivere Z0 nello stack, lasciando così solo epsilon nello stack.

In questo caso, il PDA è definito da una 6-upla, in quanto gli stati finali non ci sono in questo tipo di PDA.

**Da quello grande a quello piccolo è facile, es:**

**da stack vuoto a stato finale:**

**vuoto è piccolo, finale è grande**

**quindi:**

**L(F) C= L(E) è facile, cioè**

**se w € L(F) allora w € L(E)**

**perché devo passare per stato iniziale e finale del PDA E (stack vuoto), fuori non consumo input ulteriore.**

**Invece L(E) C= L(F) è da dimostrare con le ID**

**hp: se w € L(E) => w € L(F)**

**si fanno le ID che in zero o più passi portanno dallo stato iniziale del PDA E(stack vuoto) a un suo stato in cui accetta.**

**Per il teorema delle ID legali, sotto allo stack si possono aggiungere simboli e ottenere sequenze legali.**

**Per hp, anche la funzione di transizione di E C= f.d.t. di F, quindi la stessa sequenza di ID si ha per il PDA F (stato finale)**

**Adesso ci va la prima ID di F, che porta dal suo stato iniziale alle ID che avevi già, e anche per lo stato finale.**

DA STACK VUOTO AD ACCETTAZIONE PER STATO FINALE:

Bisogna dimostrare che il linguaggio accettato dal PDA che accetta stringhe per stack vuoto è lo stesso del PDA che accetta stringhe per stato finale (sempre consumando tutto l'input).

L = linguaggio

N(P) = linguaggio accettato da un PDA P per stack vuoto

L(P) = Linguaggio accettato da un PDA P per stato finale

PN = PDA che accetta L per stack vuoto

PF = PDA che accetta L per stato finale

**Teorema 6.9:**

Se L = N(PN) per un PDA PN = (Q, Σ, Γ, δN , q0 , Z0),

allora ∃ PDA PF , tale che L = L(P F ).

**Prova:**

Bisogna usare un nuovo simbolo X0, da mettere in fondo alo stack, così che non rimane vuoto.

Così, quando il PDA PF vede X0 in cima al suo stack, sa che con lo stesso input il PDA PN vrebbe finito il suo input e l'avrebbe accettato.

Inoltre bisogna aggiungere un nuovo stato iniziale p0, la cui sola funzione è quella di mettere in cima allo stack Z0, il simbolo di partenza di PN.

Inoltre c'è bisogno di un nuovo stato finale pf, che riceve gli archi di tutti gli stati di PN che leggono, per un certo input, X0 in cima allo stack.

PF è definito come:

PF = (Q ∪ {p0 , pf}, Σ, Γ∪ {X0}, δF , p0 , X0 , {pf })

dove δF è definita così:

1) δF (p0 , epsilon , X0) = {(q0 , Z0 X0 )}

(per lo stato iniziale)

2) Per ogni q ∈ Q, a ∈ Σ ∪ {epsilon}, Y ∈ Γ,

δF (q, a, Y) = δN (q, a, Y)

(contiene le stesse sequenza di PN)

3) (pf , epsilon) ∈ δF (q, epsilon , X0)

(tutti gli stati che leggono X0 vanno nello stato finale perché PN avrebbe finito l'input e l'avrebbe accettato)

**Dimostrazione:** (se e solo se)

**(Se): (**PF accetta w per stato finale?**)**

Per PN, supponiamo

****

Il teorema 6.5 ci dice che che possiamo aggiungere X0 in fondo allo stack e concludere che:

****

Dato che tutte le sequenze di PN sono anche in PF (vedi regola 2 di prima della prova) allora si può dire che:

****

(praticamente cambia solo che scrivi PF sotto al simbolo |-)

Infine se si aggiungono anche lo stato iniziale e finale di PF si ottiene:

****

Quindi PF accetta w per stato finale.

**(solo se): (**PN accetta w per stack vuoto?**)**

La sequenza di PF rimane come quella mostrata nella pare (se) del teorema.

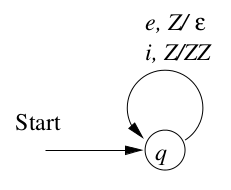


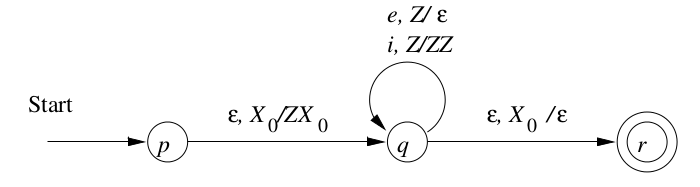
Dato che, a parte lo stato iniziale e lo stato finale di PF, PF esegue le stesse computazioni di PN con l'aggiunta di X0 in fondo allo stack, possiamo concludere che:



Quindi, w appartiene anche a N(PN).

-------------  
  
ESEMPIO:





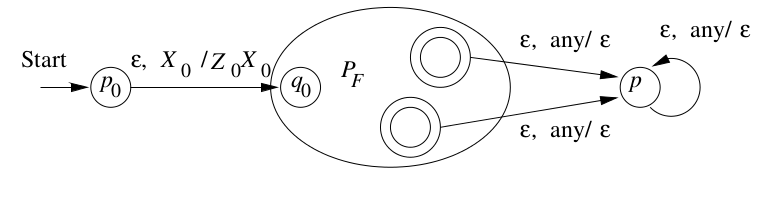
DA STATO FINALE AD ACCETTAZIONE PER PILA VUOTA:

La costruzione di un PDA che accetta lo stesso linguaggio di un PDA PF, richede queste modifiche:

Da ogni stato finale di PF, si aggiunge una transizione su input epsilon in un nuovo stato p.

Quando PN è in p, svuota tutto lo stack rimanente, sempre su input epsilon, così accetta la stessa stringa di PF.

Per evitare che PF svuoti tutto lo stack senza raggiungere uno stato finale, si aggiunge un nuovo simbolo di stack X0 che non fa parte dei simboli di stack di PF, così in tale situazione PF si blocca con X0 nello stack, e così farà anche PN.



**Teorema 6.11:**

Sia L = L(PF), per un PDA PF = (Q, Σ, Γ, δF, q0, Z0, F).

Allora ∃ PDA PN, tale che L = N (PN).

**Prova:**

Sia

PN = (Q ∪ {p0 , p}, Σ, Γ ∪ {X0}, δN , p0, X0)

dove δN è definita da:

1) δN (p0 , epsilon , X0) = {(q0 , Z0 X0)}

(lo stato iniziale del PDA PN, aggiunge il simbolo Z0 di PF in cima allo stack)

2) per ogni q ∈ Q, a ∈ Σ ∪ { epsilon }, Y ∈ Γ,

δN (q, a, Y) = δF (q, a, Y)

(cioè δN contiene tutte le coppie di δF)

3) Per ogni stato accettante q ∈ F , e Y ∈ Γ ∪ {X 0 }

(p, epsilon) ∈ δN(q, epsilon , Y)

(cioè, quando PN si trova in uno stato accettante di PF, cambia stato e va nel suo stato finale, dove nel punto 4 svuota tutto il suo stack)

4) per Y ∈ Γ ∪ {X0}

δN (p, epsilon , Y) = {(p, epsilon)}

(svuota tutto lo stack rimanendo sempre nello stato p)

La dimostrazione del teorema è un se e solo se, la parte se è una simulazione, la parte solo se richiede di esaminare la costruzione del PDA PN.

**(Se): (**w è in L(PN)?**)**

Se w è in L(PF), per ogni q ∈ F, α ∈ Γ∗:



(cioè va in uno stato finale di PF con i rimanenti simboli α nello stack).

Per il teorema 6.5, il simbolo X0 può essere messo in fondo allo stack, e si ottiene una sequenza legale di PN:



Quindi ogni computazione accettante di PN è data da:



Quindi w è in L(PN)

**(Se e solo se): (**w è in L(PF)?**)**

L'unico modo che ha PN di svuotare il suo stack è che PF entri in uno dei suoi stati accettanti, perché X0 non fa parte di PF.

Quindi w è in L(PF) perché q è uno stato accettante di PF, se si cavano i 2 stati aggiunti in PN e il simbolo X0 si hanno le stesse computazioni anche in PF, quindi si ottiene:



Quindi, w è in L(PF)

-----------------  
  
  
EQUIVALENZA DI PDA E CFG

Un linguaggio e’ generato da una CFG

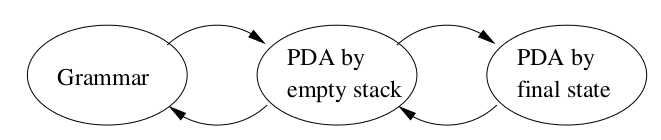
se e solo se

e’ accettato da un PDA per pila vuota

se e solo se

e’ accettato da un PDA per stato finale.

Sappiamo già andare da PDA per pila vuota a PDA per stato finale, ci manca il passaggio da CFG a PDA e da PDA a CFG.



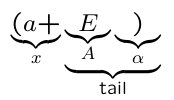
DA CFG A PDA PER PILA VUOTA:

Data G, costruiamo un PDA che simula ⇒LM.

Scriviamo le forme sentenziali come

xAα

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



Sia xAα ⇒LM xβα. Questo corrisponde al PDA che ha prima consumato x e ha Aα sulla pila, e poi, leggendo epsilon, elimina A e mette

β sulla pila.

Alla configurazione (q, y, βα) 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.

Praticamente, se in cima allo stack c'è una variabile, l'automa da solo sostituisce A con la sua produzione LEFT-MOST (β).

Se invece in cima allo stack c'è un terminale, il PDA controlla che lo stesso terminale è il prossimo carattere di input.

Se così non ', questo ramo non deterministico muore.

Se tutte le scommesse sono giuste, il PDA finisce l’input con la

pila vuota.

Formalmente, sia G = (V, T, Q, S) una CFG.

Definiamo PG come

({q}, T, V ∪ T, δ, q, S),

dove per A ∈ V: (per ogni variabile)

δ(q, , A) = {(q, β) : A → β ∈ Q},

e per a ∈ T:

δ(q, a, a) = {(q, epsilon)},

ESEMPIO:

Consideriamo la grammatica

S → |SS|iS|iSe.

Il PDA corrispondente e’

P = ({q}, {i, e}, {S, i, e}, δ, q, S),

dove

δ(q, epsilon , S) = {(q, epsilon), (q, SS), (q, iS), (q, iSe)},

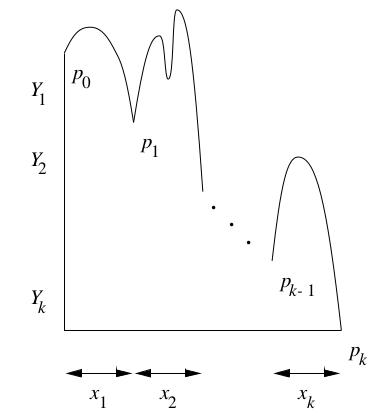
δ(q, i, i) = {(q, epsilon)},

δ(q, e, e) = {(q, epsilon)}.

DA PDA A CFG

Vediamo un PDA che consuma x = x1 x2 · · · xk e vuota la pila

Y = Y1 Y2 · · · Yk .



Definiremo una grammatica con variabili della forma [pi−1 Yi pi] che rappresentano il passaggio da pi−1 a pi con l’effetto di eliminare Yi.

(Quindi c'è solo una produzione per quella variabile, non si considerano tutti i passaggi intermedi della variabile di stack Yi.)

Formalmente, sia P = (Q, Σ, Γ, δ, q0 , Z0 ) un PDA. Definiamo G =

(V, Σ, R, S), dove

V = {[pXq] : {p, q} ⊆ Q, X ∈ Γ} ∪ {S}

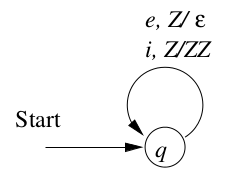
R = {S → [q 0 Z 0 p] : p ∈ Q} ∪

{[q Xr k] → a[r Y1 r1 ] · · · [rk−1 Yk rk] : a ∈ Σ ∪ { epsilon },

con {r1 , . . . , rk} ⊆ Q,

e (r , Y1 Y2 · · · Yk) ∈ δ(q , a, X)}

ESEMPIO:



PN = ({q}, {i, e}, {Z}, δ N , q, Z),

dove δN (q, i, Z) = {(q, ZZ)},

e δN (q, e, Z) = {(q, epsilon)} in una grammatica

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

dove V = {[qZq], S}, e

R = {[qZq] → i[qZq][qZq], [qZq] → e}.

Se rimpiazziamo [qZq] con A otteniamo le produzioni S → A e

A → iAA|e.

ESEMPIO:

Convertiamo P = ({p, q}, {0, 1}, {X, Z0}, δ, q, Z0),

dove δ e’ data da:

1. δ(q, 1, Z0) = {(q, X Z0 )}

2. δ(q, 1, X) = {(q, XX)}

3. δ(q, 0, X) = {(p, X)}

4. δ(q, epsilon , X) = {(q, epsilon)}

5. δ(p, 1, X) = {(p, epsilon)}

6. δ(p, 0, Z0) = {(q, Z0)}

in una CFG.

S è così per definizione:

[q0 Z0 p], per ogni stato p in Q

Dove q0 è lo stato iniziale, Z0 il simbolo iniziale del PDA.

Quindi c'è un a produzione per ogni stato, per la variabile iniziale S della CFG.

Da chiedere al prof a ricevimento, pag 59 parte2.pdf

pag 242 del libro inglese

PDA DETERMINISTICI (DPDA)

Un PDA P = (Q, Σ, Γ, δ, q0 , Z0 , F) e’ deterministico se e solo se

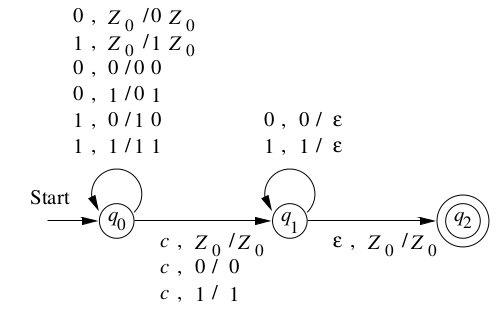
1. δ(q, a, X) e’ sempre o vuoto o con un solo elemento.

2. Se δ(q, a, X) non e’ vuoto, allora δ(q, epsilon, X) deve essere vuoto.

ESEMPIO:

L wcwr = {wcw R : w ∈ {0, 1}∗ }

Allora L wcwr e’ riconosciuto dal seguente DPDA:



Mostreremo che:

Regolari ⊂ L(DPDA) ⊂ CFL

**Teorema 6.17**:

Se L e’ regolare, allora L = L(P ) per qualche DPDA P .

**Prova**:

Dato che L e’ regolare, esiste un DFA A tale che L = L(A).

Sia

A = (Q, Σ, δ A , q 0 , F )

definiamo il DPDA

P = (Q, Σ, {Z0}, δP , q0 , Z0 , F),

dove

δP (q, a, Z0) = {(δA (q, a), Z0)},

per tutti i p, q ∈ Q e a ∈ Σ.

Un’induzione su |w| ci da:



Abbiamo visto che Regolari ⊆ L(DPDA).

L wcwr ∈ L(DPDA)\ Regolari

**Mostriamo ora L(DPDA) ⊂ CFL:**

I DPDA che accettano per pila vuota?, possono riconoscere solo CFL 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 .

Ci sono linguaggi in CFL\L(DPDA) ???

Si, per esempio L wwr .

**Cosa possiamo dire su DPDA e grammatiche ambigue?**

L wwr ha una grammatica non ambigua S → 0S0|1S1| ma non e’ L(DPDA).

Per l’inverso abbiamo:

**Teorema 6.20**:

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

(Questo è il linguaggio di un DPDA per pila vuota)

**Prova**:

Guardando la prova del teorema 6.14 vediamo che se la

costruzione e’ applicata ad un DPDA, il risultato e’ una CFG con

derivazioni a sinistra uniche per ogni stringa.

Il teorema 6.20 può essere rafforzato, dicendo che anche i linguaggi accettati dai DPDA per stato finale hanno una grammatica non ambigua.

**Teorema 6.21:**

Se L = 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 vedere che L’ ha la proprieta’ del prefisso.

Per il teorema 6.20 abbiamo che L’ = N (P’) per qualche DPDA P’.

Per il teorema 6.20 N(P’) puo’ essere generato da una CFG G non ambigua.

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

$ → epsilon

Dato che G ha derivazioni a sinistra uniche, anche G le ha, dato

che l’unica cosa nuova e’ l’aggiunta di derivazioni

w$ ⇒LM w

alla fine.

Da chiedere al prof a ricevimento

PROPRIETA’ DEI CFL

**Semplicazione**

di una CFG. Se un linguaggio e' un CFL, ha una grammatica in una possibile forma speciale.

**Pumping Lemma**

per CFL. Simile ai linguaggi regolari.

**Proprieta'** **di chiusura**.

Solo alcune delle proprieta' di chiusura dei linguaggi regolari valgono anche per i CFL.

**Proprieta' di decisione**.

Possiamo controllare l'appartenenza e l'essere vuoto, ma, per esempio, l'equivalenza di CFL e' non vericabile tramite un algoritmo (indecidibile).

FORMA NORMALE DI CHOMSKY:

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



dove A,B e C sono variabili, e a è un simbolo terminale. Questa e'

detta forma normale di Chomsky (CNF), e per ottenerla dobbiamo

innanzitutto **“pulire" la grammatica**:

1. Eliminare le produzioni epsilon, della forma



2. Eliminare le produzioni unita', cioe' produzioni della forma



dove A e B sono variabili.

3. Eliminare i simboli inutili, quelli che non appaiono in nessuna

derivazione



per simbolo iniziale S e terminale w.

1. SOMMARIO

Per pulire una grammatica bisogna fare, in quest’ordine:

* Eliminare le produzioni epsilon
* Eliminare le produzioni unità
* Eliminare i simboli inutili, eliminando prima i simboli non generanti e poi i simboli non raggiungibili

PUMPING LEMMA PER I CFL

**Pumping Lemma per linguaggi regolari**:

Se esiste una stringa abbastanza lunga per causare un ciclo in un DFA per il linguaggio, allora possiamo “ripetere" il ciclo e scoprire una infinità di stringhe che appartengono al linguaggio.

**Pumping Lemma per CFL** (un po' piu' complicato):

E’ sempre possibile trovare in una stringa sufficientemente lunga, due pezzi distinti da ripetere “in tandem":

cioe', se ripetiamo ognuno di essi lo stesso numero di

volte, otteniamo una nuova stringa appartenente al linguaggio.

**Enunciato del Pumping Lemma per i CFL:**

Per ogni CFL L, esiste un intero n, tale che per ogni stringa z € L tale che |z| >= n, esiste un troncamento di

z = uvwxy tale che:

1. |vwx| <= n
2. |vx| >0
3. Per ogni i >= 0: 

Per dimostrare il pumping lemma per i CFL, dobbiamo ricordare alcune regole degli alberi binari:

**Parse Tree:**

Se una grammatica è in CNF, allora i suoi parse tree (alberi sintattici) sono tutti alberi binari, e le foglie non hanno fratelli.

Se abbiamo un parse tree di una grammatica in CNF, e abbiamo il prodotto del parse tree = w = stringa di terminali, allora:

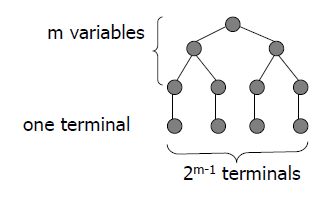
|w| <= 2^(n-1)

Dove n è il percorso più lungo del parse tree.

(il percorso di un albero è il numero di archi, cioè il numero di nodi – 1).

Ad es, se il percorso è uguale ad 1, c’è solo il nodo root con una foglia, quindi |w| <= 2^(0) = 1, cioè è un terminale e basta.

Se il percorso ha lunghezza n > 1, allora la produzione del simbolo iniziale deve contenere 2 variabili, sennò eravamo nel caso base di prima.



Un altro modo per dire la stessa cosa è:

m è il numero di variabili (che corrisponde al numero di archi dalla radice alla foglia, cioè il percorso più lungo di un parse tree)

Dato che la grammatica è in CNF, allora un nodo (variabile) ha 2 variabili (2 nodi) o un solo terminale (1 foglia, non può avere 2 foglie).

Quindi, se ci sono m variabili, il prodotto di un albero sintattico è una stringa w tale che |w| <= 2^(m-1).

**Prova:**

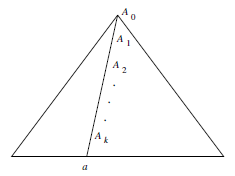
Sia G = (V,T,P,S) una grammatica in CNF, tale che |V| = m (ha m variabili).

Prendiamo n = 2^m e una stringa z € L tale che |z| >= n.

Dato che n = 2^m, allora ogni stringa w di questo parse tree ha |w| <= 2^(m-1), quindi per rappresentare una stringa w tale che |w| = 2^m ho bisogno di m+1 variabili.

Per la dimostrazione del pumping lemma dei CFL, abbiamo preso z tale che |z| >= n, quindi ogni parse tree di z contiene almeno m + 1 variabili.

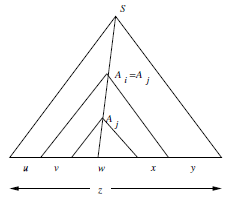
Nelle dispense intende l’altezza dell’albero con m+2



Un possibile parse tree per z quindi è formato da almeno m+1 variabili nel percorso dalla radice alle foglie, A0 A1 … Ak, dove k >= m.

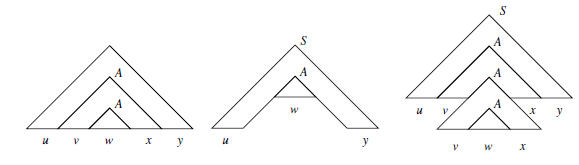
Dato che in V ci sono solo m variabili distinte, esiste una variabile ripetuta tale che Ai = Aj, con i != j e supponendo che i,j siano fra le ultime m+1 variabili del cammino.

E’ possibile dividere il parse tree di z, in modo tale che w è il prodotto del sottoalbero che ha Aj come root, e vwx è il prodotto del sottoalbero che ha come root Ai:



Dalle seguenti 3 osservazioni si dimostra l’enunciato del pumping lemma per i CFL:

1. Il sottoalbero che ha Ai come radice, ha al massimo m+1 variabili (quindi altezza m+2) perché abbiamo scelto la ripetizione delle 2 variabili Ai E Aj nelle ultime m+1 variabili del parse tree di z  
     
   Quindi la stringa corrispondente vwx ha lunghezza:  
     
   |vwx| <= 2^m, quindi |vwx| <= n
2. Le stringhe v e x non possono essere entrambe epsilon perché Ai genera 2 variabili entrambe non annullabili, quindi   
   |vx| > 0
3. Il parse tree che si ottiene sostituendo un numero arbitrario di volte il sottoalbero Ai nel sottoalbero Aj, continua ad essere un parse tree corretto, perché Ai e Aj sono la stessa variabile.  
     
   Quindi, per ogni i >= 0, 



(i = 0, i = 1, i = 2)

APPLICAZIONE DEL PUMPING LEMMA PER I CFL

FARE ANCHE ESEMPI DEL LIBRO + ES CON SOLUZIONI

ESEMPIO 1 DELLE SLIDE:

Si consideri L = 

**Prova:**

Sia n la costante del pumping lemma.

Si consideri la stringa 

Si ha che z € L e |z| > n (ha 3 tronconi di zero-n)

Per il pumping lemma, z = uvwxy, con |vwx| <= n e |vx| > 0

E  € L per ogni i >= 0.

Si considerano i 2 casi:

1. Se vx contiene almeno un 1, la stringa risultante dal pumping lemma per i > 1 ha più di un 1, quindi non appartiene ad L
2. Se vx contiene solo 0, dato che |vwx| <= n, non può contenere tutti gli zeri dei 3 gruppi di zeri.  
   Quindi per i 1, la stringa risultanti ha 3 gruppi di zeri con cardinalità diversa.

Esempio 2:

Si consideri L = 

Sia n la costante del pumping lemma.

Sia z = , si ha quindi che z € L e |z| <= n (in particolare è n^2).

Per il pumping lemma, z = uvwxy, con |vwx| <= n e |vx| > 0, e

 per ogni i >= 0.

Consideriamo il caso i = 2.

Allora n^2 <  < (n + 1) ^ 2

Non essendoci quadrati perfetti tra n^2 e (n+1)^2, allora per i = 2, , contraddicendo il pumping lemma.

PROPRIETA’ DI CHIUSURA DEI CFL

**Teorema 7.24**:

I CFL sono chiusi rispetto ai seguenti operatori

* unione,
* concatenazione
* chiusura di Kleene
* chiusura positiva +

A differenza dei linguaggi regolari, non sono chiusi né rispetto all’intersezione né rispetto alla differenza.

**Prova:**

Per Esercizio, si basa sulle sostituzioni, se ho tempo ci guardo meglio (libro pag 282 versione inglese)

PROPRIETÀ DI DECISIONE PER I CFL:

Riguarda i seguenti problemi:

• Complessita’ della conversione fra CFG e PDA

• Conversione di una CFG in CNF

• Verificare se L(G) = ∅, per una CFG G

• Verificare se w ∈ L(G), per una stringa w ed una CFG G

• Anticipazione di problemi indecidibili

**CONVERSIONE FRA CFG E PDA:**

Sia n la dimensione dell'input:

Le seguenti operazioni hanno complessità O(n):

1. Conversione della CFG in PDA

2. Conversione di un PDA che riconosce per stato finale in un

PDA che riconosce per stack vuoto

3. Conversione di un PDA che riconosce per stack vuoto in un

PDA che riconosce per stato finale

Mentre da PDA a CFG la complessità è O(n^3), perché:

al massimo ci sono n^3 variabili di tipo [pXq],

c'è un simbolo per ogni transazione, quindi ci sono O(n) transizioni.

Ogni transizione produce n^2 produzioni, quindi O(n^2)

Quindi la complessità della conversione da PDA a CFG è =(n^3).

**CONVERSIONE DI UNA CFG IN CNF:**

1. Calcolare r(G) e g(G) ed eliminare i simboli inutili richiede

tempo O(n)

2. La dimensione di u(G) e la grammatica risultante dopo l’eliminazione delle produzioni unita’ e’ O(n 2 )

3. Modificare i corpi perche’ contengano solo variabili richiede

O(n)

4. La riduzione dei corpi perche’ abbiano lunghezza 2 richiede

O(n)

L’eliminazione dei simboli annullabili puo’ rendere la grammatica

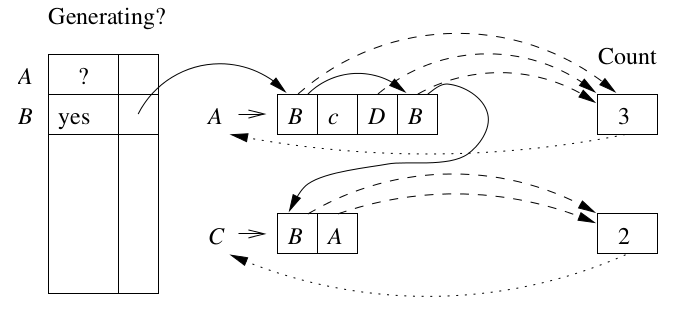
di dimensione O(2^n)

Ma se si riduce la dimensione dei corpi delle produzioni prima di eliminare i simboli annullabili, la complessità totale della conversione è O(n^2).

**VERIFICARE SE CFL È VUOTO:**

L(G) è non vuoto se il simbolo iniziale è generante.

Per calcolare l'insieme dei simboli generanti la complessità è O(n):



Per ogni variabile c'è una lista che ricorda tutte le posizioni in cui compare.

Per ogni produzione, il count indica il numero di variabili che devono essere controllate.

Ogni count è legto alla variabile di testa della produzione.

Ogni variabile che si trova generante, diminuisce il count di 1.

Quando il count raggiunge 0, la variabile di testa della produzione diventa generante, e si fa lo stesso procedimento per quella variabile.

Se alla fine il simbolo iniziale è marcato yes, allora è generante, quindi il linguaggio non è vuoto.

La complessità è O(n) perché:

- La creazione e l'inizializzazione dell'array è O(n)

- Quando un contatore raggiunge lo 0, bisogna marcare la variabile di testa come “yes”, richiede O(1)

e

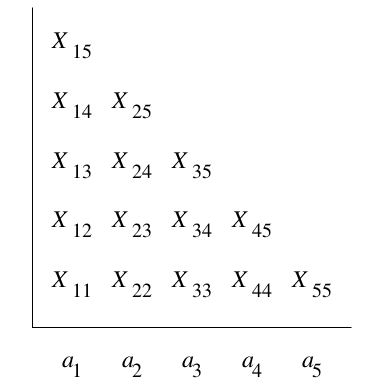
bisogna seguire i link per la variabile di testa e decrementare i contatori.

Con complessità O(n)

La complessità totale quindi è O(n).

**VERIFICARE SE W € L(G):**

Si usa l'algoritmo CYK, che permette di vedere se una stringa appartiene ad una grammatica in CNF in tempo O(n^3)



L'algoritmo si basa su una tabella triangolare, costruita dal basso verso l'alto.

Nell'asse delle x ci vanno scritti tutti i terminali della stringa w da controllare, in questo caso w = a1 a2 a3 a4 a5

Ogni variabile Xij indica l'insieme delle variabili che, in zero o più passi, producono la stringa ai...aj.

La prima riga è il caso base: dato che i = j, le variabili devono produrre direttamente il terminale

**Base:**



**Induzione:**

Si calcola Xij, con i < j, altrimenti siamo nel caso base.

Xij contiene l'insieme delle variabili che in zero o più passi producono la stringa ai...aj, quindi dato che la grammatica G è in CNF, la variabile deve contenere una produzione del tipo:



dove B produce in zero o più passi ai...ak e C produce in zero o più passi ak+1...aj.

Quindi una variabile A appartiene all'insieme Xij se:

1) i <= k < j

2) B è in Xik

3) C è in Xk+1,j

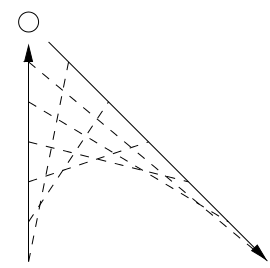


4) è una produzione di G

Il costo computazionale è =(n^3), perché la tabella triangolare è formata da O(n^2) elementi, e per ogni elemento Xij bisogna calcolare n coppie:



Per ogni coppia, si sale nella colonna e si scende nella diagonale della variabile Xij considerata:



**PROBLEMI INDECIDIBILI PER LE CFL**

1. Una data CFG G e’ ambigua?

2. Un dato CFL L e’ inerentemente ambigua?

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

4. Due CFL sono uguali?

5. Un CFL e’ universale (cioe’ uguale a Σ∗)?