

Teoria dei Linguaggi

Indice

1. Lezione 12 [04/04]	4
1.1. Varianti di automi	4
1.1.1. Automi pesati	4
1.1.2. Automi probabilistici	4
2. Lezione 13 [09/04]	5
2.1. Varianti di automi	5
2.1.1. One-way VS two-way	5
2.1.2. Read-only VS read-write	5
2.1.3. Memoria esterna	6
2.2. Automi two-way	7
2.2.1. Esempi vari	7
2.2.2. Definizione formale	9
2.2.3. Potenza computazionale	10
3. Lezione 14 [11/04]	15
3.1. Simulazione	15
3.1.1. Problema di Sakoda & Sipser	15
3.2. Automi a pila	17
3.2.1. Versione non deterministica	17
3.2.2. Accettazione	20
3.2.3. Determinismo VS non determinismo	22
3.2.4. Trasformazioni	23
4. Lezione 15 [23/04]	25
4.1. Esempi	25
4.2. Equivalenza tra grammatiche di tipo 2 e automi a pila	25
4.2.1. Ripasso e introduzione	25
4.2.2. Da grammatica di tipo 2 ad automa a pila	28
4.2.3. Automa a pila a grammatica di tipo 2	31
4.3. Forme normali per le grammatiche context-free	31
4.3.1. FN di Greibach	31
5. Lezione 16 [30/04]	34
5.1. Fine dimostrazione	34
5.1.1. Forma normale per gli automi a pila	34
5.1.2. Dimostrazione	36
5.2. Forma normale di Chomsky	39
5.2.1. Definizione	39
5.2.2. Costruzione	39
5.2.2.1. Eliminazione delle ϵ -produzioni	40
5.2.2.2. Eliminazione delle produzioni unitarie	41
5.2.2.3. Eliminazione dei simboli inutili	42
5.2.2.4. Eliminazione dei terminali	42
5.2.2.5. Smontaggio delle produzioni	43
5.2.3. Esempio	43
6. Lezione 17 [07/05]	45
6.1. Prerequisiti per il pumping lemma	45
6.2. Pumping lemma per i CFL	48

6.3. Applicazioni del pumping lemma	52
7. Lezione 18 [09/05]	56
7.1. Ancora pumping lemma	56
7.2. Fail del pumping lemma	56
7.3. Lemma di Ogden	58
7.4. Applicazioni del lemma di Ogden	61
7.5. Ambiguità	62
8. Lezione 19 [14/05]	65
8.1. Ambiguità	65
8.2. Ambiguità e non determinismo	68
9. Lezione 20 [16/05]	72
9.1. Operazioni insiemistiche	72
9.1.1. Unione	72
9.1.1.1. CFL	72
9.1.1.2. DCFL	72
9.1.2. Intersezione	73
9.1.2.1. CFL	73
9.1.2.2. DCFL	73
9.1.3. Intersezione con un regolare	73
9.1.3.1. CFL	73
9.1.3.2. DCFL	73
9.1.4. Complemento	73
9.1.4.1. CFL	73
9.1.4.2. DCFL	74
9.1.5. Riassunto	77
9.2. Operazioni regolari	77
9.2.1. Prodotto	77
9.2.1.1. CFL	77
9.2.1.2. DCFL	77
9.2.2. Star	78
9.2.2.1. CFL	78
9.2.2.2. DCFL	78
9.2.3. Riassunto	78
10. Lezione 21 [21/05]	80
10.1. Riassunto chiusura operazioni	80
10.2. CFL vs DCFL	80
10.3. Ricorsione	84
10.4. Linguaggio di Dyck	85

1. Lezione 12 [04/04]

1.1. Varianti di automi

Per finire questa lezione infinita, vediamo qualche **variante** di automi.

1.1.1. Automi pesati

La prima variante che vediamo sono gli **automi pesati**. Essi associano ad ogni transizione un peso. Il **peso di una stringa** viene calcolato come la somma dei pesi delle transizioni che la stringa attraversa per essere accettata. Questo peso poi può essere usato in problemi di ottimizzazione, come trovare il cammino di peso minimo, ma questo ha senso solo su NFA.

1.1.2. Automi probabilistici

Un tipo particolare di automi pesati sono gli **automi probabilistici**, che come pesi sulle transizioni hanno la probabilità di effettuare quella transizione. Visto che parliamo di **probabilità**, i pesi sono nel range $[0, 1]$ e, dato uno stato, tutte le transizioni uscenti sommano a 1. In realtà, potremmo sommare a meno di 1 se nascondiamo lo stato trappola. Con questi automi possiamo chiederci con che probabilità accettiamo una stringa.

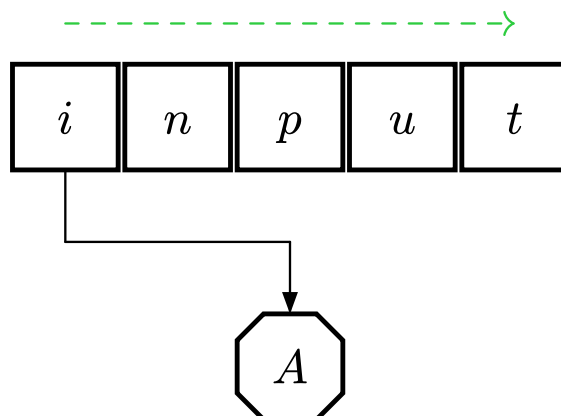
Questi automi li possiamo usare come **riconoscitori a soglia**: tutte le parole oltre una certa soglia le accettiamo, altrimenti le rifiutiamo.

Questi automi comunque non sono più potenti dei DFA: si può dimostrare che se la soglia λ è **isolata**, ovvero nel suo intorno non cade nessuna parola, allora possiamo trasformare questi automi probabilistici in DFA. Se la soglia non è isolata riusciamo a riconoscere una strana classe di linguaggi, che però ora non ci interessa.

2. Lezione 13 [09/04]

2.1. Varianti di automi

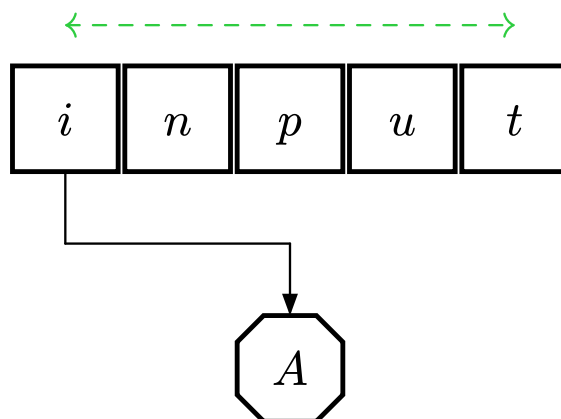
Come possiamo **rappresentare** un automa a stati finiti? Questa macchina è molto semplice: abbiamo un **nastro** che contiene l'input, esaminato da una **testina in sola lettura** che, spostandosi **one-way** da sinistra verso destra, permette ad un **controllo a stati finiti** di capire se la stringa in input deve essere accettata o meno.



La classe di linguaggi che riconosce un automa a stati finiti è la classe dei **linguaggi regolari**. Ma possiamo fare delle modifiche a questo modello? Se sì, che cosa possiamo cambiare?

2.1.1. One-way VS two-way

Se permettiamo all'automa di spostarsi da sinistra verso destra ma anche viceversa, andiamo ad ottenere gli **automi two-way**, che in base alla possibilità di leggere e basta o leggere e scrivere e in base alla lunghezza del nastro saranno in grado di riconoscere diverse classi di linguaggi.



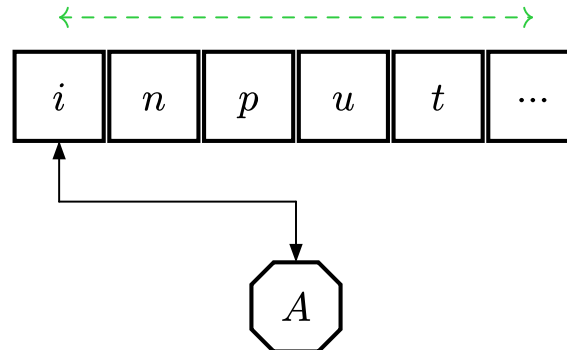
2.1.2. Read-only VS read-write

Se manteniamo l'automa one-way, rendere il nastro anche in lettura non modifica per niente il comportamento dell'automa: infatti, anche se scriviamo, visto che siamo one-way non riusciremo mai a leggere quello che abbiamo scritto.

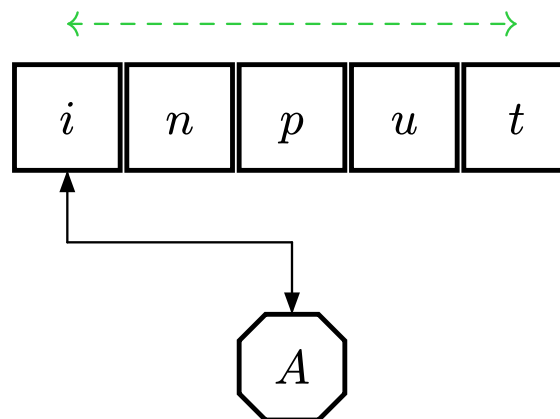
Consideriamo quindi un automa two-way che però mantiene la read-only del nastro: la classe che otteniamo è ancora una volta quella dei **linguaggi regolari**, e questo lo vedremo oggi.

Rendiamo ora la testina capace di poter scrivere sul nastro che abbiamo a disposizione. Ora, in base a come è fatto il nastro abbiamo due situazioni:

- se rendiamo il nastro illimitato oltre la porzione occupata dall'input, andiamo ad riconoscere i linguaggi di tipo 0, ovvero otteniamo una **macchina di Turing**:



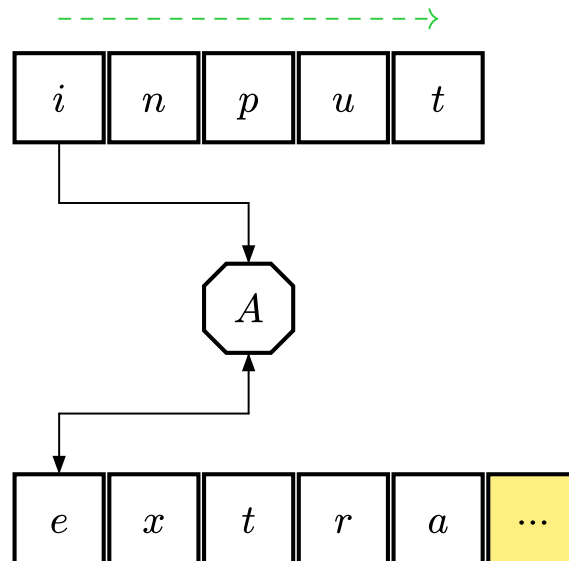
- se invece lasciamo il nastro grande quando l'input andiamo a riconoscere i linguaggi di tipo 1, ovvero otteniamo un **automa limitato linearmente**. Quest'ultima cosa vale perché nelle grammatiche di tipo 1 le regole di produzione non decrescono mai, e un automa limitato linearmente per capire se deve accettare cerca di costruire una derivazione al contrario, accorciando mano a mano la stringa arrivando all'assioma S :



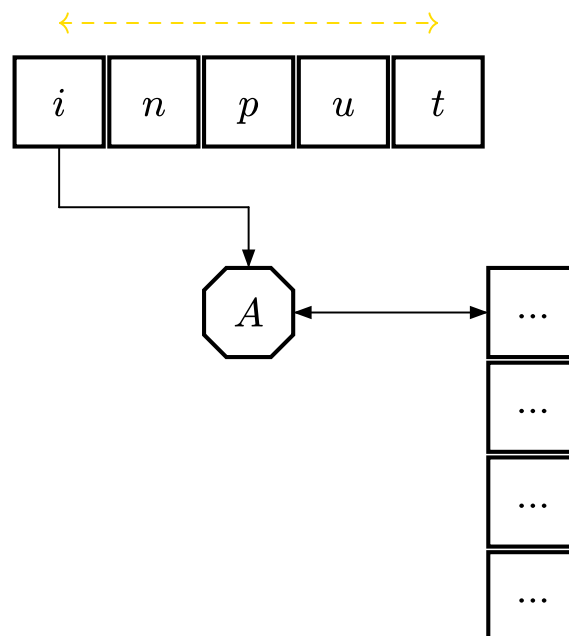
2.1.3. Memoria esterna

L'ultima modifica che possiamo pensare per queste macchine è l'aggiunta di una **memoria esterna**.

Dato un automa one-way con nastro read-only, se aggiungiamo un secondo nastro in read-write che funga da memoria esterna, otteniamo le due situazioni che abbiamo visto per gli automi two-way con possibilità di scrivere sul nastro di input.



Un caso particolare è se la memoria esterna è codificata come una **pila** illimitata, ovvero riesco a leggere solo quello che c'è in cima, allora andiamo a riconoscere i linguaggi di tipo 2, ottenendo quindi un **automa a pila**. Se passiamo infine ad un two-way con una pila diventiamo più potenti ma non sappiamo di quanto.



2.2. Automi two-way

Tra tutte queste varianti, fissiamoci sugli **automi two-way**, ovvero quelli che hanno il nastro in sola lettura e hanno la possibilità di andare avanti e indietro nell'input. Vediamo prima di tutto qualche linguaggio per il quale possiamo usare un automa two-way.

2.2.1. Esempi vari

Esempio 2.2.1.1: Riprendiamo l'operazione α . Dato L regolare, α era tale che

$$\alpha(L) = \{x \in \Sigma^* \mid xx \in L\}.$$

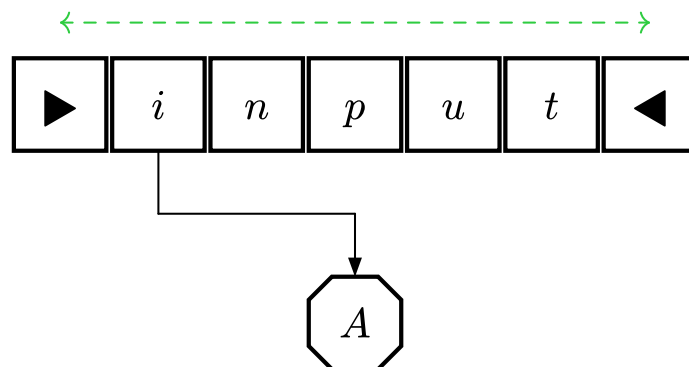
Abbiamo A un DFA che accetta L , come posso costruire un two-way per $\alpha(L)$? Potremmo leggere x la prima volta, ricordarci in che stato siamo arrivati, tornare indietro e poi ripartire a leggere x dallo stato nel quale eravamo arrivati e vedere se finiamo in uno stato finale.

Per fare ciò, ci serve sapere dove finisce il nastro: vedremo come fare tra poco.

Il numero di stati nel two-way è $3n$:

- n stati di A che usiamo per leggere x ;
- n stati che tengono traccia dello stato nel quale siamo arrivati con x e che ci permettono di ritornare all'inizio della stringa;
- n stati che fanno ripartire la computazione dallo stato nel quale siamo arrivati con x e controllano se finiamo in uno stato finale.

Abbiamo sollevato poco fa il problema: come facciamo a capire dove finisce il nastro? Andiamo a inserire dei **marcatori**, uno a sinistra e uno a destra, che delimitano la stringa. Se per caso arriviamo su un marcatore non possiamo andare oltre: possiamo solo rientrare sul nastro. In realtà, vedremo che in un particolare caso usciremo dai bordi.



Vediamo ancora un po' di esempi.

Esempio 2.2.1.2: Definiamo

$$L_n = (a + b)^* a (a + b)^{n-1}$$

il solito linguaggio dell' n -esimo simbolo da destra uguale ad una a .

Avevamo visto che con un NFA avevamo $n + 1$ stati, mentre con un DFA avevamo 2^n stati perché ci ricordavamo una finestra di n simboli. Ora diventa tutto più facile: ci spostiamo, ignorando completamente la stringa, sul marcatore di destra, poi contiamo n simboli e vediamo se accettare o rifiutare.

Come numero di stati siamo circa sui livelli dell'NFA, visto che dobbiamo solo scorrere la stringa per intero e poi tornare indietro di n .

Esempio 2.2.1.3: Definiamo infine

$$K_n = (a + b)^* a (a + b)^{n-1} a (a + b)^*$$

il linguaggio delle parole che hanno due a distanti n . Come lo scriviamo un two-way per K_n ?

Potremmo partire dall'inizio e scandire la stringa x . Ogni volta che troviamo una a andiamo a controllare n simboli dopo e vediamo se troviamo una seconda a :

- se sì, accettiamo;
- se no, torniamo indietro di $n - 1$ simboli per andare avanti con la ricerca.

Il numero di stati è:

- 1 che ricerca le a ;
- n stati per andare in avanti;
- 1 stato di accettazione;
- $n - 1$ stati per tornare indietro.

Ma allora il numero di stati è $2n + 1$.

Abbiamo trovato una buonissima soluzione per l'esempio precedente, ma se volessimo una soluzione alternativa che utilizza un automa sweeping? Ma cosa sono ste cose?

Un **automa sweeping** è un automa che non cambia direzione mentre si trova nel nastro, ma è un automa che rimbalza avanti e indietro sugli end marker. La soluzione che abbiamo trovato non usa automi sweeping perché se il simbolo a distanza n è una b noi invertiamo la direzione e torniamo indietro.

Esempio 2.2.1.4: Cerchiamo una soluzione che utilizzi un automa sweeping per K_n .

Supponiamo di numerare le celle del nastro da 1 a k . Partendo nello stato 1, andiamo a guardare tutte le celle a distanza n : se troviamo una a e poi subito dopo ancora una a accettiamo, altrimenti andiamo avanti fino a quando rimbalziamo sul marker, tornando indietro e andando sulla cella 2. Da qui facciamo ripartire la computazione, andando ogni volta avanti di una cella.

In generale, dalla cella $p \in \{1, \dots, n\}$ noi visitiamo tutte le celle $tn + p$.

Con questo approccio, il numero di stati è $O(n^2)$ perché dobbiamo muoverci di n simboli un numero n di passate. Possiamo farlo con un numero lineare di stati se facciamo la ricerca modulo n anche al ritorno, ma è questo è negli esercizi.

2.2.2. Definizione formale

Abbiamo visto come è costruito un automa two-way, ora vediamo la definizione formale. Definiamo

$$M = (Q, \Sigma, \delta, q_0, q_f)$$

un **2NFA** tale che:

- Q rappresenta l'insieme degli stati;

- Σ rappresenta l'**alfabeto** di input;
- q_0 rappresenta lo **stato iniziale**;
- δ rappresenta la **funzione di transizione**, ed è tale che

$$\delta : Q \times (\Sigma \cup \{\blacktriangleright, \blacktriangleleft\}) \longrightarrow 2^{Q \times \{+1, -1\}},$$

ovvero prende uno stato e un simbolo dell'alfabeto compresi gli end marker e ci restituisce i nuovi stati e che movimento dobbiamo fare con la testina. Ho dei **divieti**: se sono sull'end marker sinistro non ho mosse che mi portano a sinistra, idem ma specchiato su quello di destra con una piccola eccezione, che vediamo tra poco;

- q_f è lo **stato finale** e si raggiunge «passando» oltre l'end marker destro, unico caso in cui si può superare un end marker.

Con questo modello possiamo incappare in **loop infiniti**, che:

- nei DFA non ci fanno accettare;
- negli NFA magari indicano che abbiamo fatto una scelta sbagliata e c'era una via migliore.

Ci sono poi diverse modifiche che possiamo fare a questo modello, ad esempio:

- possiamo estendere le mosse con la **mossa stazionaria**, ovvero quella codificata con 0 che ci mantiene nella posizione nella quale siamo, ma possono essere eliminate con una coppia di mosse sinistra+destra o viceversa;
- possiamo utilizzare un **insieme di stati finali**;
- possiamo **non** usare gli end marker, rendendo molto difficile la scrittura di automi perché non sappiamo dove finisce la stringa.

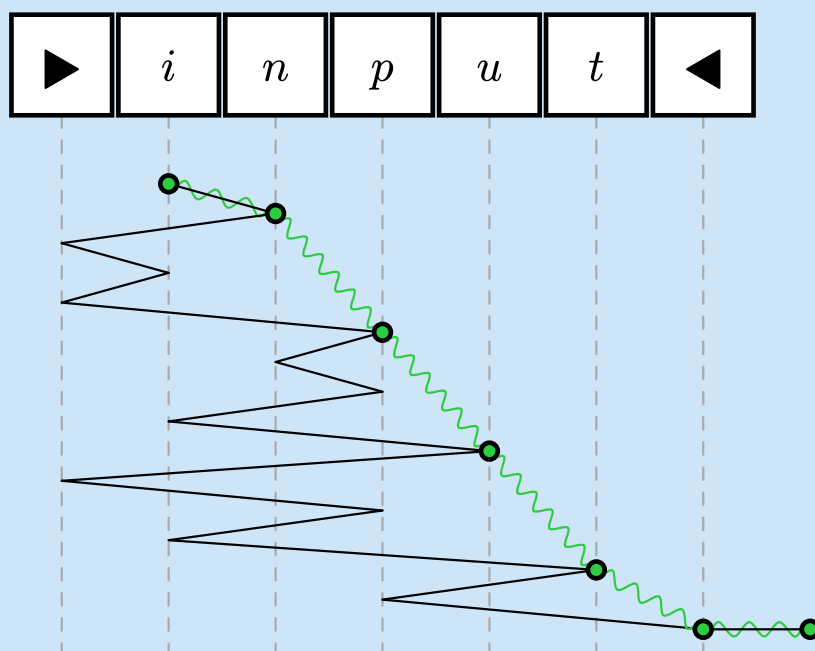
2.2.3. Potenza computazionale

Avere a disposizione un two-way sembra darci molta potenza, ma in realtà non è così: infatti, questi modelli sono equivalenti agli automi a stati finiti one-way, detti anche **1DFA**.

Teorema 2.2.3.1: Vale

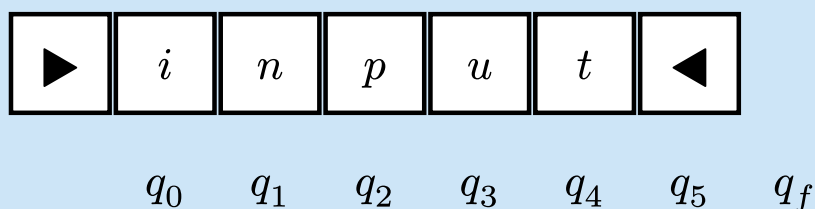
$$L(2DFA) = L(1DFA).$$

Dimostrazione 2.2.3.1.1: Abbiamo a disposizione un 2DFA nel quale abbiamo inserito un input che viene accettato. Vogliamo cambiare la computazione del 2DFA in una computazione di un 1DFA. Vediamo che stati vengono visitati nel tempo.



Prima di tutto, dobbiamo ricordarci che nei DFA non abbiamo end marker, quindi abbiamo solo l'input. Nell'automa two-way ci sono momenti dove entro nelle celle per la prima volta: nel grafico sopra sono segnati in verde. Chiamiamo questi stati $q_{i \geq 0}$.

Usiamo delle **scorciatoie**: visto che nel 1DFA non possiamo andare avanti a indietro, dobbiamo tagliare via le computazioni che tornano indietro e vedere solo in che stato esco.



Come vediamo, a me interessa sapere in che stato devo spostarmi a partire dalla mia posizione, evitando quello che viene fatto tornando all'indietro. Per tagliare le parti che tornano indietro usiamo delle **matrici**, molto simili a quelle della lezione precedente. Quelle matrici erano nella forma $M_w[p, q]$ che conteneva un 1 se e solo se partendo da q finivo in p leggendo w .

Le matrici che costruiamo ora sono nella forma

$$\tau_w : Q \times Q \longrightarrow [0, 1]$$

che mi vanno a definire il primo stato che incontriamo quando leggiamo un nuovo carattere della stringa.

Nella matrice abbiamo $\tau_w[p, q] = 1$ se e solo se esiste una sequenza di mosse che:

- inizia sul simbolo più a destra della porzione di nastro che contiene $(\blacktriangleright w)$ nello stato p ;

- termina quando la testina esce a destra dalla porzione di nastro considerata nello stato q .

Ad esempio, considerando l'esempio sopra, vale

$$\tau_{\text{inp}}[q_2, q_3] = 1.$$

Vediamo come ottenere induttivamente queste tabelle. Partiamo con $w = \varepsilon$: la porzione di nastro che stiamo considerando è formata solo da \blacktriangleright , ma non potendo andare a sinistra l'unica mossa che possiamo fare è andare a destra, quindi andare in un nuovo stato, ovvero

$$\tau_\varepsilon[p, q] = 1 \iff \delta(p, \blacktriangleright) = (q, +1).$$

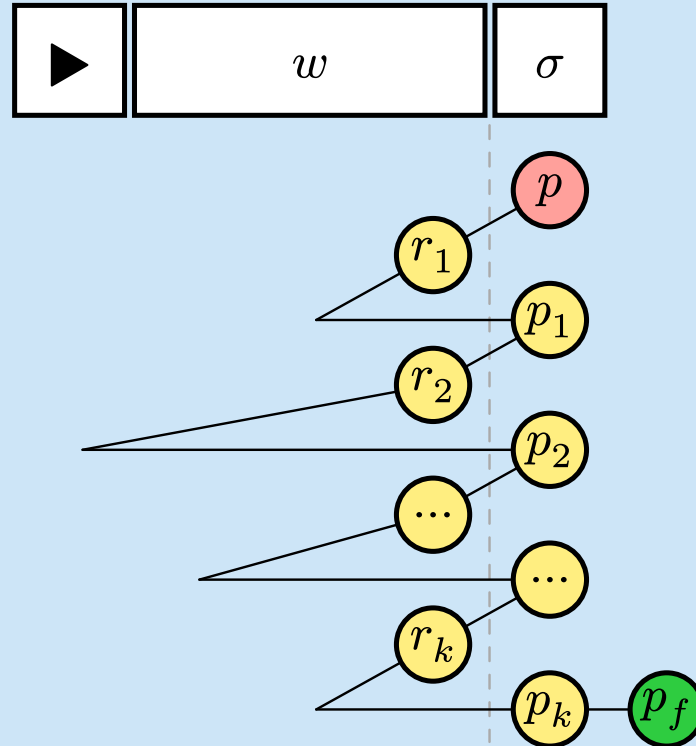
Supponiamo di aver calcolato la tabella di w , vediamo come costruire induttivamente la tabella di $w\sigma$, con $w \in \Sigma^*$ e $\sigma \in \Sigma$. Se vale

$$\delta(p, \sigma) = (q, +1)$$

la tabella è molto facile, perché sto subito uscendo dallo stato p , ovvero

$$\tau_{w\sigma}[p, q] = 1.$$

Se invece andiamo indietro dobbiamo capire cosa fare.



Ogni volta che da p_i torniamo indietro finiamo in uno stato r_{i+1} , che poi dopo un po' di giri finisce per forza in p_{i+1} . Andiamo avanti così, fino ad un certo p_k , dal quale usciamo e andiamo in q . In poche parole

$$\tau_{w\sigma}[p, q] = 1$$

se e solo se esiste una sequenza di stati

$$p_0, p_1, \dots, p_k, r_1, \dots, r_k \mid k \geq 0$$

tale che:

- $p_0 = p$, ovvero parto dallo stato p , per definizione;
- $\delta(p_{i-1}, \sigma) = (r_i, -1) \quad \forall i \in \{1, \dots, k\}$, ovvero in tutti i p tranne l'ultimo io torno indietro;
- $\tau_w[r_i, p_i] = 1$, ovvero da r_i giro in w e poi torno in p_i ;
- $\delta(p_k, \sigma) = (q, +1)$, ovvero esco fuori dal $w\sigma$.

Notiamo che se prendiamo $k = 0$ abbiamo la situazione precedente in cui uscivo direttamente. Inoltre, k è il numero massimo di stati del DFA perché se faccio ancora un giro in w dopo p_k vado in uno stato già visto ed entro in un loop infinito.

Notiamo una cosa **importantissima**: se due stringhe hanno la stessa tabella, ovvero $\tau_w = \tau_{w'}$, allora l'aggiunta di un qualsiasi carattere σ genera tabelle risultanti uguali, ovvero $\tau_{w\sigma} = \tau_{w'\sigma}$. Ma allora esiste una funzione

$$f_\sigma : [Q \times Q \rightarrow [0, 1]] \rightarrow [Q \times Q \rightarrow [0, 1]]$$

che genera una tabella a partire da una data, ed è tale che

$$\forall w \in \Sigma^* \quad \tau_{w\sigma} = f_\sigma(\tau_w).$$

In poche parole, la nuova tabella dipende solo da σ e non da w , e questa tabella è esattamente quella calcolata con i 4 punti messi sopra. Le tabelle, inoltre, sono tantissime ma sono un numero finito.

Siamo pronti per costruire il 1DFA che tanto stiamo bramando. Noi avevamo $M = (Q, \Sigma, \delta, q_0, F)$ che è un 2DFA, vogliamo costruire

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

1DFA che sia equivalente a M . Esso è tale che:

- Q è l'**insieme degli stati** e lo usiamo tenere traccia dello stato nel quale siamo e della tabella che usiamo per calcolare lo stato successivo, ovvero

$$Q' = Q \times [Q \times Q \rightarrow [0, 1]];$$

- q'_0 è lo **stato iniziale** ed è la coppia

$$q'_0 = (q_0, \tau_\epsilon);$$

- δ' è la **funzione di transizione** che manda avanti l'automa, ovvero

$$\delta'((p, T), \sigma) = (q, T')$$

con:

- T' che mi dà indicazioni sullo stato nel quale arrivo con σ , che ho però appena letto, quindi $T' = f_\sigma(T)$;
- vale $T'[p, q] = 1$ perché io devo uscire in q partendo da p ;

- F' è l'**insieme degli stati finali**, ostico perché nel two-way abbiamo gli end marker, nel one-way non li abbiamo. Per accettare dovevo sfiorare l'end marker di destra e finire in q_f , ma questa informazione la ricavo dalla tabella del right marker, ovvero

$$F' = \{(q, T) \mid (f_{\blacktriangleleft}(T))[q, q_f] = 1\}.$$

Ma allora stiamo simulando un 2DFA con un 1DFA, ma gli 1DFA riconoscono la classe dei linguaggi regolari, quindi anche la classe degli automi a stati finiti two-way riconosce la classe dei linguaggi regolari. ■

Che considerazioni possiamo fare sul numero di stati? Sappiamo che:

- il numero di stati è $|Q| = n$;
- il numero di tabelle è $|[Q \times Q] \rightarrow [0, 1]| = 2^{n^2}$.

Ma allora il numero di stati è

$$|Q'| \leq n2^{n^2}.$$

Come vediamo, la simulazione è **poli-esponenziale**.

3. Lezione 14 [11/04]

3.1. Simulazione

La scorsa lezione abbiamo visto gli automi two-way e abbiamo dimostrato che hanno la stessa potenza computazionale degli automi a stati finiti. Avevamo visto la trasformazione da 2DFA a 1DFA, ma la stessa trasformazione può essere fatta per il passaggio da 2NFA a 1NFA.

Ma quanto costano queste trasformazioni?

Nel caso partissimo da un 2DFA e volessimo arrivare in un 1DFA, il costo in termini di stati è

$$\leq \dots,$$

mentre cambiando il punto di partenza con un 2NFA il salto diventa ancora peggiore:

$$\leq 2^n 2^{n^2} = 2^{n^2+n}.$$

Ma questo ce lo potevamo aspettare: abbiamo già un salto esponenziale da NFA a DFA, quindi ciao.

Ci sono due simulazioni che sono però molto particolari e importanti.

3.1.1. Problema di Sakoda & Sipser

La prima trasformazione che vediamo è quella da 2NFA a 2DFA: qua non possiamo usare la costruzione per sottoinsiemi perché ad un certo punto potresti avere il non determinismo su una mossa che però mi sposta la testina su due caratteri diversi della stringa, e questo non è possibile. Ci serve quindi una trasformazione alternativa, ma ci arriviamo dopo.

La seconda trasformazione è quella da 1NFA a 2DFA: questa trasformazione cerca di capire se, dando il two-way ad un automa deterministico, esso è capace di simulare il non determinismo.

Vediamo un paio di esempi.

Esempio 3.1.1.1: Definiamo

$$L_n = (a + b)^* a (a + b)^{n-1}$$

il classicissimo linguaggio dell' n -esimo carattere da destra pari ad una a .

Sappiamo che:

- esiste un 1NFA di $n + 1$ stati;
- esiste un 1DFA di 2^n stati.

Abbiamo visto un automa two-way per questo linguaggio, che usa poco più di n stati, quindi in questo caso riusciamo a togliere il non determinismo a basso costo.

Esempio 3.1.1.2: Definiamo

$$K_n = (a + b)^* a (a + b)^{n-1} a (a + b)^*$$

il solito linguaggio con due a a distanza n .

Avevamo visto che un 1NFA per questo linguaggio usava $n + 2$ stati, quindi una quantità lineare in n . Per un 2DFA abbiamo visto che esiste anche qui una soluzione lineare in n , quindi anche qui eliminiamo il non determinismo a basso costo.

Abbiamo visto due esempi che sembrano dare buone notizie, ma riusciamo a dimostrare che si riesce sempre a fare un 2DFA di n stati partendo da un 1NFA di n stati? Purtroppo, nessuno ci è mai riuscito.

Questi problemi sono i **problemi di Sakoda & Sipser**, ideati nel 1978 e che riguardano il costo della simulazioni di automi non deterministici one-way e two-way per mezzo di automi two-way deterministici, ovvero si chiedono se il movimento two-way aiuta nell'eliminazione del non determinismo.

Cosa sappiamo su questi problemi? Diamo qualche **upper** e **lower bound**.

Per il problema da 1NFA a 2DFA, si sfrutta la **costruzione per sottoinsiemi** per ottenere un 1DFA, che è anche un 2DFA che non torna mai indietro, ottenendo quindi un numero di stati

$$\leq 2^n.$$

Un lower bound per questo problema invece è

$$\geq n^2.$$

Per il problema da 2NFA a 2DFA, si fa un passaggio intermedio all'1NFA e poi al 1DFA, che come prima è anche 2DFA, quindi gli stati sono

$$\leq 2^{n^2+n}.$$

Il lower bound, invece, è lo stesso del problema precedente.

Ci sono casi particolari che hanno delle dimostrazioni precise:

- se utilizziamo dei 2DFA sweeping il costo per la trasformazione è **esponenziale**, ma questo non risolve il problema perché (???) ci sono automi non sweeping che per diventarlo hanno un salto esponenziale (???)
- se $\Sigma = \{a\}$:
 - ▶ se facciamo la trasformazione da 2NFA a 2DFA l'upper bound è

$$e^{O(\log^2(n))},$$

ovvero una funzione super polinomiale ma meno di una esponenziale. Inoltre, se si dimostra che esiste un lower bound super polinomiale, allora abbiamo dimostrato che

$$L = NL \text{ (???)};$$

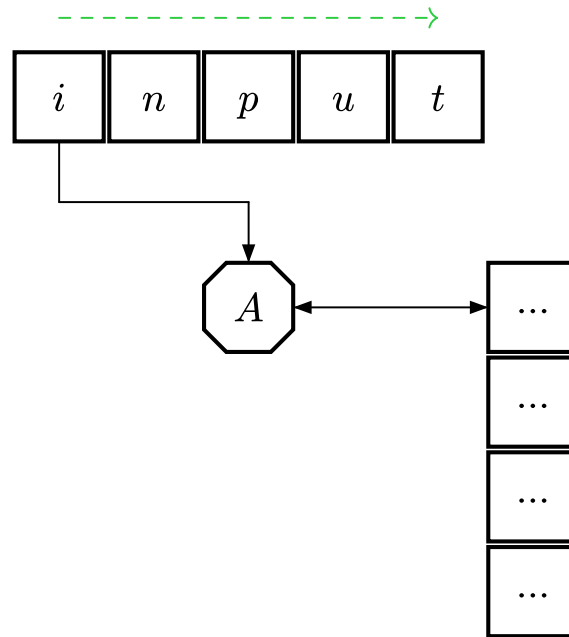
- ▶ se facciamo la trasformazione da 1NFA a 2DFA l'upper bound diventa esattamente n^2 , quindi la trasformazione fatta è ottimale.

Dei ricercatori hanno trovato degli **automi completi** per questi problemi, ovvero degli automi che permettono lo studio dei problemi solo su questi pochi automi scelti per poi far «arrivare» tutte le conseguenze a tutti gli altri automi. Scritto malissimo, sono tipo gli NP-completi.

Per ora, la **congettura** che circola tra la gente è che i costi siano **esponenziali nel caso peggiore**.

3.2. Automi a pila

Lasciamo finalmente stare gli automi a stati finiti per passare ad una nuova classe di riconoscitori: gli **automi a pila**. Essi sono praticamente degli automi a stati finiti con testina di lettura one-way ai quali viene aggiunta una **memoria infinita con restrizioni di accesso**, ovvero l'accesso avviene solo sulla cima della memoria, con politica LIFO.



Come vediamo, la parte degli automi a stati finiti ce l'abbiamo ancora, ma ora abbiamo una **memoria esterna**, che nell'immagine è sulla destra, che possiamo utilizzare con una politica di accesso LIFO. Per via di questa politica, questi automi sono anche detti **automi pushdown**, o **PDA**, perché quando inserisci qualcosa lo fai a spingere giù.

3.2.1. Versione non deterministica

Vediamo subito la definizione formale **non deterministica** dei PDA.

Sia M un PDA definito dalla tupla

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

tale che:

- Q è un **insieme finito non vuoto di stati**, che rappresenta il controllo a stati finiti;
- Σ è un **alfabeto finito non vuoto di input**;
- Γ è un **alfabeto finito non vuoto di simboli della pila**;
- δ è la **funzione di transizione**;
- $q_0 \in Q$ è lo **stato iniziale**;
- $Z_0 \in \Gamma$ è il **simbolo iniziale sulla pila**;
- $F \subseteq Q$ è un **insieme di stati finali**.

La **funzione di transizione** è definita come segue:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow 2^{Q \times \Gamma^*}.$$

In poche parole, consideriamo lo **stato corrente**, il **simbolo sulla testina** o una ε -mossa e il **simbolo sulla cima della pila** per capire in che stato dobbiamo muoverci e che stringa andare ad inserire sulla pila. La lettura del carattere in cima alla pila lo va a **distruggere**.

Questa versione però non ci piace molto perché Γ^* è potenzialmente un **insieme infinito**, e non ci piace avere un insieme infinito di possibilità, quindi sostituiamo la definizione della funzione di transizione con questa analoga, ma molto migliore per noi:

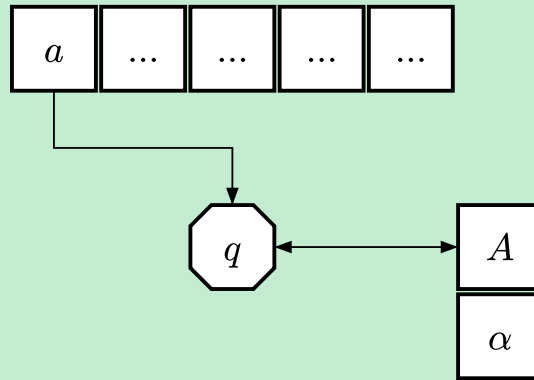
$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow \text{PF}(Q \times \Gamma^*).$$

Con PF intendiamo l'**insieme delle parti finite**, ovvero un insieme finito di possibilità prese dall'insieme delle parti. Ora sì che la definizione ci piace.

Facciamo qualche esempio. Come convenzione useremo le **maiuscole** per i simboli della pila.

Esempio 3.2.1.1: Facciamo che la funzione di transizione sia definita in questo modo:

$$\delta(q, a, A) = \{(q_1, \varepsilon), (q_2, BCC)\}.$$

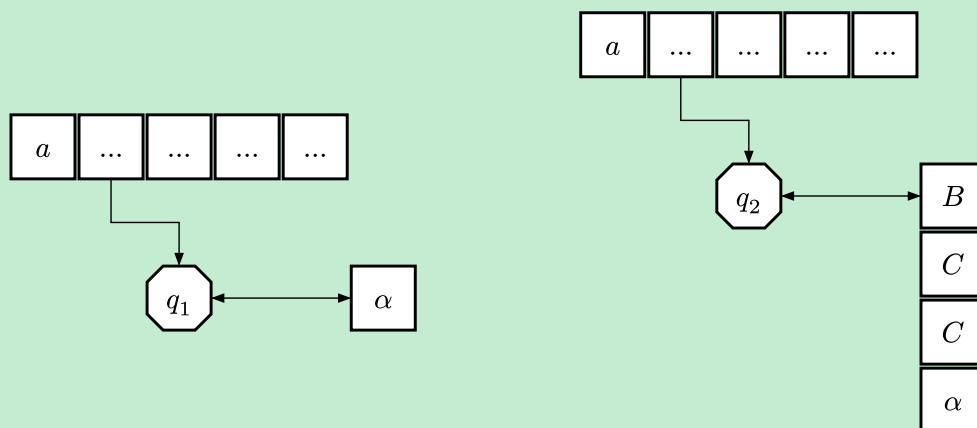


Con α nel disegno si intende una stringa in Γ^* perché oltre ad A potremmo avere altro.

Cosa vuol dire quella regola della funzione di transizione? Ci sta dicendo che se ci troviamo nello stato q , leggiamo a sul nastro leggiamo A sulla cima della pila, possiamo:

- andare in q_1 e non mettere altro sulla pila, praticamente consumando un simbolo in input;
- andare in q_2 e mettere sulla pila la stringa BCC .

Vediamo la rappresentazione dei due casi nei quali possiamo finire.

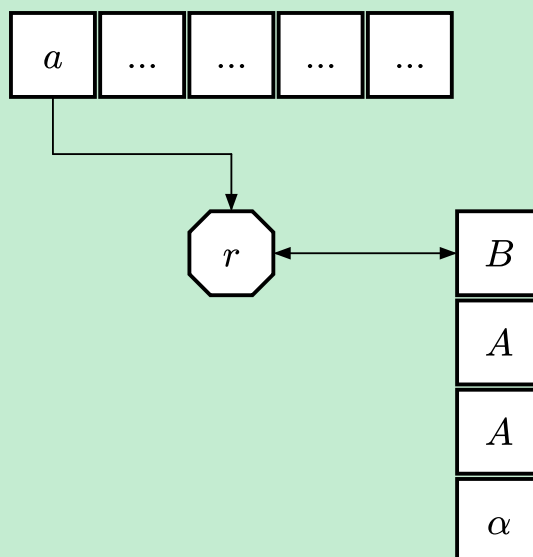


Per convenzione, quando inseriamo una stringa sulla pila, l'inserimento avviene da destra verso sinistra. In poche parole, se inseriamo la stringa $X \in \Gamma^*$ nella pila, se la togliessimo noi leggeremmo, in ordine, esattamente X . In altre parole ancora, quando leggiamo una stringa da inserire è come se la stessimo leggendo dall'alto verso il basso.

Abbiamo la possibilità anche di fare delle ε -mosse: supponiamo di aggiungere la regola

$$\delta(q, \varepsilon, A) = \{(r, BAA)\}.$$

Ora abbiamo tre scelte a disposizione. Le ε -mosse possiamo vederle come delle **mosse interne**, che avvengono senza leggere l'input, e che ci permettono di spostarci negli stati modificando eventualmente la pila.



Una **configurazione** è una fotografia dell'automa in un dato istante di tempo, e ci dice quali sono le informazioni rilevanti per il futuro per definire al meglio la macchina, ovvero:

- lo **stato corrente**;
- il **contenuto del nastro** che ci manca da leggere;
- il **contenuto della pila**.

Una configurazione è quindi una **tripla**

$$(q, ay, A\alpha)$$

che contiene lo stato corrente, il contenuto del nastro ancora da leggere indicato dal carattere corrente a unito al resto della stringa y e il contenuto della pila indicato dal carattere in testa A e dal resto della pila α .

Una **mossa** è l'applicazione della funzione di transizione, ovvero un passaggio

$$(q, ay, A\alpha) \longrightarrow (p, y, \gamma\alpha) \iff (p, \gamma) \in \delta(q, a, A).$$

Analogamente, un passaggio che usa le ε -mosse è un passaggio

$$(q, ay, A\alpha) \longrightarrow (p, ay, \gamma\alpha) \longrightarrow (p, \gamma) \in \delta(q, \varepsilon, A).$$

Una **computazione** è una serie di mosse che partono da una configurazione iniziale e mi portano in una configurazione finale. Di queste ultime parleremo tra poco. Torniamo sulle computazioni.

Come con i passi di derivazione, una computazione che usa una sola mossa si indica con

$$C' \vdash C''.$$

Se invece una computazione impiega k passi, si indica con

$$C' \vdash^k C''.$$

Infine, per indicare una computazione con un numero generico di passi, maggiori o uguali a zero, si usa

$$C' \vdash^* C''.$$

3.2.2. Accettazione

Abbiamo parlato di arrivare in una configurazione accettante, ma quando **accettiamo**? Dobbiamo capire da dove partire e dove arrivare.

Quando partiamo abbiamo la stringa w sul nastro, ci troviamo nello stato iniziale q_0 e abbiamo Z_0 sulla pila: questa è detta **configurazione iniziale** ed è la tripla

$$(q_0, w, Z_0).$$

Le **configurazioni finali** dipendono dal tipo di nozione di accettazione che vogliamo utilizzare.

L'**accettazione per stati finali** ci obbliga a leggere tutto l'input e a finire in uno stato finale, con la pila che contiene quello che vuole, ovvero dobbiamo arrivare in una configurazione

$$(q, \varepsilon, \gamma)$$

dove lo stato q è finale. Il **linguaggio accettato per stati finali** è l'insieme

$$L(M) = \left\{ w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \gamma) \mid q \in F \wedge \gamma \in \Gamma^* \right\}.$$

Questa nozione è comoda perché vede i PDA come una **estensione** degli automi a stati finiti.

L'**accettazione per pila vuota** invece è una nozione più naturale: tutto ciò che metto nella pila lo devo anche buttare via. Possiamo arrivare in un qualsiasi stato, basta aver svuotato la pila. Il **linguaggio accettato per pila vuota** è l'insieme

$$N(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash (q, \varepsilon, \varepsilon) \mid q \in Q\}.$$

Se svuotiamo la pila prima di finire l'input allora quella computazione si blocca perché noi dobbiamo sempre leggere qualcosa dalla pila.

Le due accettazioni sono **equivalenti**, o meglio, possiamo passare da una accettazione all'altra ma i linguaggi che accettano sono differenti. A parità di automa M , gli insiemi $L(M)$ e $N(M)$ in generale sono diversi, ma possiamo passare da un modello all'altro mantenendo il linguaggio accettato con facilità. Una ulteriore nozione di accettazione unisce stati finali e pila vuota, ma rimane comunque equivalente. Avere due nozioni è comodo: se una versione ci esce estremamente comoda allora la andiamo ad utilizzare, altrimenti andremo ad utilizzare l'altra.

Vediamo ora qualche esempio.

Esempio 3.2.2.1: Prendiamo il nostro migliore amico, il linguaggio

$$L = \{a^n b^n \mid n \geq 1\}.$$

Lo possiamo riconoscere con un PDA, visto che abbiamo visto che non è regolare? Bhe sì: con i DFA non riusciamo a ricordare il numero di a e poi confrontare questo numero con le b , mentre ora riusciamo a farlo, le pile sanno contare.

Possiamo pensare ad un automa che ogni volta che legge una a butta una A dentro la pila, e quando legge una b toglie una A dalla pila. Accettiamo se abbiamo messo n caratteri A dentro la pila e poi ne abbiamo tolti n , quindi qua viene comoda l'**accettazione per pila vuota**.

Andiamo a definire la funzione di transizione.

Iniziamo a togliere Z_0 dalla prima e inseriamo la prima A in segno di aver letto la prima a della stringa, che abbiamo per forza per definizione, quindi

$$\delta(q_0, a, Z_0) = \{(q_0, A)\}.$$

Utilizziamo lo stato q_0 per leggere tutte le a della stringa, ovvero

$$\delta(q_0, a, A) = \{(q_0, AA)\}.$$

Appena troviamo una b iniziamo a cancellare e cambiamo stato, visto che non ci aspettiamo più delle a nella stringa, quindi

$$\delta(q_0, b, A) = \{(q_1, \varepsilon)\}.$$

Inseriamo ε sulla stringa perché la A da cancellare per la lettura di b è già stata cancellata dalla lettura.

Andiamo a terminare la lettura delle b , quindi

$$\delta(q_1, b, A) = \{(q_1, \varepsilon)\}.$$

Abbiamo detto che accettiamo per pila vuota, quindi $L = N(M)$.

Esempio 3.2.2.2: Se invece volessimo accettare il linguaggio precedente per **stati finali**?

Non dobbiamo cancellare Z_0 dalla pila perché se una stringa viene accettata cancella tutta la pila, quindi ci serve un carattere fittizio dentro per poterlo leggere e spostarci in uno stato finale. Modifichiamo quindi la mossa iniziale con la mossa

$$\delta(q_0, a, Z_0) = \delta(q_0, AZ_0).$$

Se abbiamo una stringa del linguaggio alla fine delle b dobbiamo spostarci in uno stato finale, quindi aggiungiamo la regola

$$\delta(q_1, \varepsilon, Z_0) = \{(q_f, Z_0)\} \mid q_f \in F.$$

Con queste modifiche abbiamo $L = L(M)$.

Esempio 3.2.2.3: Se invece volessimo accettare anche ε ? Il linguaggio diventa

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

Con l'**accettazione per pila vuota**, nello stato iniziale possiamo aggiungere una regola che svuota subito la pila, ovvero aggiungiamo la regola

$$\delta(q_0, \varepsilon, Z_0) = \{(q_0, \varepsilon)\}.$$

Stiamo scommettendo che l'input è già finito, ovvero abbiamo solo ε sul nastro, ma questo ha appena aggiunto il **non determinismo** al nostro automa a pila.

Con l'**accettazione per stati finali** invece ci spostiamo direttamente nello stato q_f a partire da q_0 , ovvero aggiungiamo la regola

$$\delta(q_0, \varepsilon, Z_0) = \{(q_f, \varepsilon)\}.$$

Come prima, abbiamo aggiunto del **non determinismo** all'automa a pila, ma questo lo possiamo togliere: come facciamo a fare ciò?

Introduciamo uno stato q_I finale che diventa anche iniziale al posto di q_0 , quindi ora

$$F = \{q_I, q_f\}.$$

Se inseriamo sul nastro la stringa vuota allora noi accettiamo, perché siamo in uno stato finale e non abbiamo altri simboli da leggere. Per passare poi al vecchio automa mettiamo una regola

$$\delta(q_I, a, Z_0) = \{(q_0, AZ_0)\}.$$

3.2.3. Determinismo VS non determinismo

Con il termine **non determinismo** non intendiamo le ε -mosse da sole, quelle le possiamo avere, ma intendiamo un mix tra mosse che leggono e mosse che non leggono.

Definizione 3.2.3.1 (Determinismo): Sia M un PDA. Allora M è **deterministico** se:

1. ogni volta che ho una ε -mossa da un certo stato e con un certo simbolo sulla pila, non ho mosse che leggono simboli dal nastro a partire dallo stesso stato e con lo stesso simbolo sulla pila, ovvero

$$\forall q \in Q \quad \forall A \in \Gamma \quad \delta(q, \varepsilon, A) \neq \emptyset \implies \forall a \in \Sigma \quad \delta(q, a, A) = \emptyset;$$

2. come nel caso classico, considero un carattere, o anche ε , allora a parità di stato corrente e simbolo sulla pila, ho al massimo una transizione possibile, ovvero

$$\forall q \in Q \quad \forall A \in \Gamma \quad \forall \sigma \in \Sigma \cup \{\varepsilon\} \quad |\delta(q, \sigma, A)| \leq 1.$$

A differenza del caso classico, il determinismo e il non determinismo non sono ugualmente potenti: un automa a pila non deterministico è **più potente** di un automa a pila deterministico, che riconosce una sottoclasse di linguaggi diversa dai linguaggi di tipo 2, che sono riconosciuti dai PDA non deterministici.

3.2.4. Trasformazioni

Avevamo parlato dell'**equivalenza** dell'accettazione per stati finali e per pila vuota: infatti, esistono due trasformazioni che permettono di passare da un automa all'altro, mantenendo il linguaggio di partenza riconosciuto inalterato. L'equivalenza infatti ci diceva che, partendo da un automa M che riconosce per stati finali, abbiamo una trasformazione che ci dà M' che riconosce per pila vuota che riconosce lo stesso linguaggio di M , e viceversa.

Stati finali \rightarrow pila vuota Dobbiamo trasformare un automa che accetta per stati finali in un automa che accetta per pila vuota. Con quest'ultimo simuliamo il primo, e ogni volta che vado in uno stato finale mi sposto in uno **stato di svuotamento**, che se raggiunto in mezzo blocca la pila, ma se raggiunto alla fine mi fa accettare.

Abbiamo quindi $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA con $L = L(M)$. Definiamo ora

$$M' = (Q \cup \{q_e, q'_0\}, \Sigma, \Gamma \cup \{X\}, \delta', q'_0, X, \emptyset)$$

un PDA tale che:

- per metterci in una situazione piacevole per la fine usiamo un **truccaccio** definito dalla regola

$$\delta(q'_0, \varepsilon, X) = \{(q_0, Z_0 X)\},$$

ovvero prima di far partire la computazione dell'automa M andiamo ad inserire un carattere X in fondo alla pila, vedremo dopo perché;

- l'automa deve eseguire **tutte le mosse** di M , ovvero

$$\forall q \in Q \quad \forall \sigma \in \Sigma \cup \{\varepsilon\} \quad \forall Z \in \Gamma \quad \delta(q, \sigma, Z) \subseteq \delta'(q, \sigma, Z),$$

che scritto così significa che tutte le mosse che trovavamo nell'applicazione di delta ad una certa tripla le abbiamo anche nella nuova funzione di transizione, che però conterrà anche altro, che vedremo tra poco;

- aggiungiamo uno **stato di svuotamento** per pulire la pila, definito dalle regole

$$\begin{aligned}\forall q \in F \quad \forall Z \in \Gamma \cup \{X\} \quad (q_e, \varepsilon) &\in \delta'(q, \varepsilon, Z) \\ \forall Z \in \Gamma \cup \{X\} \quad \delta'(q_e, \varepsilon, Z) &= \{(q_e, \varepsilon)\},\end{aligned}$$

ovvero con la prima regola, ogni volta che mi trovo in uno stato finale **non deterministicamente** mi posso spostare nello stato di svuotamento, mentre con la seconda regola effettivamente svuoto.

A cosa ci serve il **carattere** X ? Facciamo finta di non mettere il carattere X . Se M accetta una stringa x arrivando con la pila vuota nessun problema, non ci spostiamo nello stato di svuotamento ma abbiamo la pila vuota quindi ottimo. Se invece M non accetta una stringa x ma arriva alla fine con la pila vuota, il simbolo X messo all'inizio ci copre da una eventuale accettazione errata, perché non riusciremo ad andare nello stato di svuotamento per avere la pila vuota, anche se M ci finisce in quel modo. Diciamo che abbiamo messo X come se fosse una guardia, che ci copre questo preciso caso.

Purtroppo, con questa costruzione abbiamo buttato dentro del **non determinismo** quando facciamo i passaggi in q_e da uno stato finale.

Pila vuota \rightarrow stati finali Il percorso opposto invece parte da un PDA

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

tale che $L = N(M)$. Definiamo il PDA

$$M' = (Q \cup \{q'_0, q_f\}, \Sigma, \Gamma \cup \{X\}, \delta', q'_0, X, \{q_f\})$$

che come idea ha quella di simulare M e, ogni volta che arriva con pila vuota, ci spostiamo nello stato finale. Vediamo i vari passi:

- come prima, usiamo un **truccaccio** per infilare X sotto la pila, quindi abbiamo la regola

$$\delta'(q'_0, \varepsilon, Z_0) = \{(q_0, Z_0 X)\}$$

che usiamo per inserire X come trigger per andare in uno stato finale;

- simuliamo l'automa M senza aggiungere niente, quindi

$$\forall q \in Q \quad \forall \sigma \in \Sigma \cup \{\varepsilon\} \quad \forall Z \in \Gamma \quad \delta'(q, \sigma, Z) = \delta(q, \sigma, Z);$$

- ogni volta che leggiamo X sulla cima della pila vuol dire che M ha svuotato la pila, quindi devo andare nello stato finale, ovvero

$$\forall q \in Q \quad \delta'(q, \varepsilon, X) = \{(q_f, \varepsilon)\};$$

ovviamente, se andiamo in questo stato a metà stringa ci blocchiamo, altrimenti se ci andiamo alla fine è tutto ok.

A differenza di prima, se partiamo da un automa **deterministico**, quello che otteniamo è ancora un automa **deterministico**.

4. Lezione 15 [23/04]

Vediamo degli esempi di qualche linguaggio che possiamo riconoscere con degli automi a pila.

4.1. Esempi

Esempio 4.1.1: Definiamo il linguaggio

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

Un automa a pila per questo linguaggio memorizza w sulla pila, legge $\#$ e poi verifica che la stringa w^R sia presente sulla pila.

Possiamo usare due stati:

- q_0 lo usiamo per copiare w sulla pila;
- q_1 lo usiamo per confrontare il carattere sulla pila con quello sul nastro.

In questo caso ci viene naturale accettare per pila vuota. Inoltre, otteniamo un automa deterministico, detto anche **DPDA**.

Un linguaggio riconosciuto da automi a pila deterministici DPDA fa parte dell'insieme dei **linguaggi context-free deterministici**, detti anche **DCFL**.

Esempio 4.1.2: Definiamo ora il linguaggio

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

insieme delle stringhe palindrome di lunghezza pari.

In questo caso non riusciamo a farlo con un DPDA (difficile da dimostrare, lo faremo avanti) perché dobbiamo scommettere di essere arrivati a metà della stringa da riconoscere, quindi dobbiamo usare del **non determinismo**.

Analogamente, un linguaggio riconosciuto da automi a pila non deterministici, detti anche **NPDA** o solo **PDA**, fa parte dell'insieme dei **linguaggi context-free**, detti anche **CFL**.

4.2. Equivalenza tra grammatiche di tipo 2 e automi a pila

Facciamo un breve ripasso sulle grammatiche di tipo 2 e poi andiamo a vedere l'equivalenza tra le grammatiche di tipo 2 e gli automi a pila.

4.2.1. Ripasso e introduzione

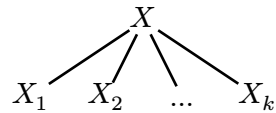
Una grammatica G di tipo 2 ha le **regole di produzione** nella forma

$$X \longrightarrow X_1 \dots X_k \quad | \quad X \in V \wedge X_1, \dots, X_k \in (V \cup \Sigma) \wedge k \geq 0.$$

Abbiamo modificato leggermente la forma ma il succo è quello: ad ogni variabile associamo una sequenza (anche vuota) di terminali e non terminali.

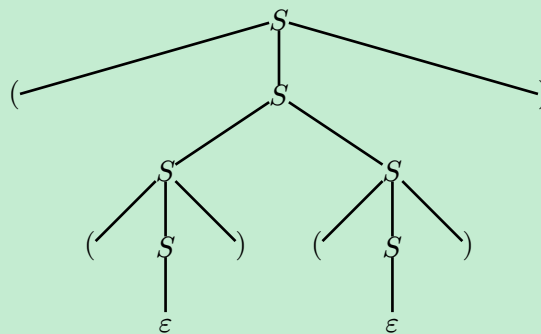
Il processo di derivazione nelle grammatiche di tipo 2 può essere espresso mediante **alberi di derivazione**: essi sono alberi che visualizzano l'applicazione delle regole di produzione.

Un nodo, con i suoi figli diretti, va a rappresentare l'**applicazione** di una regola di produzione. Ad esempio, prendendo la regola di produzione generica di una grammatica di tipo 2, abbiamo il seguente **albero di derivazione**:



Esempio 4.2.1.1: Riprendiamo la grammatica per le parentesi tonde bilanciate, che avevamo fatto a inizio corso, con le seguenti regole di produzione:

Vediamo due alberi di derivazione che abbiamo in questo linguaggio.



```

graph TD
    S1[S] --- S2[S]
    S1 --- S3[S]
    S1 --- S4[S]
    S2 --- S5[S]
    S2 --- S6[S]
    S3 --- S7[S]
    S3 --- S8[S]
    S4 --- S9[S]
    S5 --- S10[S]
    S5 --- S11[S]
    S6 --- S12[S]
    S6 --- S13[S]
    S7 --- S14[S]
    S7 --- S15[S]
    S8 --- S16[S]
    S8 --- S17[S]
    S9 --- S18[S]
    S9 --- S19[S]
    S10 --- S20[S]
    S10 --- S21[S]
    S11 --- S22[S]
    S11 --- S23[S]
    S12 --- S24[S]
    S12 --- S25[S]
    S13 --- S26[S]
    S13 --- S27[S]
    S14 --- S28[S]
    S14 --- S29[S]
    S15 --- S30[S]
    S15 --- S31[S]
    S16 --- S32[S]
    S16 --- S33[S]
    S17 --- S34[S]
    S17 --- S35[S]
    S18 --- S36[S]
    S18 --- S37[S]
    S19 --- S38[S]
    S19 --- S39[S]
    S20 --- S40[S]
    S20 --- S41[S]
    S21 --- S42[S]
    S21 --- S43[S]
    S22 --- S44[S]
    S22 --- S45[S]
    S23 --- S46[S]
    S23 --- S47[S]
    S24 --- S48[S]
    S24 --- S49[S]
    S25 --- S50[S]
    S25 --- S51[S]
    S26 --- S52[S]
    S26 --- S53[S]
    S27 --- S54[S]
    S27 --- S55[S]
    S28 --- S56[S]
    S28 --- S57[S]
    S29 --- S58[S]
    S29 --- S59[S]
    S30 --- S60[S]
    S30 --- S61[S]
    S31 --- S62[S]
    S31 --- S63[S]
    S32 --- S64[S]
    S32 --- S65[S]
    S33 --- S66[S]
    S33 --- S67[S]
    S34 --- S68[S]
    S34 --- S69[S]
    S35 --- S70[S]
    S35 --- S71[S]
    S36 --- S72[S]
    S36 --- S73[S]
    S37 --- S74[S]
    S37 --- S75[S]
    S38 --- S76[S]
    S38 --- S77[S]
    S39 --- S78[S]
    S39 --- S79[S]
    S40 --- S80[S]
    S40 --- S81[S]
    S41 --- S82[S]
    S41 --- S83[S]
    S42 --- S84[S]
    S42 --- S85[S]
    S43 --- S86[S]
    S43 --- S87[S]
    S44 --- S88[S]
    S44 --- S89[S]
    S45 --- S90[S]
    S45 --- S91[S]
    S46 --- S92[S]
    S46 --- S93[S]
    S47 --- S94[S]
    S47 --- S95[S]
    S48 --- S96[S]
    S48 --- S97[S]
    S49 --- S98[S]
    S49 --- S99[S]
    S50 --- S100[S]
    S50 --- S101[S]
    S51 --- S102[S]
    S51 --- S103[S]
    S52 --- S104[S]
    S52 --- S105[S]
    S53 --- S106[S]
    S53 --- S107[S]
    S54 --- S108[S]
    S54 --- S109[S]
    S55 --- S110[S]
    S55 --- S111[S]
    S56 --- S112[S]
    S56 --- S113[S]
    S57 --- S114[S]
    S57 --- S115[S]
    S58 --- S116[S]
    S58 --- S117[S]
    S59 --- S118[S]
    S59 --- S119[S]
    S60 --- S120[S]
    S60 --- S121[S]
    S61 --- S122[S]
    S61 --- S123[S]
    S62 --- S124[S]
    S62 --- S125[S]
    S63 --- S126[S]
    S63 --- S127[S]
    S64 --- S128[S]
    S64 --- S129[S]
    S65 --- S130[S]
    S65 --- S131[S]
    S66 --- S132[S]
    S66 --- S133[S]
    S67 --- S134[S]
    S67 --- S135[S]
    S68 --- S136[S]
    S68 --- S137[S]
    S69 --- S138[S]
    S69 --- S139[S]
    S70 --- S140[S]
    S70 --- S141[S]
    S71 --- S142[S]
    S71 --- S143[S]
    S72 --- S144[S]
    S72 --- S145[S]
    S73 --- S146[S]
    S73 --- S147[S]
    S74 --- S148[S]
    S74 --- S149[S]
    S75 --- S150[S]
    S75 --- S151[S]
    S76 --- S152[S]
    S76 --- S153[S]
    S77 --- S154[S]
    S77 --- S155[S]
    S78 --- S156[S]
    S78 --- S157[S]
    S79 --- S158[S]
    S79 --- S159[S]
    S80 --- S160[S]
    S80 --- S161[S]
    S81 --- S162[S]
    S81 --- S163[S]
    S82 --- S164[S]
    S82 --- S165[S]
    S83 --- S166[S]
    S83 --- S167[S]
    S84 --- S168[S]
    S84 --- S169[S]
    S85 --- S170[S]
    S85 --- S171[S]
    S86 --- S172[S]
    S86 --- S173[S]
    S87 --- S174[S]
    S87 --- S175[S]
    S88 --- S176[S]
    S88 --- S177[S]
    S89 --- S178[S]
    S89 --- S179[S]
    S90 --- S180[S]
    S90 --- S181[S]
    S91 --- S182[S]
    S91 --- S183[S]
    S92 --- S184[S]
    S92 --- S185[S]
    S93 --- S186[S]
    S93 --- S187[S]
    S94 --- S188[S]
    S94 --- S189[S]
    S95 --- S190[S]
    S95 --- S191[S]
    S96 --- S192[S]
    S96 --- S193[S]
    S97 --- S194[S]
    S97 --- S195[S]
    S98 --- S196[S]
    S98 --- S197[S]
    S99 --- S198[S]
    S99 --- S199[S]
    S100 --- S200[S]
    S100 --- S201[S]
    S101 --- S202[S]
    S101 --- S203[S]
    S102 --- S204[S]
    S102 --- S205[S]
    S103 --- S206[S]
    S103 --- S207[S]
    S104 --- S208[S]
    S104 --- S209[S]
    S105 --- S210[S]
    S105 --- S211[S]
    S106 --- S212[S]
    S106 --- S213[S]
    S107 --- S214[S]
    S107 --- S215[S]
    S108 --- S216[S]
    S108 --- S217[S]
    S109 --- S218[S]
    S109 --- S219[S]
    S110 --- S220[S]
    S110 --- S221[S]
    S111 --- S222[S]
    S111 --- S223[S]
    S112 --- S224[S]
    S112 --- S225[S]
    S113 --- S226[S]
    S113 --- S227[S]
    S114 --- S228[S]
    S114 --- S229[S]
    S115 --- S230[S]
    S115 --- S231[S]
    S116 --- S232[S]
    S116 --- S233[S]
    S117 --- S234[S]
    S117 --- S235[S]
    S118 --- S236[S]
    S118 --- S237[S]
    S119 --- S238[S]
    S119 --- S239[S]
    S120 --- S240[S]
    S120 --- S241[S]
    S121 --- S242[S]
    S121 --- S243[S]
    S122 --- S244[S]
    S122 --- S245[S]
    S123 --- S246[S]
    S123 --- S247[S]
    S124 --- S248[S]
    S124 --- S249[S]
    S125 --- S250[S]
    S125 --- S251[S]
    S126 --- S252[S]
    S126 --- S253[S]
    S127 --- S254[S]
    S127 --- S255[S]
    S128 --- S256[S]
    S128 --- S257[S]
    S129 --- S258[S]
    S129 --- S259[S]
    S130 --- S260[S]
    S130 --- S261[S]
    S131 --- S262[S]
    S131 --- S263[S]
    S132 --- S264[S]
    S132 --- S265[S]
    S133 --- S266[S]
    S133 --- S267[S]
    S134 --- S268[S]
    S134 --- S269[S]
    S135 --- S270[S]
    S135 --- S271[S]
    S136 --- S272[S]
    S136 --- S273[S]
    S137 --- S274[S]
    S137 --- S275[S]
    S138 --- S276[S]
    S138 --- S277[S]
    S139 --- S278[S]
    S139 --- S279[S]
    S140 --- S280[S]
    S140 --- S281[S]
    S141 --- S282[S]
    S141 --- S283[S]
    S142 --- S284[S]
    S142 --- S285[S]
    S143 --- S286[S]
    S143 --- S287[S]
    S144 --- S288[S]
    S144 --- S289[S]
    S145 --- S290[S]
    S145 --- S291[S]
    S146 --- S292[S]
    S146 --- S293[S]
    S147 --- S294[S]
    S147 --- S295[S]
    S148 --- S296[S]
    S148 --- S297[S]
    S149 --- S298[S]
    S149 --- S299[S]
    S150 --- S300[S]
    S150 --- S301[S]
    S151 --- S302[S]
    S151 --- S303[S]
    S152 --- S304[S]
    S152 --- S305[S]
    S153 --- S306[S]
    S153 --- S307[S]
    S154 --- S308[S]
    S154 --- S309[S]
    S155 --- S310[S]
    S155 --- S311[S]
    S156 --- S312[S]
    S156 --- S313[S]
    S157 --- S314[S]
    S157 --- S315[S]
    S158 --- S316[S]
    S158 --- S317[S]
    S159 --- S318[S]
    S159 --- S319[S]
    S160 --- S320[S]
    S160 --- S321[S]
    S161 --- S322[S]
    S161 --- S323[S]
    S162 --- S324[S]
    S162 --- S325[S]
    S163 --- S326[S]
    S163 --- S327[S]
    S164 --- S328[S]
    S164 --- S329[S]
    S165 --- S330[S]
    S165 --- S331[S]
    S166 --- S332
```

Per leggere la stringa che viene generata basta una **visita in profondità** dell'albero.

26

Esempio 4.2.1.2: Rimaniamo con il linguaggio dell'ultimo esempio, e prendiamo in considerazione l'ultima stringa $()()()$ che abbiamo derivato.

Proviamo a scrivere la derivazione usando proprio le regole di produzione che abbiamo a disposizione. Ci accorgiamo subito che abbiamo diversi modi di arrivare a quella stringa:

$$\begin{aligned} S &\Rightarrow SS \Rightarrow SSS \Rightarrow (S)SS \Rightarrow ()SS \Rightarrow ()(S)S \Rightarrow ()()S \Rightarrow ()()(S) \Rightarrow ()()() \\ S &\Rightarrow SS \Rightarrow SSS \Rightarrow S(S)S \Rightarrow S()S \Rightarrow S()(S) \Rightarrow (S)()(S) \Rightarrow ()()(S) \Rightarrow ()()() \\ S &\Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()SS \Rightarrow ()(S)S \Rightarrow ()()S \Rightarrow ()()(S) \Rightarrow ()()(). \end{aligned}$$

In blu viene indicata la variabile S che viene sostituita ad ogni passo.

Queste derivazioni generano la stessa stringa ma hanno alberi di derivazione diversi (non lo disegno, ma vale quanto scritto).

Cerchiamo di dare una corrispondenza tra derivazioni e alberi di derivazione. Andiamo ad utilizzare le **derivazioni leftmost**: esse sono derivazioni in cui, ad ogni passo, la variabile che andiamo a sostituire è quella più a sinistra nella forma sentenziale.

Esempio 4.2.1.3: Nelle tre derivazioni precedenti ci accorgiamo che la prima e la terza derivazione sono leftmost, mentre la seconda non lo è.

Abbiamo quindi creato una corrispondenza 1 : 1 tra derivazioni leftmost e alberi di derivazione. Da questo momento, parleremo di derivazioni leftmost riferendoci ad alberi di derivazione e viceversa.

Definizione 4.2.1.1 (Grammatica ambigua): Una grammatica G è **ambigua** se $\exists w \in L(G)$ che ammette due alberi di derivazione differenti, oppure, in maniera equivalente, se $\exists w \in L(G)$ che ammette due derivazioni leftmost diverse.

Esempio 4.2.1.4: La grammatica delle parentesi tonde bilanciate è **ambigua**, mentre la grammatica delle parole palindrome di lunghezza pari è **non ambigua**.

La pila è la struttura che ci permette di implementare la **ricorsione**. I linguaggi CFL hanno in più, rispetto ai regolari, l'accesso alle strutture ricorsive, e questo lo vediamo negli esempi che abbiamo fatto: l'esempio delle parentesi tonde bilanciate ha come flow

- inizio qualcosa (trovo una tonda aperta);
- vedo se ho ancora qualcosa di bilanciato;
- finisco quel qualcosa (trovo una tonda chiusa).

Tra l'altro, molto figo che **tutti i CFL** si possono ricondurre al **linguaggio delle parentesi bilanciate**, molto molto bello.

4.2.2. Da grammatica di tipo 2 ad automa a pila

Abbiamo a disposizione una grammatica G di tipo 2, detta anche **CFG**. Vogliamo costruire un automa a pila M che simuli il processo di derivazione tramite derivazioni leftmost.

Sia quindi

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

una CFG. Costruiamo

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \mathbb{Q})$$

un PDA che accetterà **per pila vuota** tale che:

- l'insieme degli stati Q contiene un solo stato, ovvero

$$Q = \{q\};$$

- lo stato iniziale, vista la presenza di un solo stato, è

$$q_0 = q;$$

- l'alfabeto di lavoro della pila Γ contiene tutto l'alfabeto della grammatica, ovvero l'insieme di tutti i simboli di G , ovvero

$$\Gamma = V \cup \Sigma;$$

- il simbolo iniziale della pila Z_0 è l'assioma della grammatica, ovvero

$$Z_0 = S;$$

useremo la pila per metterci sopra quello che vogliamo espandere mano a mano;

- infine, la **funzione di transizione** è **non deterministica** e ha due regole:
 - ogni volta che ho una variabile sulla cima della pila, con una epsilon mossa per sostituirla con il lato destro di una produzione, ovvero

$$\forall A \in V \quad \delta(q, \varepsilon, A) = \{(q, \alpha) \mid (A \rightarrow \alpha) \in P\};$$

- ogni volta che ho un simbolo terminale sulla cima della pila, andiamo a leggere dal nastro e verifichiamo che i due valori siano uguali, ovvero

$$\forall a \in \Sigma \quad \delta(q, a, a) = \{(q, \varepsilon)\}.$$

Lemma 4.2.2.1: Vale

$$L(G) = N(M).$$

Dimostriamo questo con un esempio.

Esempio 4.2.2.1: Definiamo la grammatica $G = (V, \Sigma, P, S)$ tale che:

$$V = \{S, T, U\}$$

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

$$S \rightarrow TU$$

$$T \rightarrow aTb \mid \varepsilon$$

$$U \rightarrow bUa \mid \varepsilon$$

Questa grammatica genera il linguaggio

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

Andiamo a scrivere un PDA M per questa grammatica. Partiamo con le regole del primo tipo:

$$\begin{aligned}\delta(q, \varepsilon, S) &= \{(q, TU)\} \\ \delta(q, \varepsilon, T) &= \{(q, aTb), (q, \varepsilon)\} \\ \delta(q, \varepsilon, U) &= \{(q, bUa), (q, \varepsilon)\}.\end{aligned}$$

E terminiamo con le regole del secondo tipo:

$$\begin{aligned}\delta(q, a, a) &= \{(q, \varepsilon)\} \\ \delta(q, b, b) &= \{(q, \varepsilon)\}.\end{aligned}$$

Simuliamo l'automa a pila che abbiamo costruito e il processo di derivazione con la stringa

$$w = abbbaa.$$

Ricordiamoci di usare una **derivazione leftmost**: questo è comodo perché i simboli che scriviamo più in alto sono quelli più a sinistra nella stringa aggiunta, e noi facciamo le sostituzioni proprio a partire da sinistra, quindi ottimo.

Nella colonna della **testina**, in blu indichiamo il carattere che la testina può leggere, mentre nella colonna della **derivazione**, sempre in blu indichiamo i caratteri che sono già stati verificati. Di quest'ultimo fatto ne parleremo meglio dopo.

Pila	Testina	Derivazione	Spiegazione della mossa
S	$abbbaa$	S	Configurazione iniziale
T U	$abbbaa$	TU	L'unica sostituzione che posso fare per S la faccio. Inoltre, non sposto la testina
a T b U	$abbbaa$	$aTbU$	Avendo a disposizione il non determinismo, sono fortunato e scelgo la derivazione corretta e, come prima, non sposto la testina
T b U	$abbbaa$	$aTbU$	Ora che ho un carattere sulla cima della pila, verifico che sono uguali (lo sono), sposto avanti la testina consumando il carattere della pila e quello del nastro
b U	$abbbaa$	abU	Avendo a disposizione il non determinismo, sono fortunato e scelgo la derivazione corretta e, come prima, non sposto la testina

U	$abbbaa$	abU	Ora che ho un carattere sulla cima della pila, verifico che sono uguali (lo sono), sposto avanti la testina consumando il carattere della pila e quello del nastro
b U a	$abbbaa$	$abbUa$	Avendo a disposizione il non determinismo, sono fortunato e scelgo la derivazione corretta e, come prima, non sposto la testina
U a	$abbbaa$	$abbUa$	Ora che ho un carattere sulla cima della pila, verifico che sono uguali (lo sono), sposto avanti la testina consumando il carattere della pila e quello del nastro
b U a a	$abbbaa$	$abbbUaa$	Avendo a disposizione il non determinismo, sono fortunato e scelgo la derivazione corretta e, come prima, non sposto la testina
U a a	$abbbaa$	$abbbUaa$	Ora che ho un carattere sulla cima della pila, verifico che sono uguali (lo sono), sposto avanti la testina consumando il carattere della pila e quello del nastro
a a	$abbbaa$	$abbbaa$	Avendo a disposizione il non determinismo, sono fortunato e scelgo la derivazione corretta e, come prima, non sposto la testina
a	$abbbaa$	$abbbaa$	Ora che ho un carattere sulla cima della pila, verifico che sono uguali (lo sono), sposto avanti la testina consumando il carattere della pila e quello del nastro
ε	$abbbaa\varepsilon$	$abbbaa$	Ora che ho un carattere sulla cima della pila, verifico che sono uguali (lo sono), sposto avanti la testina consumando il carattere della pila e quello del nastro

Riprendiamo quello detto prima: possiamo notare che, guardando la derivazione, dalla variabile più a sinistra in poi c'è esattamente quello che troviamo sulla pila nello stesso momento, mentre prima della variabile troviamo la parte dell'input su nastro che abbiamo già controllato.

Le mosse che noi abbiamo etichettato come non deterministiche sono le mosse che avvengono nei **parser**:

- quando facciamo una predizione, ovvero quando cerchiamo di indovinare l'espansione, stiamo facendo una mossa di tipo **predictor**;

- quando controlliamo la predizione fatta, ovvero quando controlliamo le lettere sul nastro e sulla pila, stiamo facendo una mossa di tipo **scanner**.

Siamo partiti quindi da una CFG e abbiamo costruito un PDA **equivalente** che accetta per pila vuota e con un solo stato, tanta tanta roba.

4.2.3. Automa a pila a grammatica di tipo 2

Questo la prossima volta, sium.

4.3. Forme normali per le grammatiche context-free

Abbiamo detto che le produzioni sono nella forma

$$A \longrightarrow \alpha \quad | \quad A \in V \wedge \alpha \in (V \cup \Sigma)^*$$

Possiamo dare diverse forme alle regole di produzione che abbiamo nelle grammatiche di tipo 2, ognuna delle quali ha alcuni punti di forza che possono essere comodi in altri contesti.

Queste forme sono dette **forme normali** e le più conosciute sono la **forma normale di Greibach** e la **forma normale di Chomsky**.

4.3.1. FN di Greibach

Nella **forma normale di Greibach**, spesso abbreviata con **FNG**, le produzioni sono nella forma

$$A \longrightarrow \sigma A_1 A_2 \dots A_k \quad | \quad \sigma \in \Sigma \wedge A_1, A_2, \dots, A_k \in V \wedge k \geq 0.$$

Data una grammatica G qualunque, si può sempre scrivere una grammatica in FN di Greibach per lo stesso linguaggio a meno della parola vuota: infatti, se in G abbiamo la parola vuota, nella sua trasformata non ce l'abbiamo e la dobbiamo aggiungere a mano. In poche parole vale

$$L(\text{FNG}) = L(G) / \{\varepsilon\}.$$

La trasformazione da Greibach ad automa a pila in alcuni casi **elimina il non determinismo**, però da fare è abbastanza pesante e non vedremo come fare.

Con la FN di Greibach possiamo costruire un PDA leggermente più semplice di prima.

Data la grammatica G in FN di Greibach, vogliamo costruire un PDA

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

che **accetta per pila vuota** definito da:

- **insieme degli stati** Q formato da un solo stato, ovvero

$$Q = \{q\};$$

- lo **stato iniziale**, vista la presenza di un solo stato, è

$$q_0 = q;$$

- l'**alfabeto di lavoro della pila** Γ contiene tutto solo le variabili della grammatica, questo perché faremo un accorgimento per cancellare i terminali senza metterli sulla pila, quindi

$$\Gamma = V;$$

- il **simbolo iniziale della pila** Z_0 è l'assioma della grammatica, ovvero

$$Z_0 = S;$$

useremo la pila per metterci sopra quello che vogliamo espandere mano a mano;

- la **funzione di transizione** non usa più ε -mosse e, soprattutto, non mette più nella pila i simboli terminali, ovvero

$$\delta(q, \sigma, A) = \{(q, A_1 \dots A_k) \mid (A \rightarrow \sigma A_1 \dots A_k) \in P\},$$

ovvero guardo tutte le regole che iniziano in A e hanno σ come primo carattere e tutto il resto della sostituzione lo andiamo a mettere nella pila. In questo modo, stiamo già consumando σ senza metterlo direttamente sulla pila e lo stiamo già controllando quindi. Ci avvanzeranno poi tutte le variabili che sono rimaste sulla pila.

Il simbolo in input mi **aiuta** nella scelta della produzione, e questo può **ridurre il non determinismo**, ma dipende strettamente dalla grammatica che si ha davanti.

Esempio 4.3.1.1: Modifichiamo leggermente la grammatica dell'ultimo esempio usando le seguenti regole di produzione:

$$S \rightarrow TU \qquad T \rightarrow aTb \mid ab \qquad U \rightarrow bUa \mid ba$$

In questo caso non abbiamo più ε quindi il linguaggio diventa

$$L = \{a^n b^{n+m} a^m \mid n, m \geq 1\}.$$

Rendiamola in forma normale di Greibach (grazie Pighizzini, io non sono capace). Per fare ciò le produzioni diventano nella forma:

$$\begin{aligned} S &\rightarrow aTBU \mid aBU & T &\rightarrow aB \mid aTB & U &\rightarrow bA \mid bUA \\ A &\rightarrow a & B &\rightarrow b \end{aligned}$$

Come prima, vediamo l'evoluzione della stringa

$$w = abbbaa.$$

Pila	Testina	Derivazione	Spiegazione della mossa
S	$abbbaa$	S	Configurazione iniziale
B U	$abbbaa$	aBU	Avendo a disposizione il non determinismo, sono fortunato e scelgo la derivazione corretta, aiutandomi anche con il carattere che c'era sulla testina
U	$abbbaa$	abU	Come prima
U A	$abbbaa$	$abbUA$	Come prima
A A	$abbbaa$	$abbAA$	Come prima

A	$abbbbaa$	$abbbbaA$	Come prima
ε	$abbbbaa\varepsilon$	$abbbbaa$	Come prima

Ho ancora del non determinismo, ma è molto **ridotto**. In alcuni casi riusciamo addirittura a **toglierlo completamente**, ma dipende molto dalla grammatica.

Abbiamo quindi trovato un PDA che accetta per pila vuota, utilizza un solo stato e non utilizza le ε -mosse. Tutto bello, ma rimane comunque non deterministico.

Questa FN di Greibach nella pratica è **poco usata**: fare il passaggio non è banale e questo potrebbe stravolgere la grammatica iniziale rendendola illeggibile.

5. Lezione 16 [30/04]

5.1. Fine dimostrazione

Per finire la dimostrazione che gli automi a pila sono equivalenti alle grammatiche di tipo 2, ci manca da vedere il passo da automa a grammatica. Per fare ciò, dobbiamo introdurre una **nuova forma normale** per gli automi a pila, per rendere i conti più facili.

Esempio 5.1.1: Definiamo l'alfabeto $\Sigma = \{ (,) \}$ e consideriamo l'alfabeto delle stringhe che rappresentano sequenze di parentesi tonde bilanciate. Come costruiamo un automa a pila per questo linguaggio?

Facilmente, ogni volta che trovo una parentesi aperta metto un simbolo sulla pila, mentre ogni volta che trovo una parentesi chiusa tolgo un simbolo dalla pila, se possibile. Se a fine input arrivo in una configurazione

$$(q, \varepsilon, Z_0)$$

vado ad accettare, visto che tutto quello che ho messo sulla pila l'ho tolto. Andiamo quindi ad accettare «per stati finali», e non per pila vuota, come dice Pighizzini.

Inoltre, la versione «per stati finali» è deterministica, mentre quella per pila vuota non lo è.

Quello che stiamo facendo, in ogni caso, è buttare sulla pila delle robe da controllare dopo: ogni volta che apro devo fare altri lavori e poi andare a chiudere.

5.1.1. Forma normale per gli automi a pila

Vediamo una **forma normale** più semplice. Diamo delle regole:

1. all'inizio la pila contiene solo Z_0 e lo usiamo per marcare il fondo della pila. Questo carattere non è mai rimosso e mai aggiunto;
2. l'input è accettato se si raggiunge una configurazione in cui:
 - tutto l'input è stato letto;
 - la pila contiene solo Z_0 ;
 - lo stato è finale.

In poche parole, tutto ciò che metto sulla pila sono attività che ho lasciato in sospeso, alla fine devo aver terminato tutto. È un po' un mix tra pila vuota (circa) e stati finali;

3. le mosse che facciamo sulla pila sono:
 - **push** di un simbolo (uno alla volta);
 - **pop** del simbolo in cima alla pila;
 - **pila invariata**.

Esempio 5.1.1.1: Avendo delle regole nella forma

$$\delta(q, \sigma, A) = \{ (p, \alpha) \}$$

allora:

- se $\alpha = \varepsilon$ stiamo eseguendo una pop;

- se $\alpha = A$ stiamo facendo una pila invariata;
- se $\alpha = \beta A$ stiamo facendo una pila invariata e $|\beta|$ push;
- se $\alpha = \beta A' \mid A \neq A'$ stiamo facendo una pop, una push e $|\beta|$ push.

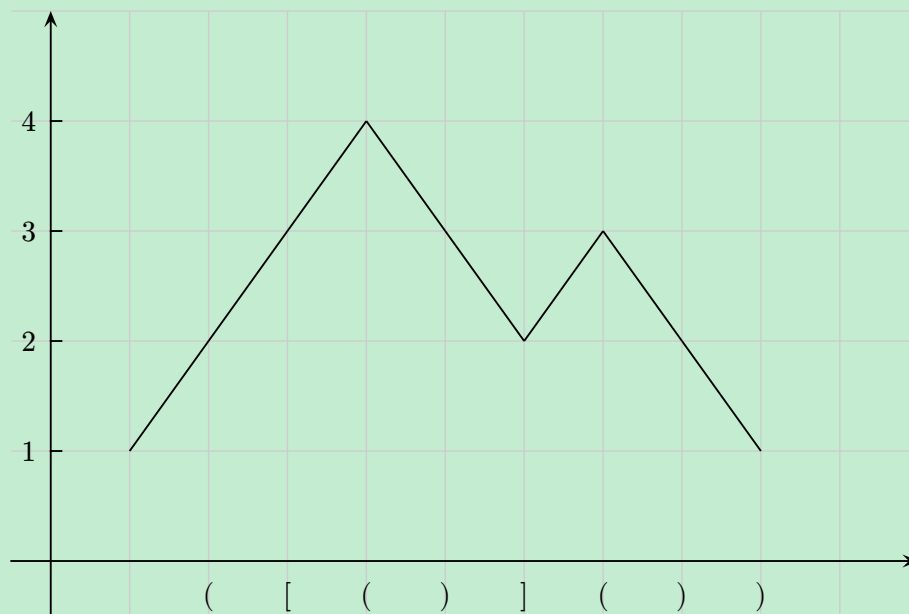
Ci rendiamo conto, visto l'esempio precedente, che **i due modelli sono equivalenti**.

Esempio 5.1.1.2: Definiamo l'alfabeto $\Sigma = \{ (,), [,] \}$. Vogliamo descrivere la computazione accettante della stringa $w = ([()])()$ data in input ad un automa a pila.

Simuliamo la computazione dell'automa su w visualizzando la pila dell'automa:

$$\begin{array}{cccccccc}
 & & & & A & & & \\
 & & B & B & B & & A & \\
 & A & A & A & A & A & A & A & . \\
 Z_0 & Z_0 & Z_0 & Z_0 & Z_0 & Z_0 & Z_0 & Z_0 & Z_0 \\
 \hline
 & (& [& (&) &] & (&) &)
 \end{array}$$

Andiamo a graficare anche l'altezza della pila che abbiamo ottenuto durante la computazione.



Sulle ascisse abbiamo rappresentato l'**input**, che possiamo vedere anche come **tempo** (a meno delle ε -mosse) nel quale l'automa si trova durante lo spostamento della testina da sinistra verso destra. Sulle ordinate invece abbiamo rappresentati l'**evoluzione** della pila.

Come vediamo, tra la prima tonda l'ultima tonda non andiamo mai sotto, idem quando apriamo e chiudiamo la quadra. Infatti, quando apro qualcosa, in mezzo non vado mai sotto e prima di tornare a livello devo riconoscere altre sequenze bilanciate.

In poche parole stiamo facendo una **chiamata ricorsiva**.

Studieremo le computazioni di questo tipo per scrivere la grammatica dall'automa a pila.

Aggiungiamo ancora una regola:

4. quando l'automa **legge** un simbolo di input e muove la testina la pila **non viene modificata**.

La nuova **funzione di transizione** di questi automi è nella forma

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow 2^{Q \times \{-, \text{push}(A) \mid A \in \Gamma, \text{pop}\}},$$

ovvero delle coppie formate da nuovo stato e operazione tra pila invariata, push e pop. Qua, a differenza di prima, non mi servono le parti finite perché ho già sottoinsiemi finiti.

Come sono fatte le regole della funzione di transizione? Dipende se stiamo leggendo o meno, per la nuova regola 4, quindi possiamo avere:

- **mosse di lettura**, che facciamo lasciando inalterata la pila per la regola che abbiamo appena aggiunto, ovvero

$$(p, -) \in \delta(q, a, A) \mid p, q \in Q \wedge a \in \Sigma, \wedge A \in \Gamma;$$

- **mosse pop**, che non leggono niente dal nastro ma liberano la prima posizione sulla pila, ovvero

$$(p, \text{pop}) \in \delta(q, \varepsilon, A);$$

- **mosse push**, che non leggono niente dal nastro ma aggiungono un elemento sulla pila, ovvero

$$(p, \text{push}(B)) \in \delta(q, \varepsilon, A) \mid B \in \Gamma;$$

- **mosse di cambio stato**, che non leggono niente e non modificano la pila, ovvero

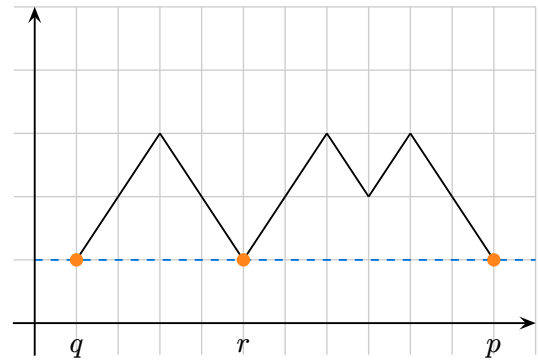
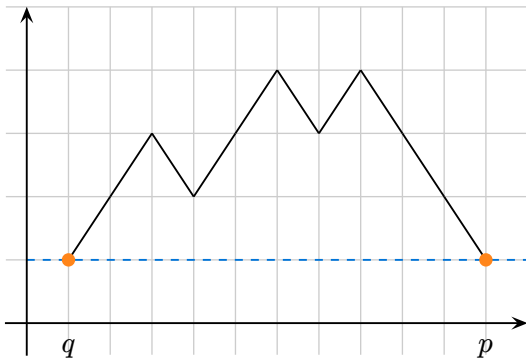
$$(p, -) \in \delta(q, \varepsilon, A).$$

5.1.2. Dimostrazione

Abbiamo detto che con questo nuovo automa noi vogliamo partire da Z_0 sulla pila e finire con lo stesso carattere alla fine della stringa. Abbiamo due casi possibili:

- durante la computazioni saliamo e scendiamo di livello ma raggiungiamo Z_0 alla fine;
- durante la computazione torniamo in Z_0 in almeno un punto.

In generale, possiamo sostituire a Z_0 un qualsiasi carattere che inseriamo sulla pila. Infatti, una volta inserito il carattere A sulla pila nello stato q , noi saliamo e scendiamo e poi usciamo con ancora A sulla pila nello stato p o nello stato r se siamo in uno intermedio.



Definiamo la grammatica $G = (V, \Sigma, P, S)$ formata dalle variabili

$$V = S \cup \{[qAp] \mid q, p \in Q \wedge A \in \Gamma\}.$$

Le variabili, oltre a S , sono delle **triple** che mi indicano lo stato nel quale sono, il simbolo corrente sulla pila e lo stato nel quale arrivo dopo essere tornato nel simbolo che ho trovato sulla pila.

Vediamo le **mosse** che possiamo fare.

Se stiamo **leggendo un simbolo in input**, non modifichiamo la pila, quindi

$$\forall(p, -) \in \delta(q, a, A) \quad [qAp] \rightarrow a.$$

In poche parole, ci stiamo spostando in linea retta, consumando il carattere a sul nastro:

$$q \boxed{A} \xrightarrow{a} \boxed{A} p.$$

Aggiungiamo anche la produzione

$$[qAq] \rightarrow \varepsilon$$

che serve per chiudere delle ricorsioni banali come se fosse un caso base. In poche parole, andiamo a generare solo la **parola vuota**:

$$q \boxed{A} \xrightarrow{\varepsilon} \boxed{A} q.$$

Se invece stiamo facendo la mossa che **cambia stato**, dobbiamo solo spostarci tra gli stati, ovvero

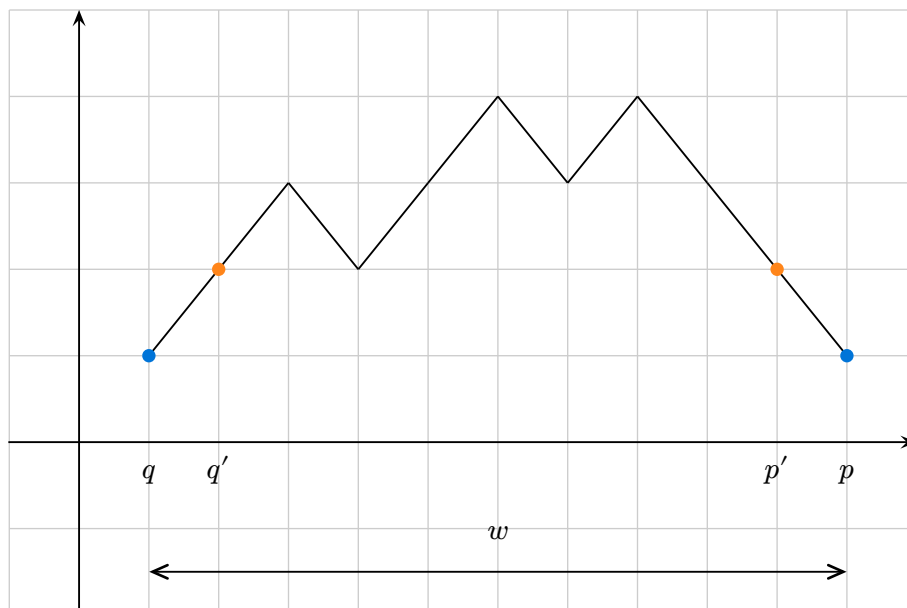
$$\forall(p, -) \in \delta(q, \varepsilon, A) \quad [qAp] \rightarrow \varepsilon.$$

Come prima, possiamo vedere graficamente il movimento come una linea retta:

$$q \boxed{A} \xrightarrow{\varepsilon} \boxed{A} p.$$

Ora facciamo le costruzioni **induttive**.

Supponiamo di essere nel caso in cui, dopo aver caricato A sulla pila, torniamo con la lettera A in cima alla pila alla fine della computazione. In questo caso, la pila è sempre più alta di A , che dopo essere stata caricata «induce» una chiamata ricorsiva nella quale carico, ad esempio, il carattere B e poi lo devo scaricare prima o poi.



Nel grafico, abbiamo B che viene indicato dal pallino arancione, che è posizionato sopra il pallino blu, che indica invece la lettera A che viene scaricata poi alla fine.

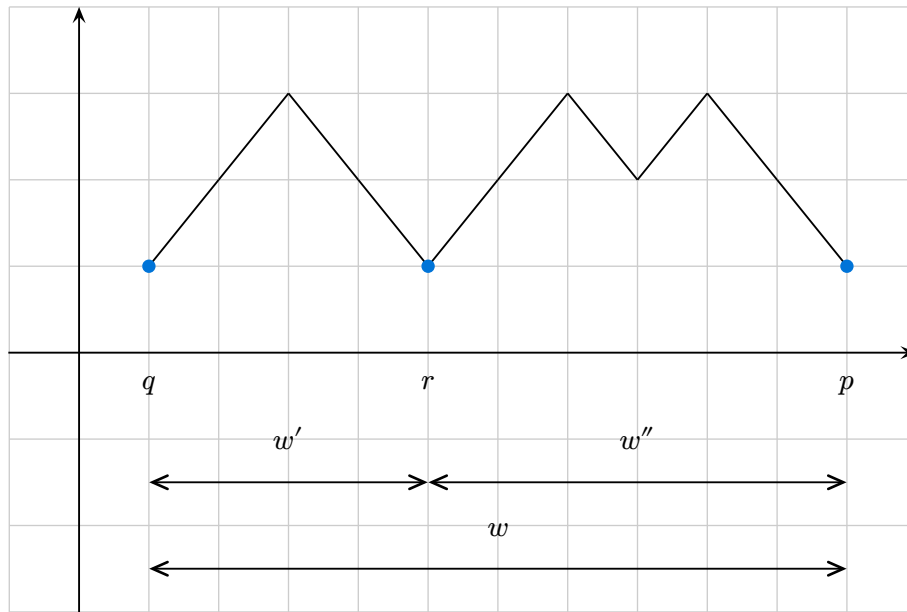
Se chiamiamo w la parte di stringa che stiamo riconoscendo tra la prima A e la seconda A , allora visto che non leggiamo altro durante la push di B stiamo andando a consumare ancora w ma partendo da B . Supponiamo che la mossa che carica la B mi manda nello stato q' e che la mossa che scarica la B parte da p' . Possiamo definire allora

$$\forall(q', \text{push}(B)) \in \delta(q, \varepsilon, A) \quad \forall(p, \text{pop}) \in \delta(p', \varepsilon, B) \quad [qAp] \longrightarrow [q'Bp'].$$

In poche parole, se da q ed A faccio una push di B in q' e poi da p' faccio una pop per andare in p ho fatto una chiamata ricorsiva che ora parte da B e usa gli stati accentati. Come detto prima, la stringa che stiamo consumando rimane sempre w perché con le push e le pop non leggiamo caratteri dal nastro.

L'ultimo caso che ci manca è quando abbiamo almeno una configurazione intermedia nella quale mi trovo allo stesso livello. In questo caso finiamo in uno stato r diverso, quindi abbiamo due percorsi:

- uno da q a r ;
- uno da r a p .



Come regole aggiungiamo

$$\forall r \in Q \quad [qAp] \longrightarrow [qAr][rAp].$$

In poche parole, spezziamo la computazione in due parti, entrambe che partono e finiscono in A .

Tra tutte le computazioni noi vogliamo quelle **accettanti**, quindi vorremmo arrivare in una configurazione del tipo

$$[q_0 Z_0 p] \mid p \in F.$$

Per far sì che ciò accada, dobbiamo imporre le regole

$$\forall p \in F \quad S \longrightarrow [q_0 Z_0 p].$$

Lemma 5.1.2.1: Vale

$$\forall q, p \in Q \quad \forall w \in \Sigma^* \quad [qAp] \xRightarrow{*} w \iff (q, w, A) \vdash^* (p, \varepsilon, A).$$

Lemma 5.1.2.2: Vale

$$S \xRightarrow{*} w \iff (q_0, w, Z_0) \vdash^* (p, \varepsilon, Z_0) \mid p \in F.$$

Hanno inoltre dimostrato che le triple per le variabili sono necessarie: non possiamo farne a meno.

Con questa **costruzione** abbiamo appena fatto vedere che i PDA sono **equivalenti** alle grammatiche di tipo 2.

5.2. Forma normale di Chomsky

L'altra volta abbiamo visto la FN di Greibach, oggi vediamo la **FN di Chomsky**, forma utile e maneggevole per alcune dimostrazioni che faremo.

5.2.1. Definizione

Le produzioni che abbiamo nella FN di Chomsky sono di due tipi:

$$A \longrightarrow a \mid BC \quad \text{tale che} \quad A, B, C \in V \wedge a \in \Sigma.$$

Questa rappresentazione è molto comoda perché riesce a generare degli alberi di derivazione che sono **binari**, quindi abbiamo molte indicazioni su numero di foglie, altezza, e altro.

Come nella FN di Greibach, anche qui non possiamo generare la parola vuota.

Infatti, se G è una grammatica di tipo 2, allora $\exists G'$ in FN di Chomsky quasi equivalente, ovvero

$$L(G') = L(G)/\{\varepsilon\}$$

ma solo se prima ce l'avevamo, sennò sono **totalmente equivalenti**.

5.2.2. Costruzione

Vediamo una **costruzione** per costruire effettivamente una grammatica in FN di Chomsky. Prima abbiamo detto che esiste, qua stiamo facendo vedere che esiste veramente.

Vogliamo costruire la grammatica $G' = (V, \Sigma, P', S)$ in FN di Chomsky a partire dalla grammatica G di tipo 2. Lo possiamo fare seguendo i seguenti passi:

FN di Chomsky

- 1: Eliminazione delle ε -produzioni
 - 2: Eliminazione delle produzioni unitarie
 - 3: Eliminazione dei simboli inutili
 - 4: Eliminazione dei terminali
 - 5: Smontaggio delle produzioni
-

I passi che abbiamo indicato devono essere eseguiti **in ordine**. Partiamo.

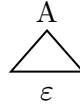
5.2.2.1. Eliminazione delle ε -produzioni

L'eliminazione delle ε -produzioni richiede la ricerca delle **variabili cancellabili**.

Definizione 5.2.2.1.1 (Variabile cancellabile): Una variabile A è **cancellabile** se

$$A \xRightarrow{*} \varepsilon.$$

L'albero di computazione di una variabile cancellabile A lo possiamo vedere come:



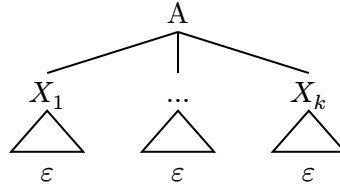
Banalmente, una variabile è cancellabile se nella grammatica ho una regola nella forma

$$A \rightarrow \varepsilon.$$

Se però non abbiamo questa produzione, ma abbiamo delle regole

$$A \rightarrow X_1 \dots X_k \quad \text{tale che} \quad X_1, \dots, X_k \in V$$

quello che possiamo fare è controllare queste variabili. Se sono tutte cancellabili, allora anche A sarà sicuramente cancellabile. Se vogliamo vedere l'albero di computazione, è nella forma:



Se invece non tutte sono cancellabili dobbiamo cercarle con un algoritmo simile a quello che abbiamo usato per dimostrare la decidibilità dei linguaggi di tipo 1. Definiamo l'insieme

$$\mathcal{C}_0 = \{A \in V \mid A \rightarrow \varepsilon\}$$

insieme di tutte le variabili banalmente cancellabili. Definiamo per induzione l'insieme

$$\mathcal{C}_i = \{A \in V \mid \exists (A \rightarrow X_1 \dots X_k) \in P \mid X_1, \dots, X_k \in \mathcal{C}_{i-1}\} \cup \mathcal{C}_{i-1}$$

formato da tutte le variabili che potremmo cancellare usando variabili già cancellabili.

Vale ovviamente la catena

$$\mathcal{C}_0 \subseteq \mathcal{C}_1 \subseteq \dots \subseteq V$$

che è bloccata da un insieme finito, quindi prima o poi non posso più aggiungere degli elementi all'insieme e mi devo fermare, ovvero

$$\exists i \mid \mathcal{C}_{i-1} = \mathcal{C}_i.$$

Una volta che ho tutte le variabili cancellabili creiamo delle **scorciatoie**: cancelliamo prima di tutto tutte le ε -produzioni, e poi

$$\forall (A \rightarrow Y_1 \dots Y_k) \in P \mid k > 0 \wedge Y_i \in V \cup \Sigma \quad (A \rightarrow Y_{j_1} \dots Y_{j_s}) \in P'.$$

In poche parole, aggiungiamo delle regole a P' che otteniamo eliminando alcune variabili cancellabili da una regola di produzione. L'eliminazione non deve toglierle per forza tutte: possiamo toglierne zero come toglierle tutte, ma dobbiamo ricordarci di non creare ε -produzioni e che vanno provate tutte le combinazioni possibili.

Esempio 5.2.2.1.1: Data la regola di produzione

$$A \rightarrow BCaD$$

con C, D variabili cancellabili, vogliamo costruire le nuove regole di una grammatica in FN di Chomsky eliminando le ε produzioni.

In questo caso possiamo:

- far sparire la C ;
- far sparire la D ;
- far sparire la C e la D ;
- lasciare tutto.

Le produzioni che otteniamo sono

$$A \rightarrow BCaD \mid BaD \mid BCa \mid Ba.$$

Vediamo un esempio un po' tedioso.

Esempio 5.2.2.1.2: Data la regola di produzione

$$A \rightarrow CDE$$

con C, D, E variabili cancellabili (quindi anche A), come ci comportiamo?

In questo caso dobbiamo cancellare tutti i possibili sottoinsiemi di variabili cancellabili, escluso l'insieme completo, quindi otteniamo le produzioni

$$A \rightarrow CDE \mid DE \mid CE \mid CD \mid E \mid D \mid C.$$

5.2.2.2. Eliminazione delle produzioni unitarie

L'**eliminazione delle produzioni unitarie** va a rimuovere le produzioni che ha destra hanno solo una variabile, ovvero sono nella forma

$$A \rightarrow B \quad \text{tale che} \quad A, B \in V.$$

Cerchiamo di cancellare le **sequenze di produzioni unitarie**: se consideriamo solo produzioni unitarie abbiamo una sequenza del tipo

$$A \Rightarrow B \Rightarrow C \Rightarrow \dots$$

che si può fermare quando deriviamo un terminale oppure una stringa con più di 2 stringhe. In poche parole abbiamo una sequenza

$$A \Rightarrow \dots \Rightarrow B \Rightarrow \alpha \mid \alpha \in \Sigma \vee |\alpha| > 1.$$

Quando trovo queste sequenze posso fare ancora una scorciatoia: al posto di fare tutta la sequenza di unitarie, che possiamo vedere solo come cambi di variabili, facciamo direttamente il salto da A ad α , ovvero

$$\forall(A, B) \mid A \xRightarrow{+} B \quad \forall(B \longrightarrow \alpha) \mid \alpha \in \Sigma \vee |\alpha| > 1 \quad (A \longrightarrow \alpha) \in P'.$$

5.2.2.3. Eliminazione dei simboli inutili

L'**eliminazione dei simboli inutili** rimuove tutti i simbolo che non ci servono a niente, che non sono utili a generare delle stringhe del linguaggio.

Definizione 5.2.2.3.1 (Simbolo utile): Un simbolo $X \in V \cup \Sigma$ è **utile** se:

- il simbolo è raggiungibile, quindi esiste una computazione che genera una forma sentenziale che contiene quel simbolo, ovvero

$$\exists S \xRightarrow{*} \alpha X \beta;$$

- dalla forma sentenziale centrale si riesce a generare una stringa del linguaggio, ovvero

$$\alpha X \beta \xRightarrow{*} w \in \Sigma^*.$$

Riassunto in una sola sentenza, possiamo dire che

$$\exists S \xRightarrow{*} \alpha X \beta \xRightarrow{*} w \in \Sigma.$$

Con questi tre passi abbiamo rimosso tutte le ε -produzioni, tutti i cammini unitari e tutti i simboli inutili. Le produzioni che abbiamo ora sono nella forma

$$A \longrightarrow \alpha$$

con α terminale oppure $|\alpha| > 1$.

5.2.2.4. Eliminazione dei terminali

L'**eliminazione dei terminali** si applica alle produzioni che derivano una forma sentenziale α con $|\alpha| > 1$. Per fare ciò dobbiamo usare delle **variabili ausiliarie** per rimuovere i simboli terminali nelle produzioni.

Come convenzione possiamo dire che i terminali $\sigma \in \Sigma$ vengono sostituiti dalle variabili X_σ .

Esempio 5.2.2.4.1: Date le regole di produzione

$$A \longrightarrow AaabcC \mid bC \mid bb$$

cerchiamo di applicare l'eliminazione dei terminali.

Usiamo due variabili ausiliarie X_a e X_b , una per ogni terminale, ottenendo

$$A \longrightarrow AX_aX_aX_bC \mid X_bC \mid X_bX_b$$

$$X_a \longrightarrow a$$

$$X_b \longrightarrow b.$$

Se avessi avuto anche la regola

$$C \rightarrow b$$

non avrei applicato il cambio in X_b perché avremmo ottenuto un cammino unitario, che però non possiamo avere in questo momento.

In questo penultimo passo ora abbiamo solo produzioni che derivano terminali oppure delle liste di variabili. L'ultimo passo sarà manipolare queste per raggiungere, finalmente, la FN di Chomsky.

5.2.2.5. Smontaggio delle produzioni

L'ultimo passo è lo **smontaggio delle produzioni** nella forma

$$A \rightarrow B_1 \dots B_k \quad \text{con } k > 2.$$

Infatti, le produzioni:

- con una sola variabile non ci sono;
- con due variabili vanno bene in questa FN;
- con tre o più variabili vanno ridotte.

Per fare ciò, usiamo ancora delle **variabili ausiliarie** per aggiungere delle regole formate da una variabile della produzione e da una variabile ausiliaria. Fa eccezione l'ultima regola, che sostituisce direttamente gli ultimi due caratteri della produzione da cambiare.

Esempio 5.2.2.5.1: Data la produzione

$$A \rightarrow B_1 B_2 B_3 B_4 B_5$$

la andiamo a smontare creando le produzioni

$$\begin{aligned} A &\rightarrow B_1 Z_1 \\ Z_1 &\rightarrow B_2 Z_2 \\ Z_2 &\rightarrow B_3 Z_3 \\ Z_3 &\rightarrow B_4 B_5. \end{aligned}$$

5.2.3. Esempio

Applichiamo questa costruzione ad un esempio.

Esempio 5.2.3.1: Costruire la FN di Chomsky a partire da queste produzioni:

$$\begin{aligned} S &\rightarrow aB \mid bA \\ A &\rightarrow a \mid aS \mid bAA \\ B &\rightarrow b \mid bS \mid aBB. \end{aligned}$$

Non abbiamo ε -produzioni e variabili cancellabili, come non abbiamo cammini unitari, come non abbiamo simboli inutili, quindi i primi tre passi non vengono eseguiti.

Partiamo quindi con il cancellare tutti i terminali:

$$\begin{aligned}S &\longrightarrow X_a B \mid X_b A \\A &\longrightarrow a \mid X_a S \mid X_b AA \\B &\longrightarrow b \mid X_b S \mid X_a BB \\X_a &\longrightarrow a \\X_b &\longrightarrow b.\end{aligned}$$

Infine, smontiamo le catene di variabili che abbiamo ottenuto:

$$\begin{aligned}S &\longrightarrow X_a B \mid X_b A \\A &\longrightarrow a \mid X_a S \mid X_b Y_1 \\Y_1 &\longrightarrow AA \\B &\longrightarrow b \mid X_b S \mid X_a Y_2 \\Y_2 &\longrightarrow BB \\X_a &\longrightarrow a \\X_b &\longrightarrow b.\end{aligned}$$

6. Lezione 17 [07/05]

Prima di iniziare con il topic di questa lezione facciamo qualche **esempio**.

Esempio 6.1: Definiamo il linguaggio

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

Riusciamo a dire che L è un linguaggio CF? **SI**, riusciamo a costruire una grammatica di tipo 2 o un automa a pila per questo linguaggio. Quest'ultimo è molto facile: carichiamole a , scarichiamo le b , scorriamo le c .

Esempio 6.2: Definiamo ora il linguaggio

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

Riusciamo ancora a costruire un automa a pila? **NO**, con un automa a pila riusciamo a caricare le a , confrontare le b ma facendo questo andiamo a distruggere l'informazione sul numero di n quindi non abbiamo indicazioni sulle c .

Abbiamo un modo formale per dimostrare che l'ultimo linguaggio non è CF?

6.1. Prerequisiti per il pumping lemma

Come nei linguaggi regolari, anche nei CFL abbiamo un **pumping lemma**. Questa è una **condizione necessaria** affinché un linguaggio sia CF, quindi questo lemma è usato come «arma» per dimostrare che un linguaggio non appartiene ai CFL.

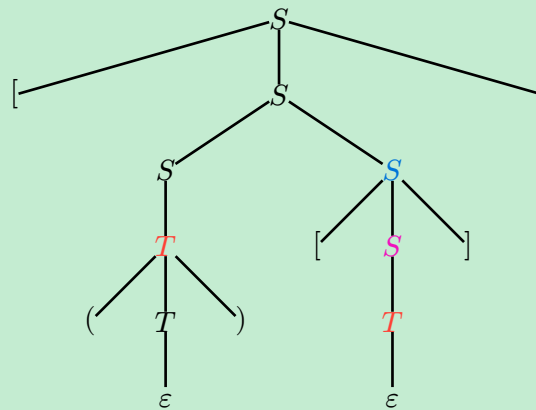
Esempio 6.1.1: Le regole di produzione

$$S \longrightarrow [S] \mid SS \mid T$$

$$T \longrightarrow (T) \mid TT \mid \varepsilon$$

sono in grado di generare le stringhe di parentesi bilanciate dove le parentesi tonde stanno solo dentro le parentesi quadre. La variabile S genera le quadre e dà un livello «esterno», mentre la variabile T genera le tonde e dà un livello «interno».

Vediamo un albero di derivazione per una stringa di questo linguaggio.

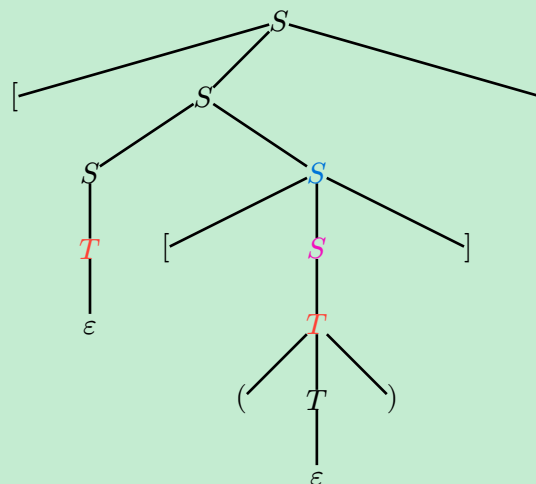


Questo è l'albero di derivazione della stringa

$$S \xRightarrow{*} [()[]].$$

Facciamo un paio di **osservazioni**.

Prendiamo i due alberi che hanno radice T colorata in rosso. Dal primo albero generiamo la stringa $()$ mentre dal secondo albero generiamo ε . Visto che questi due alberi hanno come radice la stessa variabile, possiamo **invertirli**.



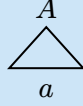
Con questa operazione otteniamo la stringa

$$S \xRightarrow{*} [[()]].$$

Prendiamo ora l'albero con radice in S colorato di blu. Questo albero ha radice in S che genera, in una sola mossa, la stringa $[S]$, dove la S tra quadre è rappresentata dalla S fucsia nel disegno. Notiamo che possiamo innestare questo albero in sé stesso un numero arbitrario di volte.

[Passo base: $k = 1$]

Visto che devo derivare almeno un terminale nella stringa, devo partire la derivazione usando la regola di produzione $A \rightarrow a$. Gli alberi con profondità 1 sono nella forma

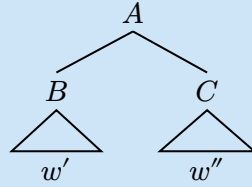


Avendo questa profondità, e generando quindi solo un terminale, la lunghezza delle stringhe è

$$|w| = |a| = 1 \leq 2^{k-1} \stackrel{k=1}{=} 2^0 = 1.$$

[Passo induttivo: $k - 1 \rightarrow k$]

Supponiamo di avere alberi di profondità $k > 1$, ovvero alberi che nella radice hanno usato la seconda regola di produzione $A \rightarrow BC$.



La stringa w la possiamo vedere come la concatenazione delle due stringhe generate dai due sotto-alberi, quindi

$$w = w'w''.$$

La profondità di questo albero è k , quindi gli alberi in B e C sono di profondità al massimo $k - 1$. Questi alberi sono

$$T' : B \xRightarrow{*} w' \quad | \quad T'' : C \xRightarrow{*} w''$$

che per ipotesi di induzione generano stringhe al massimo lunghe $2^{k-1-1} = 2^{k-2}$. Visto che w è la concatenazione delle due stringhe, possiamo dire che

$$|w| = \underbrace{|w'|}_{\leq 2^{k-2}} + \underbrace{|w''|}_{\leq 2^{k-2}} \leq 2^{k-1}.$$

■

6.2. Pumping lemma per i CFL

Nel pumping lemma per i linguaggi regolari prendevamo delle stringhe lunghe almeno quanto il numero di stati e osservavamo che si ripeteva almeno uno stato nella computazione.

Nel **pumping lemma per i CFL** non ci muoviamo più in linea, ma ci muoviamo all'interno dell'albero di derivazione cercando una variabile che viene ripetuta.

Vediamo la definizione formale e la dimostrazione.

Lemma 6.2.1 (Pumping lemma per i CFL): Sia L un linguaggio CF. Allora $\exists N > 0$ tale che $\forall z \in L$ tale che $|z| \geq N$ essa può essere scritta come $z = uvwxy$ tale che:

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

Se nel **PL3** ripetevamo la parte centrale dicendo che questa non poteva essere vuota, ora invece:

1. la parte centrale della decomposizione è lunga al massimo N ;
2. la seconda e la quarta parte non sono entrambe vuote allo stesso momento;
3. visto che almeno una parte tra la seconda e la quarta è non vuota, possiamo pompare quelle due parti lo stesso numero di volte generando nuove stringhe che mi fanno rimanere comunque dentro L .

Dimostrazione 6.2.1.1: Abbiamo a disposizione un linguaggio CF, dal quale possiamo ricavare facilmente una grammatica CF e, per costruzione, una grammatica in FN di Chomsky. Sia quindi $G = (V, \Sigma, P, S)$ una grammatica in FN di Chomsky per $L/\{\varepsilon\}$.

Fissato $k = |V|$, definiamo

$$N = 2^k.$$

Prendiamo delle stringhe $z \in L$ tali che $|z| \geq N$. Se $z \in L$ allora esiste un albero di derivazione che, partendo da S , mi genera la stringa z :

$$T : S \xRightarrow{*} z.$$

Questo albero, molto piccolino, è nella forma:



Abbiamo definito la stringa z in modo che $|z| \geq N = 2^k$. Grazie al lemma precedente sappiamo che la lunghezza di una stringa è limitata dalla profondità del suo albero di derivazione: se h è la profondità dell'albero T , allora sappiamo che

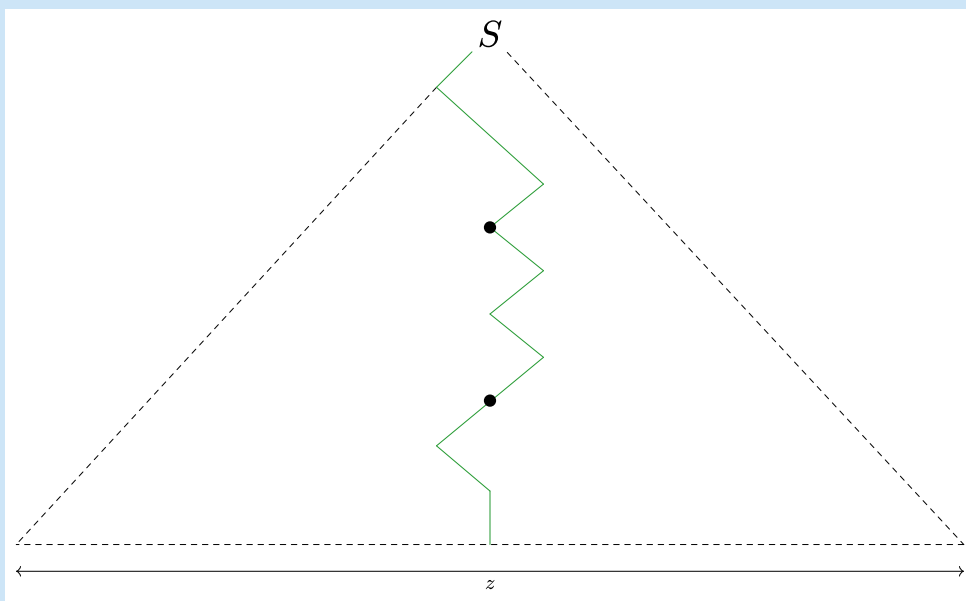
$$|z| \leq 2^{h-1}.$$

Unendo le due disuguaglianze otteniamo che

$$2^{h-1} \geq |z| \geq 2^k \implies h - 1 \geq k \implies h \geq k + 1.$$

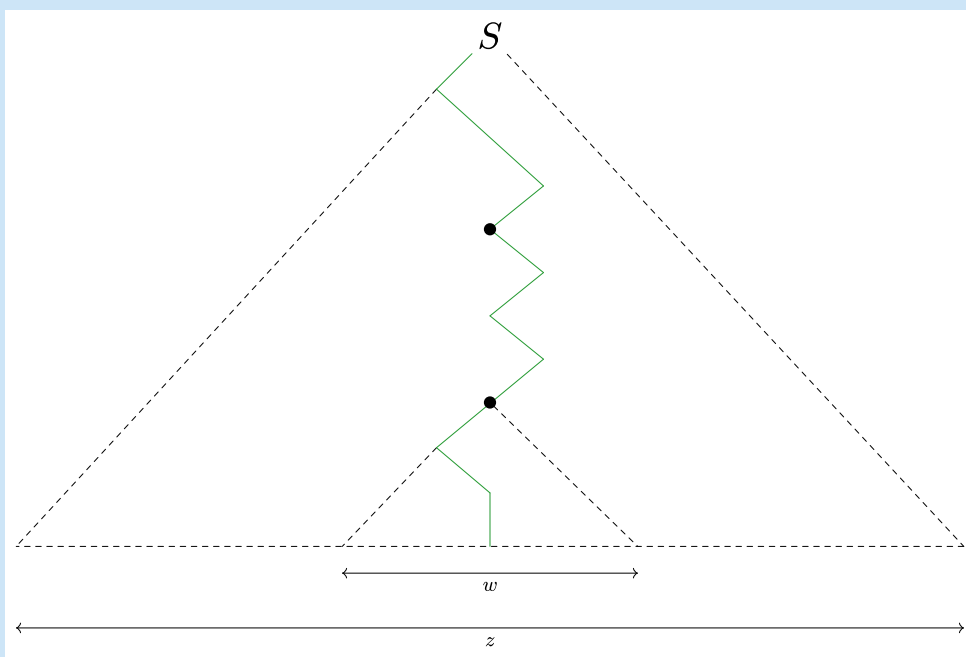
La profondità di un albero è il cammino massimo dalla radice alla foglia più lontana. Visto che questa profondità è almeno $k + 1$, stiamo attraversando $k + 1$ archi e quindi $k + 2$ nodi.

Di questi $k + 2$ nodi, l'ultimo che visitiamo è il terminale presente nella stringa z , quindi stiamo visitando $k + 1$ variabili. Avendo a disposizione k variabili, vuol dire che visitiamo una variabile almeno due volte. Sia A questa variabile che viene ripetuta.

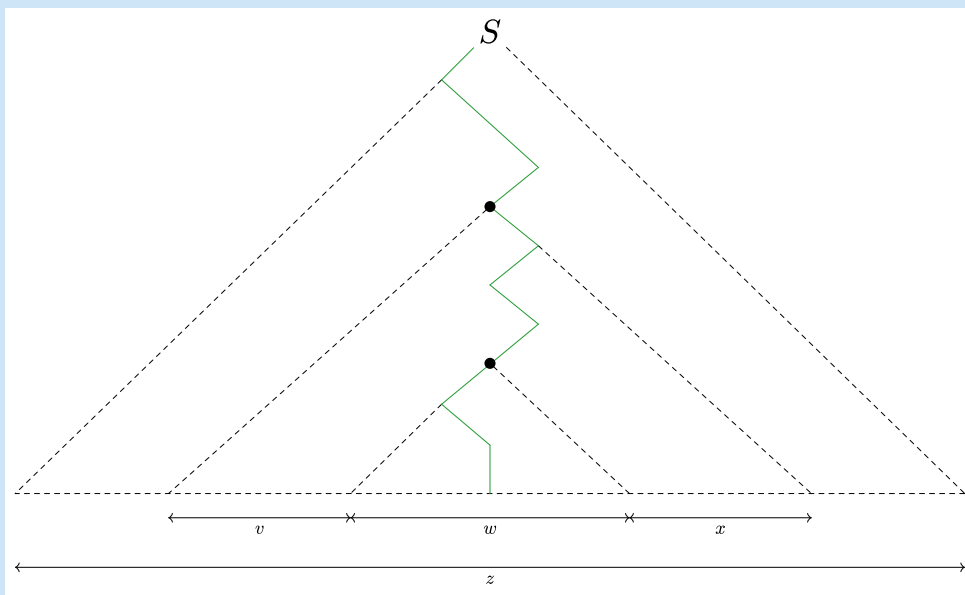


Nella figura precedente abbiamo indicato con due pallini la variabile A che viene ripetuta durante il cammino dal fondo verso la radice. Ora iniziamo la divisione in fattori.

Consideriamo solo l'albero che parto dalla A più sotto: esso genera un fattore di z , che chiamiamo w , ovvero $A \stackrel{*}{\Rightarrow} w$.

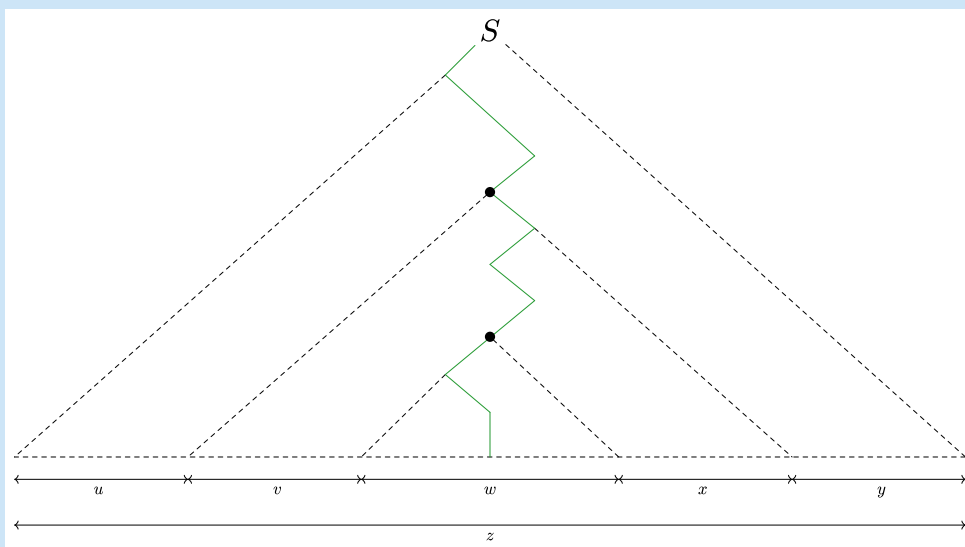


Consideriamo ora l'albero che parte dalla A più sopra: esso genera un altro fattore di z , che contiene quello precedente più due fattori esterni, che chiamiamo v e x , ovvero $A \stackrel{*}{\Rightarrow} vwx$.



Infine, prendiamo i due fattori esterni, che chiamiamo u e y , trovando quindi la derivazione completa di z come

$$S \Rightarrow^* uvwxy.$$



Abbiamo quindi mostrato che esiste la decomposizione.

[TERZO PUNTO]

Cosa osserviamo dal disegno?

Prendiamo la parte esterna dell'albero, ovvero la stringa generata dalla S alla prima A . La produzione che sta avvenendo ora è

$$S \Rightarrow^* uAy.$$

Prendiamo ora la parte intermedia dell'albero, ovvero la stringa generata tra le due A . La produzione che sta avvenendo è

$$A \xRightarrow{*} vAx.$$

Infine, prendiamo la parte interna dell'albero, ovvero la stringa generata dalla seconda A . La produzione che sta avvenendo è

$$A \xRightarrow{*} w.$$

Facciamo il gioco che abbiamo fatto prima con le parentesi: prendiamo l'albero intermedio, lo innestiamo tante volte in sé stesso e poi mettiamo l'albero interno come tappo, ovvero

$$S \xRightarrow{*} uAy \Rightarrow uvAxy \xRightarrow{*} uvvAxxxy \xRightarrow{*} \dots \xRightarrow{*} uv^i Ax^i y \xRightarrow{*} uv^i wx^i y.$$

Questo possiamo farlo un numero arbitrario di volte, anche 0: infatti, con $i = 0$ è come mettere subito il tappo al posto di vAx .

[SECONDO PUNTO]

Le due parti vuote non sono entrambe vuote allo stesso tempo: noi all'inizio dell'albero di derivazione stiamo applicando una regola del tipo $A \rightarrow BC$. Supponiamo che in B ci sia la ripetizione della variabile A e che da qui esca il fattore v . Nell'albero che invece parte da C esce il fattore x , che però non può essere vuoto perché non abbiamo, in una FN di Chomsky, delle ε -produzioni.

[PRIMO PUNTO]

Consideriamo il cammino di z che parte dalla foglia e arriva fino a S . Se saliamo di $k + 1$ archi abbiamo attraversato $k + 2$ nodi, ma uno di questi è il carattere terminale della foglia, quindi abbiamo attraversato $k + 1$ variabili. Avendo a disposizione k variabili, una viene ripetuta.

Questo albero ha altezza massima $k + 1$, perché al massimo in quel punto otteniamo la ripetizione della variabile. La variabile che troviamo ripetuta fa partire un albero che genera la stringa vwx , ma per il lemma precedente vale

$$|vwx| \leq 2^{k+1-1} = 2^k = N. \quad \blacksquare$$

6.3. Applicazioni del pumping lemma

Andiamo ad **applicare il pumping lemma** al linguaggio di prima.

Esempio 6.3.1: Sia quindi

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

Come nel PL3, dobbiamo assumere per assurdo che L sia CFL e vedere quale delle tre proprietà va a decadere per via di questa assunzione.

Sia per assurdo L un CFL, allora $\exists N$ costante del PL per L . Cerchiamo una stringa che fa cadere una delle tre proprietà: non stiamo a risparmiare, abbondiamo, prendiamo la stringa

$$z = a^N b^N c^N.$$

Sappiamo che la possiamo decomporre come $z = uvwxy$.

Sfruttiamo la condizione 1: la parte centrale è al massimo N , quindi se contiene una a non contiene una c , viceversa se contiene una c non contiene una a . Quindi abbiamo

$$\#_a(vwx) = 0 \vee \#_c(vwx) = 0.$$

Prendiamo il caso in cui $\#_a(vwx) = 0$, l'altro è totalmente simmetrico.

Sappiamo che possiamo ripetere in modo arbitrario i due fattori pompabili, quindi se scegliamo $i = 0$ allora otteniamo

$$z' = uwy \in L.$$

Contiamo il numero di terminali che abbiamo:

- $\#_a(z') = \#_a(z) = N$ perché non le avevo nella parte cancellata;
- $\#_b(z') = \#_b(z) - \#_b(vx) = N - \#_b(vx)$ per la parte cancellata;
- $\#_c(z') = \#_c(z) - \#_c(vx) = N - \#_c(vx)$ per la parte cancellata.

Ora usiamo la seconda condizione, quindi sapendo che $vx \neq \varepsilon$ questo implica che

$$\#_b(vx) + \#_c(vx) > 0$$

perché avendo almeno una lettera in vx , e non avendo a , ho cancellato almeno una b o una c , quindi

$$\#_b(z') < N \vee \#_c(z') < N.$$

Ma questo è assurdo: z' dovrebbe essere in L ma non lo è.

Vediamo un altro **linguaggio a blocchi**.

Esempio 6.3.2: Definiamo ora

$$L = \{a^h b^j c^k \mid j = \max(h, k)\}.$$

Questo linguaggio non è CFL: infatti, in maniera non deterministica possiamo scommettere all'inizio di avere a e b uguali o b e c uguali, ma non possiamo controllare che il numero di b sia massimo. Infatti, potrei avere a e b uguali, e questo la pila lo sa fare, ma un numero di c superiore. Se la richiesta fosse

$$j = h \vee j = k$$

potremmo scrivere un NPDA per il linguaggio, ma qua è diverso.

Per assurdo sia L un CFL, e sia quindi N la costante del PL per L . Prendiamo una stringa

$$z = a^N b^N c^N$$

che ovviamente sta nel linguaggio ed è lunga almeno N . Decomponiamola quindi in $z = uvwxy$ e, sapendo che $|vwx| \leq N$, sappiamo che

$$vwx \in a^* b^* \vee vwx \in b^* c^*.$$

Sappiamo inoltre che $vw \neq \varepsilon$, quindi qui abbiamo almeno un carattere.

Andiamo per **casi**:

- se $vw \in a^+$ allora pompriamo la stringa con $i = 2$ e otteniamo una stringa con un numero di a più grande del numero di b e di c , ma b deve essere uguale al massimo, quindi questo è un assurdo;
- se $vw \in c^+$ allora pompriamo nello stesso modo con $i = 2$;
- se $vw \in b^+$ allora togliamo un po' di b con $i = 0$ per renderle in numero minore del numero massimo, che è N , quindi questo è un assurdo;
- se $vw \in a^+b^+$ dobbiamo andare per **casi** per capire dove avviene la divisione tra a e b :
 - ▶ se $v \in a^+b^+$ allora la divisione avviene nel primo fattore; se pompriamo con $i = 2$ otteniamo la stringa $uvvwxy$ formata da una serie di a , da una serie di b , poi ancora una serie di a , ma questa operazione ci fa perdere la struttura del linguaggio, quindi questo è un assurdo;
 - ▶ se $x \in a^+b^+$ allora la divisione avviene nel terzo fattore; se pompriamo con $i = 2$ otteniamo la stessa perdita di struttura di prima;
 - ▶ se $v = a^l \wedge x = b^r$ allora la divisione è nel fattore centrale; visto che questi due fattori assieme non sono vuoti, vuol dire che $l + r > 0$; se andiamo a prendere la stringa uv^iwx^iy noi stiamo aggiungendo $i - 1$ volte un numero l di a e un numero r di b , ovvero

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

Andiamo ancora per **casi**:

- se $l \neq r$ allora per $i = 2$ il numero di a e b sono diverse ma quelle di b non sono uguali al massimo, che può essere il numero di a o il numero di b ;
- se $l = r$ prendiamo $i = 0$ così che le a e le b siano uguali ma siano in numero minore di N , che però è il massimo.
- se $vw \in b^+c^+$ facciamo il simmetrico del caso precedente.

In ogni caso possibile abbiamo ottenuto un assurdo, quindi L non è CFL.

Questi esempi sono facili perché hanno la struttura a blocchi. Vediamo un esempio che è riconducibile ad una struttura a blocchi.

Esempio 6.3.3: Definiamo ora

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

Non riusciamo a scrivere un automa a pila per L : possiamo mettere la stringa α sulla pila, anche in maniera deterministica aggiungendo un separatore, ma per andare poi a confrontare il primo carattere sulla stringa dobbiamo distruggere tutta l'informazione.

Sia per assurdo L un CFL e sia N la costante del PL. Prendiamo $z \in L$ tale che $|z| \geq N$ nella forma $z = \alpha\alpha$. Come facciamo a costruire una roba del genere? Cerchiamo di ritornare in una **struttura a blocchi**, ovvero prendiamo la stringa

$$z = \underbrace{a^N b^N}_{\alpha} \underbrace{a^N b^N}_{\alpha}.$$

Non lo facciamo vedere, ma la dimostrazione è molto simile a quella precedente.

Infine, vediamo un esempio di una struttura non a blocchi.

Esempio 6.3.4: Definiamo infine

$$L = \{a^p \mid p \text{ è un numero primo}\}.$$

Non riusciamo a scrivere un automa a pila perché per sapere se p è primo l'automa dovrebbe saper fare delle divisioni, che infatti non sa fare.

Per assurdo sia quindi L un CFL. Sia N la costante del PL per L e definiamo un numero primo m tale che $m \geq N$ che usiamo per definire

$$z = a^m.$$

Decomponiamo la stringa come $z = uvwxy$. Sappiamo che $vx \neq \varepsilon$ quindi

$$|vx| = k > 0.$$

La stringa uv^iwx^iy è ottenuta da z aggiungendo i fattori v e x per $i - 1$ volte, ovvero

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

Sappiamo inoltre che

$$|vwx| \leq N \implies |vx| \leq N \rightsquigarrow k \leq N \leq m.$$

Scegliamo $i = m + 1$: allora otteniamo la stringa

$$a^{m+(m+1-1)k} = a^{m+mk} = a^{m(k+1)}$$

che chiaramente non è lunga quanto un numero primo perché è fattorizzato.

Ma questo è un assurdo, quindi L non è CFL.

Una cosa molto interessante è che questo linguaggio lo potevamo dimostrare anche con il pumping lemma per i linguaggi regolari: infatti, nei linguaggi unari i CFL sono uguali ai linguaggi regolari.

7. Lezione 18 [09/05]

7.1. Ancora pumping lemma

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

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

Qua abbiamo due casi:

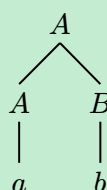
- se $P = A$ allora abbiamo subito la ripetizione e potremmo avere v o x uguali ad ε ma non tutti e due, perché da B tiriamo fuori almeno un terminale, non avendo ε -produzioni;
- se $P \neq A$ allora ancora meglio di prima perché tutti e due potrebbero non essere nulli, visto che ci biforchiamo ancora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

Vediamo un esempio per capire meglio.

Esempio 7.1.1: Abbiamo una grammatica in FN di Chomsky con le regole di produzione

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

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



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

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

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

7.2. Fail del pumping lemma

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

Esempio 7.2.1: Definiamo il linguaggio

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

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

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

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

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

Abbiamo diversi casi da controllare:

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

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

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

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

$[vwx \in c^+]$

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

Infatti, questo dipende dalla z che abbiamo a disposizione:

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

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

ma allora scegliendo

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

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

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

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

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

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

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

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

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

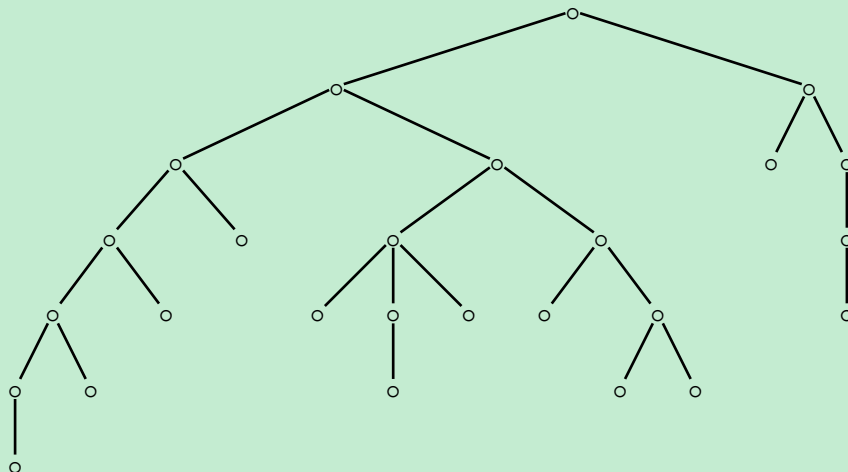
Quindi anche in questo caso ci sono casi favorevoli ma non tutti lo sono.

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

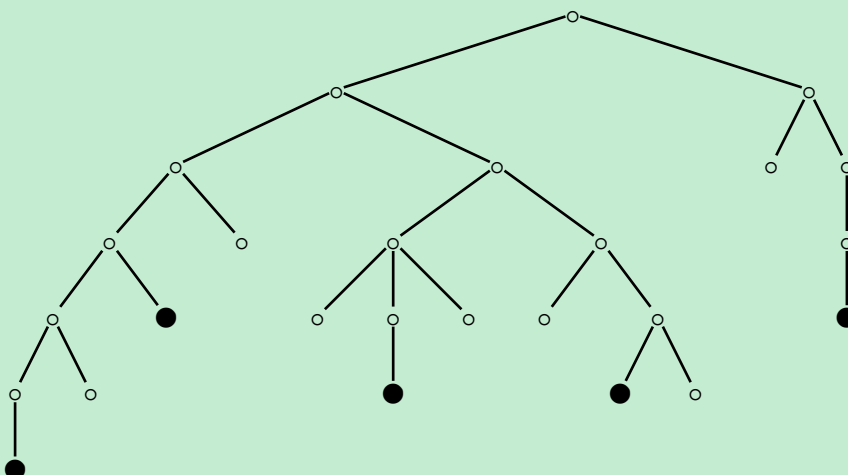
7.3. Lemma di Ogden

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

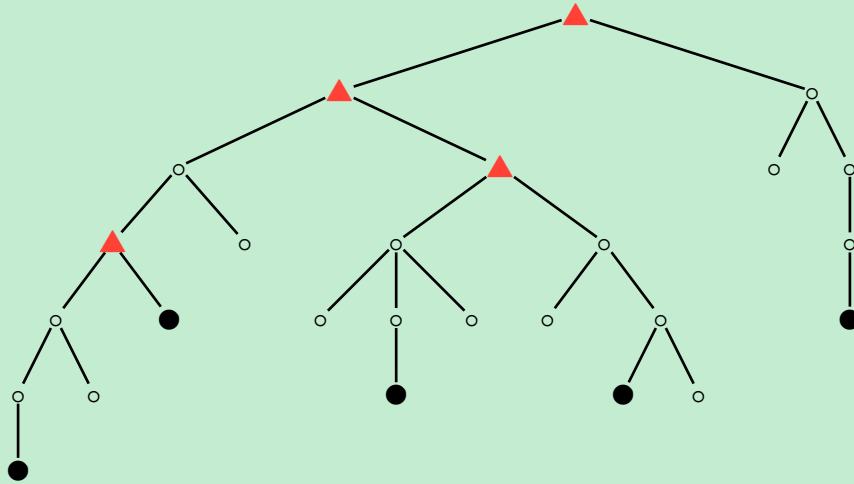
Esempio 7.3.1: Ci viene dato un albero di derivazione di una grammatica generica.



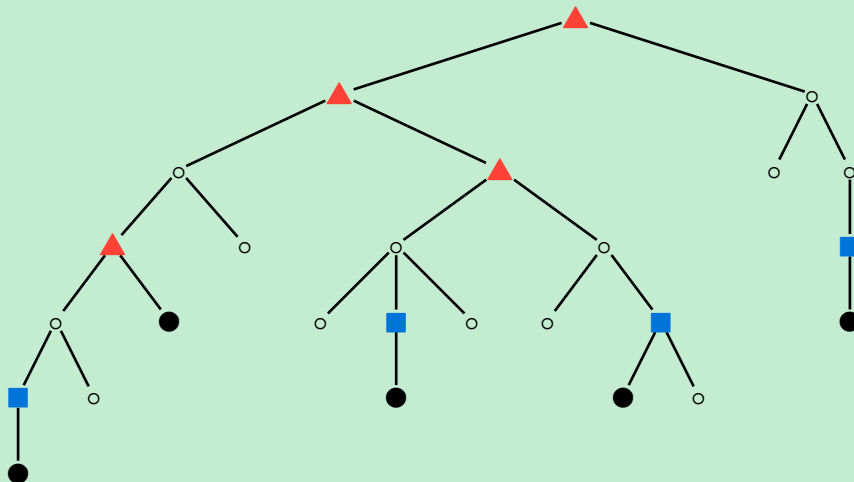
Andiamo a **marcare**, con un pallino nero grosso, alcune foglie dell'albero.



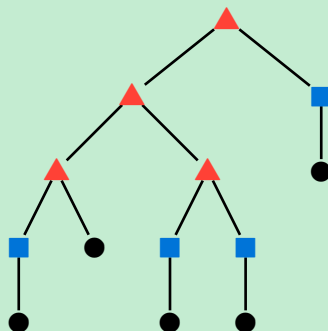
Una volta marcate alcune foglie dell'albero, individuiamo i **branch point**: essi sono nodi interni con almeno due figli, o loro discendenti, che sono nodi marcati. Indichiamo questi nodi con un triangolo rosso pieno.



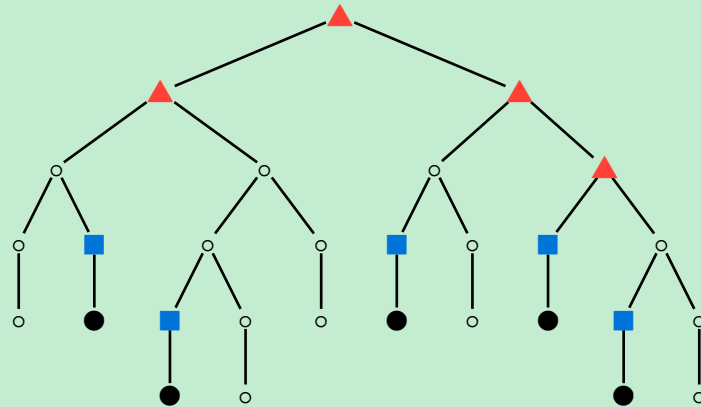
Identifichiamo infine con un quadrato blu pieno tutti i **padri** dei nodi marcati. Se un nodo è già stato segnato come branch point viene lasciato così.



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

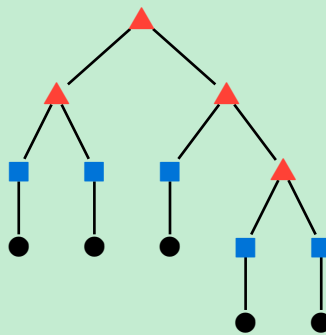


Esempio 7.3.2: Vediamo ora un albero in FN di Chomsky, già con nodi marcati e speciali.



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

Ora vediamo l'albero semplificato.



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

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

Lemma 7.3.1: Sia $G = (V, \Sigma, P, S)$ una grammatica in FN di Chomsky. Sia

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

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

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

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

Dimostrazione 7.3.1.1: Si dimostra per induzione, come il lemma della lezione scorsa. ■

Introduciamo finalmente il **lemma di Ogden**.

Lemma 7.3.2 (Lemma di Ogden): Sia $L \subseteq \Sigma^*$ un linguaggio CFL. Allora $\exists N > 0$ tale che $\forall z \in L$ in cui vengono marcate almeno N posizioni può essere decomposta come $z = uvwxy$ tale che:

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

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

Dimostrazione 7.3.2.1: La dimostrazione di questo teorema è analoga a quella del pumping lemma, ma ragiona sull'albero semplificato associato a quello di derivazione di z . ■

7.4. Applicazioni del lemma di Ogden

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

Esempio 7.4.1: Sia di nuovo

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Ma questo è assurdo, quindi L non è CFL.

Esempio 7.4.2: Definiamo ora

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

Possiamo vedere questo linguaggio come

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

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

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

7.5. Ambiguità

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

Esempio 7.5.1: Definiamo il linguaggio

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

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

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

Esempio 7.5.2: Diamo una grammatica per il linguaggio precedente. Le produzioni sono

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

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

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

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

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

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

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

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

$$A \Rightarrow a^k$$

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

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



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

Definizione 7.5.1 (Grado di ambiguità di una stringa): Sia $G = (V, \Sigma, P, S)$ una grammatica CF. Sia $w \in \Sigma^*$. Chiamiamo **grado di ambiguità** di w rispetto a G il numero di alberi di derivazione di w , oppure il numero di derivazioni leftmost di w .

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

Definizione 7.5.2 (Grado di ambiguità di una grammatica): Il **grado di ambiguità** di una grammatica G è il massimo grado di ambiguità delle stringhe $w \in \Sigma^*$.

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

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

Definizione 7.5.3 (Grado di ambiguità di un automa a pila): Il **grado di ambiguità** di un automa a pila è il numero di computazioni accettanti.

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

Definizione 7.5.4 (Grammatiche inerentemente ambigue): Sia L un CFL. Allora L è **inerentemente ambigua** se ogni grammatica CF per L è ambigua.

Andiamo avanti la prossima volta.

8. Lezione 19 [14/05]

8.1. Ambiguità

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

Lemma 8.1.1 (Lemma di Ogden): Sia $G = (V, \Sigma, P, S)$ una grammatica CF. Allora $\exists N$ tale che $\forall z \in L$ in cui sono marcate almeno N , possiamo scrivere z come $z = uvwxy$ con:

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

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

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

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

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

Dimostrazione 8.1.1.1: La dimostrazione cambia leggermente dalla scorsa volta.

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

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

e poi la dimostrazione va avanti allo stesso modo. ■

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

Esempio 8.1.1: Definiamo il linguaggio

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

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

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

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

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

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

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

Teorema 8.1.1: Il linguaggio L dell'Esempio 8.1.1 è inerentemente ambiguo.

Dimostrazione 8.1.1.2: Dobbiamo dimostrare che ogni grammatica G che genera L è ambigua, quindi esiste almeno una stringa in ogni G che è generata in almeno due modi.

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

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

Definiamo la stringa

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

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

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

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

Di questa stringa sappiamo che

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

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

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

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

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

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

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

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

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

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

Riprendiamo la terza condizione, considerando stringhe generiche nella forma

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

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

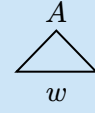
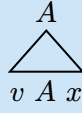
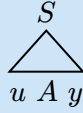
$$(i-1)j = m! \implies i = \frac{m!}{j} + 1$$

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

Con questa imposizione otteniamo la stringa

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

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



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

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

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

Quello che otteniamo è che:

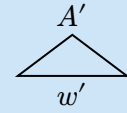
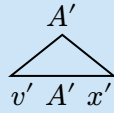
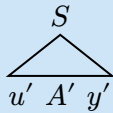
- i fattori v' e x' di z' sono formati rispettivamente dalle sole b e dalle sole c , ovvero sono le stringhe

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

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

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

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



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

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

Nel caso del linguaggio Esempio 8.1.1, il **grado di ambiguità** è 2. In alcuni casi, il grado di ambiguità cresce in base alla lunghezza della stringa che si sta riconoscendo, rendendo di fatto **infinito** il grado della grammatica e/o del linguaggio.

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

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

8.2. Ambiguità e non determinismo

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

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

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

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

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

definiti da una funzione di transizione

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

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

Definizione 8.2.1 (PDA deterministico): Il PDA M è **deterministico** se:

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

In poche parole, le due condizioni ci dicono che:

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

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

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

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

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

Esempio 8.2.1: Definiamo il linguaggio regolare

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

che possiamo riconoscere tranquillamente con un DFA a due stati.

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

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

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

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

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

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

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

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

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

Esempio 8.2.2: Sia L il linguaggio delle palindrome pari, ovvero

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

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

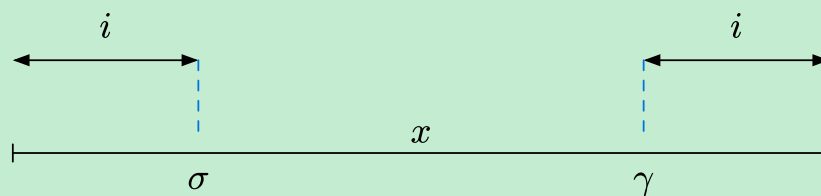
Anche se gli automi per questo linguaggio sono non deterministici, mi va bene una sola scommessa se la stringa è palindroma, quindi non abbiamo ambiguità.

Vediamo una grammatica G per L :

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

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

Esempio 8.2.3: Vediamo il complemento del linguaggio precedente, ovvero il linguaggio delle stringhe nelle quali esiste almeno una posizione alla stessa distanza dai bordi in cui i caratteri sono diversi.



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

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

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

Possiamo evitare l'ambiguità in questo automa?

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

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

Esempio 8.2.4: Per sfizio scriviamo L^C in termini di grammatica.

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

Le regole di produzione sono

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

$$T \longrightarrow aUb \mid bUa$$

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

9. Lezione 20 [16/05]

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

Introduciamo subito due linguaggi che ci serviranno durante la lezione.

Definizione 9.1 (Linguaggi comodi): Definiamo il **linguaggio**

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

e il **linguaggio**

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

entrambi DCFL, molto facile da dimostrare.

9.1. Operazioni insiemistiche

Partiamo con le **operazioni insiemistiche**.

9.1.1. Unione

9.1.1.1. CFL

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

Siano

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

due grammatiche CF. Creiamo la grammatica

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

formata da:

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

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

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

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

9.1.1.2. DCFL

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

Esempio 9.1.1.2.1: Prendiamo i due linguaggi definiti nella Definizione 9.1.

Abbiamo detto che sono entrambi DCFL, ma la sua unione

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

deve essere riconosciuta da un automa non deterministico.

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

9.1.2. Intersezione

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

9.1.2.1. CFL

Esempio 9.1.2.1.1: Prendiamo ancora i due linguaggi della Definizione 9.1.

Definiamo il linguaggio

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

che abbiamo visto non essere nemmeno CFL.

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

9.1.2.2. DCFL

Vale lo stesso discorso dei CFL.

9.1.3. Intersezione con un regolare

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

9.1.3.1. CFL

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

9.1.3.2. DCFL

Stesso discorso per i DCFL.

9.1.4. Complemento

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

9.1.4.1. CFL

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

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

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

Esempio 9.1.4.1.1: Definiamo il linguaggio

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

Questo linguaggio è CFL, ma il suo complemento

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

invece non è CFL.

Esempio 9.1.4.1.2: Al contrario, dato il linguaggio

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

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

9.1.4.2. DCFL

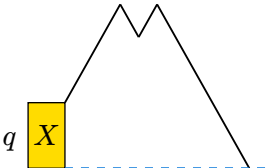
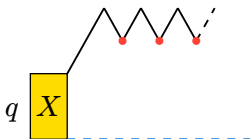
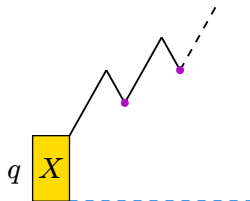
Nel caso deterministico non possiamo usare De Morgan perché non siamo chiusi l'unione, quindi nessun ragionamento di quel tipo ha senso.

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

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

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

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

Nessun loop	Loop sullo stesso piano	Loop crescente
		

In ordine abbiamo:

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

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

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

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

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

Costruiamo quindi l'automa

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

definito da:

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

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

- **Γ' alfabeto di lavoro** che aggiunge solo il carattere del truccaccio, ovvero

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Definiamo quindi M' formato da:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Finita, finalmente, questa costruzione senza senso.

9.1.5. Riassunto

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

9.2. Operazioni regolari

Vediamo ora le **operazioni regolari**.

9.2.1. Prodotto

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

9.2.1.1. CFL

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

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

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

9.2.1.2. DCFL

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

Esempio 9.2.1.2.1: Prendiamo di nuovo i linguaggi della Definizione 9.1, che sono entrambi deterministici, e creiamo il linguaggio

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

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

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

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

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

Analizziamo i vari casi:

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

Filtriamo alcune stringhe di L_1 calcolando

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

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

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

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

9.2.2. Star

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

9.2.2.1. CFL

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

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

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

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

9.2.2.2. DCFL

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

Esempio 9.2.2.2.1: Non ho ben capito l'esempio che ha scritto.

9.2.3. Riassunto

	CFL	DCFL

Prodotto	✓	✗
Star	✓	✗

10. Lezione 21 [21/05]

10.1. Riassunto chiusura operazioni

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

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

10.2. CFL vs DCFL

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

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

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

Esempio 10.2.1: Avevamo visto, con questa tecnica, la dimostrazione di

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

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

Esempio 10.2.2: Definiamo il linguaggio

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

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

Calcoliamo il suo complemento

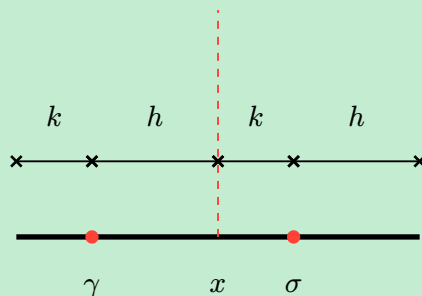
$$L^C = \{ww \mid w \in \{a, b\}^*\}.$$

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

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

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

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



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

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

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

Vediamo la soluzione divisa in fasi:

1. prima fase

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

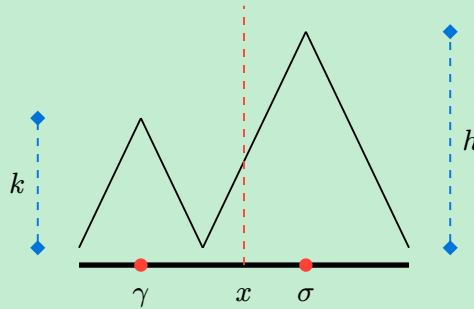
2. seconda fase

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

3. terza fase

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

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



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

Esempio 10.2.3: Vediamo una grammatica per il linguaggio precedente.

Le regole di produzione sono:

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

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

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

Ma allora stiamo generando della stringhe

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

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

Esempio 10.2.4: Ora che abbiamo visto un automa a pila e anche una grammatica per L , possiamo usare il primo risultato per dire che L non può essere deterministico perché con le proprietà di chiusura L^C dovrebbe essere DCFL.

Vediamo ora un altro linguaggio con un esempio.

Esempio 10.2.5: Definiamo quindi il linguaggio

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

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

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

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

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

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

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

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

Teorema 10.2.1: Se ogni classe di equivalenza di R ha cardinalità finita allora L non è DCFL.

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

Applichiamolo subito all'ultimo esempio visto.

Esempio 10.2.6: Definiamo di nuovo il linguaggio

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

Facciamo vedere che

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

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

Prendiamo quindi due stringhe generiche

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

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

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

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

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

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

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

perché

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

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

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

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

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

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

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

Se applichiamo questa stringa alle due che abbiamo a disposizione otteniamo

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

perché

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

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

10.3. Ricorsione

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

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

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

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

Definizione 10.3.1 (Grammatiche self-embedding): Prendiamo una grammatica $G = (V, \Sigma, P, S)$ context-free. Diciamo che G è **self-embedding** se

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

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



È importante che entrambe siano diverse dal vuoto:

- se è vuota α abbiamo ricorsione all'inizio, che si può eliminare;
- se è vuota β abbiamo ricorsione in coda, che si può eliminare.

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

Teorema 10.3.1: Se G non è self-embedding allora $L(G)$ è regolare.

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

Corollario 10.3.1.1: Se L è un linguaggio CFL e non regolare allora ogni G per L è self-embedding.

10.4. Linguaggio di Dyck

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

Definizione 10.4.1 (Linguaggio di Dyck): Definiamo l'alfabeto

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

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

Il **linguaggio di Dyck**

$$D_k \subseteq \Omega_k^*$$

è l'insieme delle parentesi bilanciate costruite sull'insieme Ω_k .

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

Teorema 10.4.1 (Teorema di Chomsky-Schutzenberger): Dato $L \subseteq \Sigma^*$ un CFL, allora:

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

tali che

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

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

Esempio 10.4.1: Definiamo il linguaggio

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

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

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

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

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

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

Esempio 10.4.2: Se prendiamo L il linguaggio delle parentesi bilanciate, allora scegliamo l'identità come morfismo e come linguaggio regolare quello che fa passare tutto.

Esempio 10.4.3: Definiamo il linguaggio

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

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

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

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

$$R = [(1 + (2)^*[1_1 + 2_2])^*].$$

Esempio 10.4.4: Definiamo infine il linguaggio PAL delle stringhe palindrome di lunghezza anche dispari. Qua dobbiamo modificare leggermente la soluzione precedente

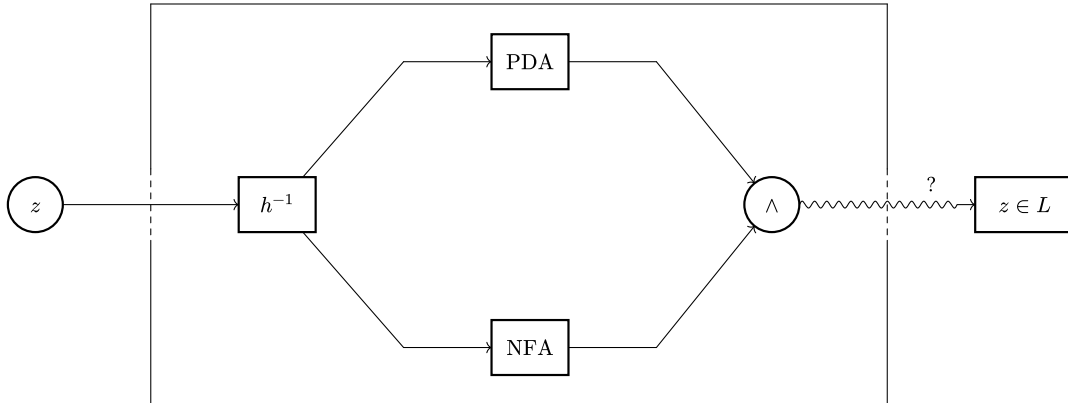
Scegliamo $k = 4$ usando le parentesi definite prima, alle quali aggiungiamo due parentesi, che usiamo per codificare l'eventuale simbolo centrale, che può essere una a o una b , quindi:

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

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

$$R = [(1 + (2)^*[\varepsilon + (3)_3 + (4)_4][1_1 + 2_2])^*].$$

Se non abbiamo a disposizione un riconoscitore per L , ma conosciamo tutto ciò che serve per costruirlo con il Teorema 10.4.1, ovvero conosciamo il morfismo h , il linguaggio di Dyck D_k e il linguaggio regolare R , possiamo **costruire un riconoscitore** per L .



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

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

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

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

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