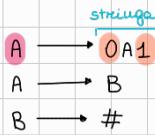


## 2.1 GRAMMATICHE CONTEXT-FREE

Un esempio di grammatica context-free, che chiamiamo  $G_1$ , è la seguente:



Una grammatica consiste di un insieme di **regole di sostituzione**, anche chiamate **produzioni**. Ogni regola appare come una linea della grammatica, costituita da un simbolo e una stringa separati da una freccia. Il simbolo è chiamato una **variabile** - la stringa consiste di variabili e altri simboli chiamati **terminali**. Le variabili sono spesso rappresentate da lettere maiuscole. I terminali sono analoghi ai simboli dell'alfabeto di input e sono spesso rappresentati da lettere minuscole, numeri o simboli speciali.

Una sequenza di sostituzioni per ottenere una stringa è chiamata una **derivazione**. Per esempio, una derivazione della stringa  $000\#\underline{111}$  nella grammatica  $G_1$  è:

$$A \Rightarrow OA1 \Rightarrow 0OA11 \Rightarrow 00OA111 \Rightarrow 000B111 \Rightarrow 000\#\underline{111}$$

Tutte le stringhe generate in questo modo costituiscono il **linguaggio della grammatica**. Ogni linguaggio che può essere generato da una grammatica context-free è chiamato un **linguaggio context-free (CFL)**.

## DEFINIZIONE FORMALE DI GRAMMATICA CONTEXT-FREE

### Definizione 2.2

Una grammatica context-free è una quadrupla  $(V, \Sigma, R, S)$ , dove:

1.  $V$  è un insieme finito i cui elementi sono chiamati variabili;
2.  $\Sigma$  è un insieme finito, disgiunto da  $V$ , i cui elementi sono chiamati terminali;
3.  $R$  è un insieme finito di regole;
4.  $S \in V$  è la variabile iniziale;

## 2.2 AUTOMI A PILA

In questa sezione introduciamo un nuovo tipo di modello computazionale chiamato **automa a pila** (pushdown automata). Questi automi sono come gli automi finiti non deterministici ma hanno una componente in più chiamata **pila** (stack). La pila consente a tali automi di riconoscere alcuni linguaggi non regolari.

Un automa a pila (PDA) può scrivere simboli nella pila e rileggerli in seguito. Scrivere un simbolo "spinge giù" tutti gli altri simboli nella pila. In un qualunque momento il simbolo sulla cima (top) della pila può essere letto e rimosso. I rimanenti simboli allora ritornano più in alto. L'operazione di scrivere un simbolo sulla pila è chiamata **push**, quella di eliminare un simbolo dalla pila è spesso chiamata **pop**. Una pila è un dispositivo "last in, first out".

## DEFINIZIONE FORMALE DI AUTOMA A PILA

La pila è un dispositivo che contiene simboli presi da un qualche alfabeto. La macchina può usare differenti alfabeti per il suo input e la sua pila, quindi ora specifichiamo sia un alfabeto di simboli di input  $\Sigma$  sia un alfabeto di pila  $\Gamma$ .

Il dominio della funzione di transizione è  $Q \times \Sigma \times \Gamma^*$ . Quindi lo stato corrente, il prossimo simbolo di input letto e il simbolo sulla cima della pila determinano la mossa seguente di un automa a pila. Per quanto riguarda il codominio della funzione di transizione, dobbiamo considerare cosa fa l'automa quando è in una specifica situazione. Esso può trovarsi in un nuovo stato ed eventualmente scrivere un simbolo sulla cima della pila. La funzione  $S$  può indicare quest'azione restituendo un elemento di  $Q$  e un elemento di  $\Gamma^*$ , quindi un elemento di  $Q \times \Gamma^*$ . Poiché permettiamo il non determinismo in questo modello, una situazione può avere diverse mosse successive legite. Dunque la funzione  $S$  restituisce  $P(Q \times \Gamma^*)$ .

### Definizione 2.13

Un automa a pila è una struttura  $(Q, \Sigma, \Gamma, q_0, F)$ , dove

$Q, \Sigma, \Gamma$  ed  $F$  sono tutti insiemi finiti e

1.  $Q$  è l'insieme degli stati;
2.  $\Sigma$  è l'alfabeto di input;
3.  $\Gamma$  è l'alfabeto della pila;
4.  $\delta: Q \times \Sigma \times \Gamma \rightarrow P(Q \times \Gamma)$  è la funzione di transizione;
5.  $q_0 \in Q$  è lo stato iniziale;
6.  $F \subseteq Q$  è l'insieme degli stati accettanti.

## DEFINIZIONE FORMALE DI COMPUTAZIONE DI UN PDA

Siamo dati:

- Un automa a pila  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ ;
- Una stringa di ingresso  $w$  tale che  $w = w_1 w_2 \dots w_m$ , dove ogni  $w_i$  fa parte dell'alfabeto  $\Sigma_\Sigma$ ;
- Una stringa  $s$  tale che  $s = s_1 s_2 \dots s_m$ , dove ogni  $s_i$  appartiene allo stack;

Diciamo allora che  $M$  accetta  $w$  se esiste una sequenza di stati  $r_0, r_1, \dots, r_m \in Q$  che rispettino tre condizioni:

1.  $r_0 = q_0$  e  $s_0 = \epsilon$  (la macchina parte dallo stato iniziale e con lo stack vuoto);
2. Per ogni  $i = 0, \dots, m-1$ , abbiamo che  $(r_i, a, b) \in \delta(r_i, w_{i+1}, s_i)$ , dove  $a \in \Sigma$  e  $s_{i+1} = b t$  per qualche  $t \in \Gamma^*$  (lo stato futuro di  $M$  dipende dallo stato attuale, dal simbolo in ingresso e dallo stack);
3.  $r_m \in F$  (alla fine dell'input  $M$  si trova in uno stato accettante).

## EQUIVALENZA CON LE GRAMMATICHE CONTEXT-FREE

In questa sezione mostriamo che grammatiche context-free e automi a pila sono computazionalmente equivalenti. Mostreremo come trasformare ogni grammatica context-free in un automa a pila che riconosce lo stesso linguaggio e viceversa.

### Teorema 2.20

Un linguaggio è context-free se e solo se esiste un automa a pila che lo riconosce.

Come al solito, per i teoremi "se e solo se", abbiamo due enunciati da provare, nelle due direzioni.

### Lemma 2.21

Se un linguaggio è context-free, allora esiste un automa a pila che lo riconosce.

**IDEA:** Dato che per definizione un CFL è generato da una grammatica context-free, non dobbiamo fare altro che dimostrare che ogni CFG  $G$  può essere convertita in un equivalente PDA  $P$ . In pratica dobbiamo costruire  $P$  in modo che se gli passiamo in ingresso una stringa  $w$  generata da  $G$ , questo dovrà accettarla. Ma come fa a capire se è davvero generata dalla grammatica a lui equivalente?

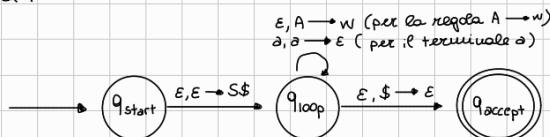
Nel dettaglio,  $P$  comincia la sua computazione scrivendo la variabile iniziale sullo stack; poi prosegue attraverso una serie di stringhe intermedie ottenute sostituendo le variabili secondo le regole della grammatica; infine quando arriva a una stringa composta da soli terminali la confronta con  $w$ : se corrispondono allora  $P$  accetta, altrimenti abbandona quel ramo della computazione.

Ma dove le memorizzo le stringhe intermedie? Non posso metterle così come sono sullo stack. Se infatti dovesse fare una sostituzione di una variabile in una stringa intermedia e questa non si trova in cima allo stack, sarebbe un problema. La soluzione è conservare solo una parte della stringa intermedia nello stack, quella che va dalla prima variabile in poi; tutti i simboli che vengono prima devono corrispondere con la stringa in ingresso, altrimenti buttiamo via tutto.

**DIH:** Sia dato un automa a pila  $P = (Q, \Sigma, \Gamma, \delta, q_1, F)$ . I tre stati fondamentali di  $P$  sono  $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\}$ . Diamo ora una descrizione del funzionamento di  $P$ :

1.  $\delta(q_{\text{start}}, \epsilon, \epsilon) = \{q_{\text{loop}}, \$\}$  (metti il simbolo  $\$$  e la variabile iniziale nello stack, quindi vai in  $q_{\text{loop}}$ )
2. Continua a ripetere i seguenti passi:
  - 1)  $\delta(q_{\text{loop}}, \epsilon, A) = \{(q_{\text{loop}}, w) / A \rightarrow w\}$  (Se in cima allo stack c'è una variabile  $A$ , seleziona in modo non deterministico una delle regole per  $A$  e sostituisci  $A$  con la stringa che si trova a destra della regola).
  - 2)  $\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \epsilon)\}$  (Se in cima allo stack c'è un simbolo terminale  $a$ , leggi il prossimo simbolo dall'ingresso e confrontalo con  $a$ . Se corrispondono, ripeti; altrimenti abbandona questo ramo della computazione non deterministica)
  - 3)  $\delta(q_{\text{loop}}, \epsilon, \$) = \{(q_{\text{accept}}, \epsilon)\}$  (Se in cima allo stack c'è il simbolo  $\$$ , entra nello stato di accettazione).

Ecco il diagramma degli stati di  $P$ :



### Lemma 2.27

Se un linguaggio è riconosciuto da un automa a pila, allora esso è context-free.

**IDEA:** Situazione inversa: abbiamo un PDA  $P$  e vogliamo creare un CFG  $G$  che genera tutte le stringhe accettate da  $P$ .

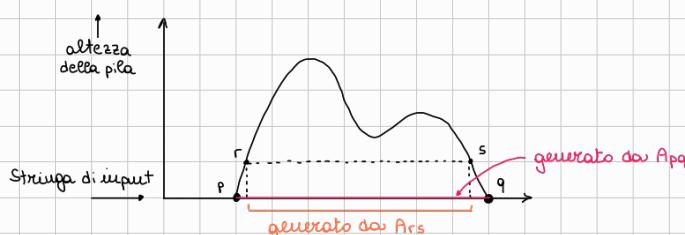
Come prima cosa, dovremo aggiungere alla nostra grammatica  $G$  la variabile  $A_{pq}$  per ogni coppia di stati  $p$  e  $q$  in  $P$ . Non è una variabile a caso, ma quella che genera tutte le stringhe che possono portare il PDA da  $p$  a  $q$  lasciando lo stack nelle stesse condizioni in cui lo si è trovato. Per riuscirci dobbiamo fornire a  $P$  tre nuove caratteristiche:

1. Deve avere un unico stato accettante  $q_{\text{accept}}$ ;
2. deve svuotare il suo stack prima di accettare;
3. Ogni transizione deve consistere o in un'operazione di push o in una di pop, non entrambe.

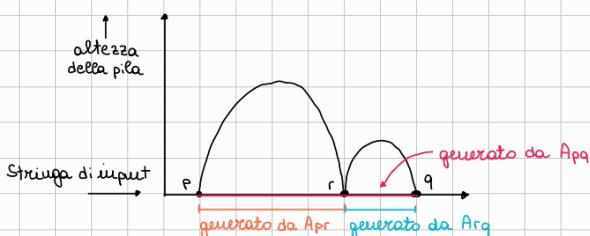
Qualcosa sul comportamento di  $P$  già lo si capisce da queste caratteristiche: la prima operazione su una qualsiasi stringa  $x$  sarà di push, l'altra di pop. Ma le altre? Distinguiamo due casi:

- Il simbolo estratto dallo stack alla fine della computazione è quello che avevo messo all'inizio, quindi lo stack è vuoto solo in questi due momenti. Simuliamo questa eventualità con la regola:  $A_{pq} \rightarrow a A_{rs} b$ , dove  $a$  è il simbolo letto in ingresso al primo passo,  $b$  è il simbolo letto in ingresso all'ultimo passo,  $r$  è lo stato che segue  $p$  ed  $s$  è quello che precede  $q$ .

Aiutiamoci con uno schema:



- Seconda situazione: il simbolo estratto dallo stack alla fine della computazione non è quello che avevo messo all'inizio, quindi lo stack si è svuotato completamente tra  $p$  e  $q$  almeno una volta, in uno stato che chiameremo  $r$ . Per questa eventualità introduciamo la regola:  $A_{pq} \rightarrow A_{pr} A_{rq}$ . Aiutiamoci anche in questo caso con uno schema:



la costruzione di  $G$  termina aggiungendo per ogni  $p \in Q$  la regola  $\text{App} \rightarrow E$ .

La costruzione è questa ma per dimostrare la seconda implicazione del teorema dovremo dimostrare che  $\text{App}$  genera una stringa  $x$  se e solo se  $\exists x$  può portare  $P$  da  $p$  con stack vuoto a  $q$  con stack vuoto. Distinguiamo ancora i due versi dell'implicazione:

### Fatto 2.30

Se  $\text{App}$  genera  $x$ , allora  $x$  può portare  $P$  da  $p$  con la pila vuota a  $q$  con la pila vuota.

DIH: la dimostrazione viene fatta per induzione.

- Passo base: la derivazione ha un solo passo, quindi nella parte destra della regola ci sono solo terminali. L'unica regola di questo tipo in  $G$  è  $\text{App} \rightarrow E$ , che è verificata perché se non prendo niente in ingresso parto con lo stack vuoto e finisco con lo stack vuoto.

- Passo induttivo: Consideriamo vere le derivazioni con al più  $K$  passi, con  $K \geq 1$ , e verifichiamo se sono vere anche per  $K+1$  passi.

Supponiamo di avere  $\text{App} \xrightarrow{*} x$  ( $\text{App}$  deriva  $x$ ) in  $K+1$  passi. Come abbiamo visto, nel primo passo di derivazione sono due le regole che possono essere applicate:

- 1)  $\text{App} \rightarrow a \text{ Ars } b$ : consideriamo una porzione  $y$  della  $x$  generata da  $\text{Ars}$ , tale che  $x = ayb$ . Dato che  $\text{Ars} \xrightarrow{*} y$  in  $K$  passi, l'ipotesi induttiva ci dice che  $P$  può andare da  $r$  con stack vuoto ad  $s$  con stack vuoto. Vediamo bene cosa succede:  $P$  parte da  $p$  con lo stack vuoto, e dopo aver letto  $a$  va nello stato  $r$  mettendo  $t$  in cima allo stack; poi  $P$  legge  $b$  che lo porta allo stato  $q$ , facendogli fare un pop di  $t$  dallo stack. Perfetto: parto e arrivo con lo stack vuoto;
2.  $\text{App} \rightarrow \text{Apr } \text{Arg}$ : consideriamo le porzioni  $y$  e  $z$  delle  $x$  che generano rispettivamente  $\text{Apr}$  e  $\text{Arg}$ , tali che  $x = yz$ . Dato che  $\text{Apr} \xrightarrow{*} y$  e  $\text{Arg} \xrightarrow{*} z$  in massimo  $K$  passi, l'ipotesi induttiva ci dice che  $y$  può portare  $P$  da  $p$  a  $r$  con lo stack vuoto all'inizio e alla fine; stessa cosa per  $z$  che porta  $P$  da  $r$  a  $q$ . Quindi, sommando gli effetti,  $x$  può portare  $P$  dallo stato  $p$  con stack vuoto allo stato  $q$  con stack vuoto. Perfetto, abbiamo completato il passo induttivo.

### Fatto 2.31

Se  $x$  può portare  $P$  da  $p$  con la pila vuota a  $q$  con la pila vuota,  $\text{App}$  genera  $x$ .

DIH: Anche in questo caso la dimostrazione va fatta per induzione.

- Passo base: la computazione ha 0 passi, quindi stato iniziale e finale coincidono. Chiamiamo  $p$  questo stato, per cui vale la regola  $\text{App} \xrightarrow{*} x$ . Ma dato che in 0 passi  $P$  può leggere solo la stringa vuota, allora  $x = \epsilon$ ; sostituendo il valore di  $x$  nella regola di prima e otteniamo  $\text{App} \rightarrow E$ , che è una regola di  $G$  per costruzione. La base è provata.

- Passo induttivo: consideriamo vere le computazioni con al più  $K$  passi, con  $K \geq 0$ , e verifichiamo se sono vere per  $K+1$  passi. Se supponiamo che  $P$  abbia una computazione in cui  $x$  porta  $p$  a  $q$  con stack vuoto in  $K+1$  passi, i casi sono due:

- 1) "Lo stack è vuoto solo all'inizio e alla fine della computazione": chiamiamo  $t$  il simbolo che metto in cima allo stack alla prima mossa e che tolgo solo all'ultima. Sia  $a$  l'ingresso letto al primo passo,  $b$  quello letto all'ultimo,  $r$  lo stato in cui vado dopo la prima mossa, ed  $s$  lo stato che precede l'ultima. Abbiamo allora che  $(r, t) \in \delta(p, a, E)$  e  $(q, \epsilon) \in \delta(s, b, t)$ , perciò la regola  $\text{App} \rightarrow a \text{ Ars } b$  è in  $G$ .

Se adesso chiamiamo  $y$  la parte di  $x$  priva dei simboli  $a$  e  $b$  (dove  $x = ayz$ ), l'ingresso  $y$  può portare  $P$  dallo stato  $r$  allo stato  $s$  senza toccare il simbolo  $t$  nello stack. Evitando di considerare  $a$  e  $b$  abbiamo rimosso due passaggi dai  $K+1$  necessari per la computazione di  $x$ , quindi  $y$  richiede  $K-1$  passi. Grazie all'ipotesi induttiva possiamo dunque affermare che  $\text{Ars} \xrightarrow{*} y$ , e di conseguenza  $\text{App} \xrightarrow{*} x$ .

- 2) "Lo stack è vuoto all'inizio e alla fine della computazione, e anche in un altro punto": sia  $r$  lo stato diverso da quello iniziale e finale, in cui lo stack è vuoto. Quindi le parti che vanno da  $p$  a  $r$  a  $q$  impiegano al massimo  $K$  passi. Chiamiamo ora  $y$  l'ingresso letto da  $P$  nella prima parte e  $z$  quello letto nella seconda. L'ipotesi induttiva ci dice che  $\text{App} \xrightarrow{*} y$  e  $\text{Arg} \xrightarrow{*} z$ . Ma dato che  $\text{App} \rightarrow \text{Apr } \text{Arg}$  è in  $G$ ,  $\text{App} \xrightarrow{*} x$ , e quindi abbiamo completato il passo induttivo.

### Cordillario 2.32

Ogni linguaggio regolare è context-free.

### 2.3 LINGUAGGI NON CONTEXT-FREE

In questa sezione presenteremo una tecnica per dimostrare che alcuni linguaggi non sono context-free.

#### IL PUMPING LEMMA PER I LINGUAGGI CONTEXT-FREE

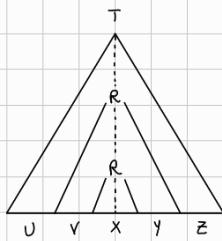
##### Teorema 2.24

Se  $A$  è un linguaggio context-free, allora esiste un numero  $p$  (la lunghezza del pumping) tale che, se  $s$  è una qualsiasi stringa in  $A$  di lunghezza almeno  $p$ , allora si può essere divisa in cinque parti:  $s = uvxyz$  che soddisfano le condizioni:

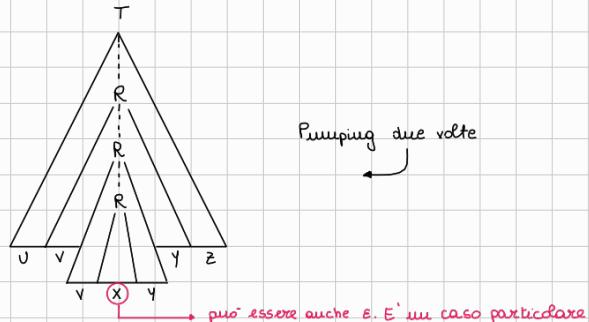
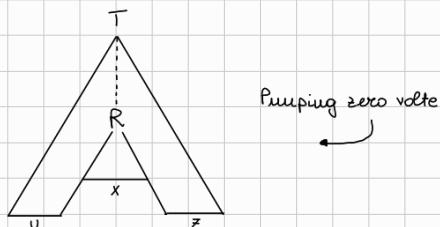
1. per ogni  $i \geq 0$ ,  $uv^ixy^z \in A$
2.  $|vy| > 0$
3.  $|vxy| \leq p$

Quando  $s$  è divisa in  $uvxyz$ , la condizione 2 afferma che  $v$  o  $y$  non è la stringa vuota, altrimenti il teorema sarebbe banalmente vero.

**IDEA:** Sia  $A$  un CFL e sia  $G$  un CFG che lo genera. Sia  $s$  una stringa "molto lunga" in  $A$ . Dato che  $s \in A$ , essa è derivabile da  $G$  e quindi ha un albero sintattico. Poiché  $s$  è molto lunga, l'albero sintattico sarà "molto alto".



Per il principio della piccionaia, qualche simbolo di variabile  $R$  si deve ripetere lungo il cammino. Questa ripetizione ci permette di sostituire il sottoalbero sotto la seconda concorrenza di  $R$  con il sottoalbero sotto la prima concorrenza di  $R$  e ottenere ancora un albero sintattico consentito. Pertanto possiamo dividere  $s$  in cinque parti:  $uvxyz$  e possiamo replicare il secondo e quarto pezzo e ottenere una stringa ancora nel linguaggio.



**DIH:** Sia  $G$  un CFG per il CFL  $A$ . Sia  $b$  il massimo numero di simboli nel lato destro di una regola (assumiamo che sia almeno 2). Sappiamo che, in ogni albero sintattico costruito usando questa grammatica, un nodo non può avere più di  $b$  figli. In altre parole, ci sono al più  $b$  foglie in un passo dalla variabile iniziale; ci sono al più  $b^2$  foglie in 2 passi dalla variabile iniziale; e ci sono al più  $b^n$  foglie in  $n$  passi dalla variabile iniziale. Viceversa, se una stringa generata è almeno  $b^{n+1}$ , qualunque dei suoi alberi sintattici deve essere almeno  $n+1$  alto.

## FORMA NORMALE DI CHOMSKY

La forma normale di Chomsky ci permette di riscrivere una CFG in forma più compatta e semplificata.

### Definizione 2.8

Una grammatica context-free è in forma normale di Chomsky se ogni regola è della forma:

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow \alpha \end{aligned}$$

dove  $\alpha$  è un terminale e  $A, B, C$  sono variabili qualsiasi - tranne che  $B$  e  $C$  non possono essere la variabile iniziale. Inoltre, permettiamo la regola  $S \rightarrow E$ , dove  $S$  è la variabile iniziale.

### Teorema 2.9

Ogni linguaggio context-free è generato da una grammatica context-free in forma normale di Chomsky.

DIM: Dato che per definizione un CFL è generato da una CFG, non dobbiamo fare altro che dimostrare che ogni CFG  $G$  può essere espressa nella forma normale di Chomsky. Lo facciamo per costruzione, scandendo i vari passi.

#### Passo 1:

Definiamo una nuova variabile iniziale  $S_0$  e aggiungiamo la regola:  $S_0 \rightarrow S$ , dove  $S$  era la variabile iniziale di  $G$ .

#### Passo 2:

Dobbiamo far sparire tutte le regole espresse nella forma  $A \rightarrow E$ , dove  $A$  è una variabile diversa da quella iniziale. Si fa in due tempi:

- 1) Eliminiamo tutte le regole nella forma  $A \rightarrow E$ ;
- 2) Per ogni occorrenza di  $A$  nella parte destra della regola, aggiungiamo una nuova regola in cui  $A$  non vi compare. In pratica, se abbiamo una regola del tipo  $R \rightarrow uAv$  (dove  $u$  e  $v$  sono stringhe di variabili e/o terminali), la facciamo diventare  $R \rightarrow uAv/uv$ .

#### Passo 3:

Dobbiamo far sparire tutte le regole unitarie espresse nella forma  $A \rightarrow B$ . Anche in questo caso si opera in due tempi:

- 1) Eliminiamo tutte le regole unitarie nella forma  $A \rightarrow B$ ;
- 2) Ogni volta che appare la regola  $B \rightarrow v$ , aggiungiamo la regola  $A \rightarrow v$ .

#### Passo 4:

Concludiamo la costruzione convertendo tutte le altre regole nella forma appropriata. In particolare dovremo individuare ogni regola del tipo:

$A \rightarrow U_1 U_2 \dots U_K$ , con  $K \geq 3$  e ogni  $U_i$  variabile o terminale e sostituirla con le regole:

$$A \rightarrow U_1 A_1$$

$$A_1 \rightarrow U_2 A_2$$

$$A_2 \rightarrow U_3 A_3$$

...

$$A_{K-2} \rightarrow U_{K-1} U_K, \text{ dove le varie } A_i \text{ sono nuove variabili.}$$

Quando arriveremo ad avere tutte le regole con  $K=2$ , sostituiamo il terminale  $U_i$  nelle regole precedenti con la nuova variabile  $U_i$ , e aggiungiamo la regola:  $U_i \rightarrow U_i$ .

### Esempio di conversione

La grammatica  $G$  da convertire è la seguente:

$$\begin{aligned} S &\rightarrow ASA/\alpha B \\ A &\rightarrow B/S \\ B &\rightarrow b/\epsilon \end{aligned}$$

Applichiamo il passo 1. Aggiungiamo una nuova variabile iniziale:

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA/\alpha B \\ A &\rightarrow B/S \\ B &\rightarrow b/\epsilon \end{aligned}$$

Applichiamo il passo 2. Eliminiamo la regola  $B \rightarrow \epsilon$ :

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA/\alpha B/\alpha \\ A &\rightarrow B/S/\epsilon \\ B &\rightarrow b \end{aligned}$$

Eliminiamo la regola  $A \rightarrow \epsilon$ :

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA/\alpha B/\alpha/SA/AS/S \\ A &\rightarrow B/S \\ B &\rightarrow b \end{aligned}$$

Applichiamo il passo 3. Eliminiamo la regola unitaria  $S \rightarrow S$ :

$$\begin{aligned} S_0 &\rightarrow S/ASA/\alpha B/\alpha/SA/AS \\ S &\rightarrow ASA/\alpha B/\alpha/SA/AS \\ A &\rightarrow B/S \\ B &\rightarrow b \end{aligned}$$

Eliminiamo la regola unitaria  $S_0 \rightarrow S$ :

$$\begin{aligned} S_0 &\rightarrow ASA/\alpha B/\alpha/SA/AS \\ S &\rightarrow ASA/\alpha B/\alpha/SA/AS \\ A &\rightarrow B/S \\ B &\rightarrow b \end{aligned}$$

Eliminiamo la regola unitaria  $A \rightarrow B$ :

$$\begin{aligned} S_0 &\rightarrow ASA/\alpha B/\alpha/SA/AS \\ S &\rightarrow ASA/\alpha B/\alpha/SA/AS \\ A &\rightarrow S/b \\ B &\rightarrow b \end{aligned}$$

Eliminiamo la regola unitaria  $A \rightarrow S$ :

$$\begin{aligned} S_0 &\rightarrow ASA/\alpha B/\alpha/SA/AS \\ S &\rightarrow ASA/\alpha B/\alpha/SA/AS \\ A &\rightarrow b/ASA/\alpha B/\alpha/SA/AS \\ B &\rightarrow b \end{aligned}$$

Applichiamo il passo 4. Convertiamo le regole rimanenti nella forma appropriata:

$$\begin{aligned} S_0 &\rightarrow AA_1/UB/\alpha/SA/AS \\ S &\rightarrow AA_1/UB/\alpha/SA/AS \\ A &\rightarrow b/AA_1/UB/\alpha/SA/AS \\ A_1 &\rightarrow SA \\ U &\rightarrow \alpha \\ B &\rightarrow b \end{aligned}$$