

Teoria dei Linguaggi

Indice

1. Lezione 12 [04/04]	3
1.1. Problemi di decisione per i linguaggi regolari	3
1.1.1. Linguaggio vuoto e infinito	3
1.1.2. Appartenenza	4
1.1.3. Universalità	4
1.1.4. Inclusione e uguaglianza	4
1.2. Varianti di automi	4
1.2.1. Automi pesati	4
1.2.2. Automi probabilistici	4
2. Lezione 13 [09/04]	6
2.1. Varianti di automi	6
2.1.1. One-way VS two-way	6
2.1.2. Read-only VS read-write	6
2.1.3. Memoria esterna	7
2.2. Automi two-way	8
2.2.1. Esempi vari	8
2.2.2. Definizione formale	10
2.2.3. Potenza computazionale	11
3. Lezione 14 [11/04]	16
3.1. Simulazione	16
3.1.1. Problema di Sakoda & Sipser	16
3.2. Automi a pila	18
3.2.1. Versione non deterministica	18
3.2.2. Accettazione	21
3.2.3. Determinismo VS non determinismo	23
3.2.4. Trasformazioni	24
4. Lezione 15 [23/04]	26
4.1. Esempi	26
4.2. Equivalenza tra grammatiche di tipo 2 e automi a pila	26
4.2.1. Ripasso e introduzione	26
4.2.2. Da grammatica di tipo 2 ad automa a pila	29
4.2.3. Automa a pila a grammatica di tipo 2	32
4.3. Forme normali per le grammatiche context-free	32
4.3.1. FN di Greibach	32

1. Lezione 12 [04/04]

1.1. Problemi di decisione per i linguaggi regolari

I **problemi di decisione** sono dei problemi che hanno come unica risposta **SI** oppure **NO**. Sui linguaggi regolari abbiamo una serie di problemi di decisione interessanti che possono essere risolti in maniera automatica. Questa lista di problemi diventerà ancora più briosa quando andremo nei linguaggi context-free.

1.1.1. Linguaggio vuoto e infinito

Dato un linguaggio L possiamo chiederci se $L \neq \emptyset$, ovvero se L **non è vuoto**, o se L è **infinito**.

Lemma 1.1.1.1: Sia L un linguaggio regolare e sia N la costante del pumping lemma per L . Allora:

1. $L \neq \emptyset \iff L$ contiene almeno una stringa di lunghezza $< N$;
2. L è infinito $\iff L$ contiene almeno una stringa z con $N \leq |z| < 2N$.

Dimostrazione 1.1.1.1.1: Partiamo con la dimostrazione del punto 1.

[\Leftarrow] Se L contiene una stringa di lunghezza $< N$ allora banalmente $L \neq \emptyset$.

[\Rightarrow] Se $L \neq \emptyset$ sia $z \in L$ la stringa di lunghezza minima in L . Per assurdo sia $|z| \geq N$, ma allora per il pumping lemma possiamo dividere z come $z = uvw$. Sappiamo dal terzo punto che possiamo ripetere un numero arbitrario di volte la stringa v e stare comunque in L . Ripetiamo v un numero di volte pari a 0: otteniamo la stringa $z' = uw$ che appartiene a L per il pumping lemma. Avevamo detto che z era la stringa più corta di L , ma abbiamo appena mostrato che $|z'| < |z|$: questo è assurdo e quindi $|z| < N$.

Dimostriamo ora il punto 2.

[\Leftarrow] Se L contiene una stringa z di lunghezza $N \leq |z| \leq 2N$, visto che $|z| \geq N$ applichiamo il pumping lemma per scomporre z in $z = uvw$. Per il terzo punto possiamo ripetere un numero arbitrario di volte il fattore v e rimanere comunque in L . Ma allora L è infinito.

[\Rightarrow] Se L è infinito, sicuramente esiste una stringa z che è lunga almeno N . Tra tutte queste stringhe, scegliamo la più corta di tutte. Per assurdo sia $|z| \geq 2N$, ma allora possiamo scomporre z come $z = uvw$ con $|uv| \leq N$ e $v \neq \epsilon$. Adesso andiamo a ripetere un numero di volte pari a 0 il fattore v , ottenendo $z' = uw \in L$. Quale è la sua lunghezza? Possiamo dire che $|z'| = |z| - |v|$ ma $|v|$ è al massimo N per il primo punto analizzato, quindi $|z'|$ è almeno N e al massimo $2N$, ma questo è assurdo perché z era la più corta tra tutte le stringhe e, per assurdo, l'avevamo posta di lunghezza $\geq 2N$. ■

Questo lemma ci dice che se vogliamo sapere se un linguaggio non è vuoto basta generare tutte le stringhe di lunghezza fino a N escluso e vedere se ne abbiamo una nel linguaggio, mentre se vogliamo sapere se un linguaggio è infinito basta generare tutte le stringhe di lunghezza compresa tra N incluso e $2N$ escluso e vedere se ne abbiamo una nel linguaggio.

Quale è il problema? Sicuramente è un approccio **inefficiente**, visto che dobbiamo generare un numero esponenziale di casi da analizzare. Possiamo fare di meglio? **SI**: per vedere se un

linguaggio non è vuoto devo cercare un **cammino** dallo stato iniziale ad uno stato finale, mentre per vedere se un linguaggio è infinito potrei cercare i **cicli** sui cammini del punto precedente. Ci siamo quindi ricondotti a dei **problemi su grafi**, che sappiamo risolvere efficientemente.

1.1.2. Appartenenza

Dato L un linguaggio regolare e $x \in \Sigma^*$ una stringa, il problema di **appartenenza** si chiede se $x \in L$. Questo lo sappiamo fare tranquillamente in tempo lineare: basta eseguire il DFA (se ce l'abbiamo) e vedere se finiamo in uno stato finale.

1.1.3. Universalità

Dato L un linguaggio regolare, il problema di **universalità** si chiede se $L = \Sigma^*$, ovvero L contiene tutte le stringhe della chiusura di Kleene dell'alfabeto Σ . Questo sembra difficile, ma possiamo sfruttare le operazioni che rendono chiusi i linguaggi regolari: infatti, possiamo chiederci invece se

$$L = \Sigma^* \iff L^C = \emptyset$$

e questo lo sappiamo fare grazie al lemma precedente.

1.1.4. Inclusione e uguaglianza

Infine, l'ultimo problema che vediamo prende due linguaggi L_1 e L_2 regolari e si chiede se $L_1 \subseteq L_2$. Questo problema si chiama problema dell'**inclusione** e lo possiamo risolvere manipolando quello che ci viene chiesto: infatti, al posto dell'inclusione, possiamo chiederci se

$$L_1 \subseteq L_2 \iff L_1 \cap L_2^C = \emptyset,$$

che sappiamo fare tranquillamente grazie al lemma.

Se non volessimo utilizzare le proprietà di chiusura, possiamo costruire un **automa prodotto** che ha come stati finali tutte le coppie di stati dove il primo accetta L_1 e il secondo rifiuta L_2 .

E se invece volessimo risolvere il problema di **uguaglianza**, ovvero quello che si chiede se $L_1 = L_2$? Basta dimostrare la doppia inclusione $L_1 \subseteq L_2 \wedge L_2 \subseteq L_1$ e siamo a cavallo. Un algoritmo diverso utilizza le classi di equivalenza, ma non lo vedremo.

1.2. Varianti di automi

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

1.2.1. 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.

1.2.2. 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.

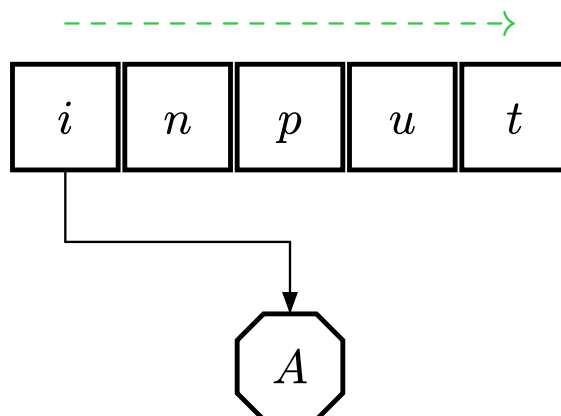
Questi automi li possiamo usare come **riconoscitori a soglia**: tutte le parole oltre una certa soglia le accettiamo, altrimenti le rifiutiamo.

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

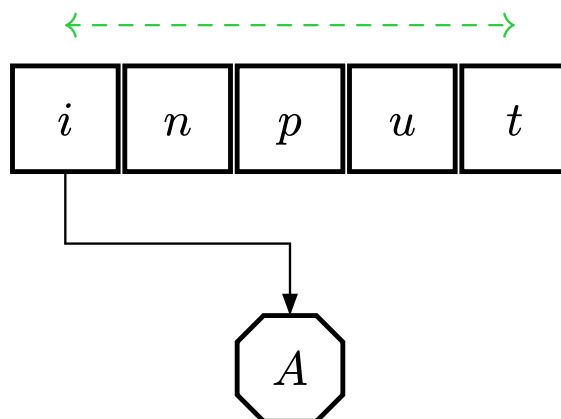
Come possiamo **rappresentare** un automa a stati finiti? Questa macchina è molto semplice: abbiamo un **nastro** che contiene l'input, esaminato da una **testina in sola lettura** che, spostandosi **one-way** da sinistra verso destra, permette ad un **controllo a stati finiti** di capire se la stringa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



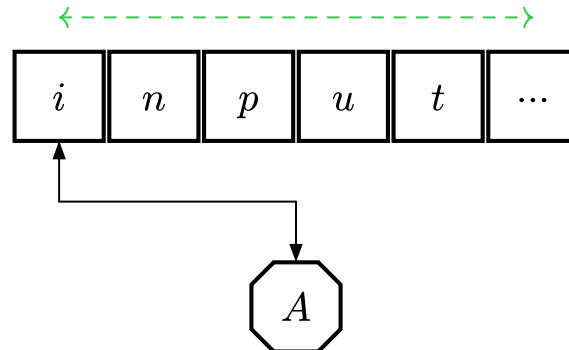
2.1.2. Read-only VS read-write

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.

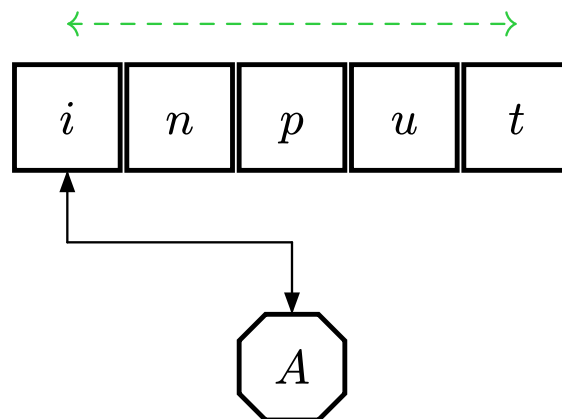
Consideriamo quindi un automa two-way che però mantiene la read-only del nastro: la classe che otteniamo è ancora una volta quella dei **linguaggi regolari**, e questo lo vedremo oggi.

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**:



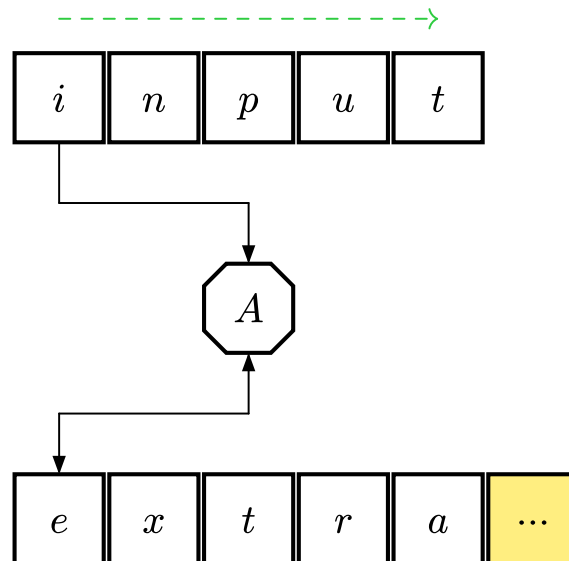
- se invece lasciamo il nastro grande quando l'input andiamo a riconoscere i linguaggi di tipo 1, ovvero otteniamo un **automa limitato linearmente**. Quest'ultima cosa vale perché nelle grammatiche di tipo 1 le regole di produzione non decrescono mai, e un automa limitato linearmente per capire se deve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



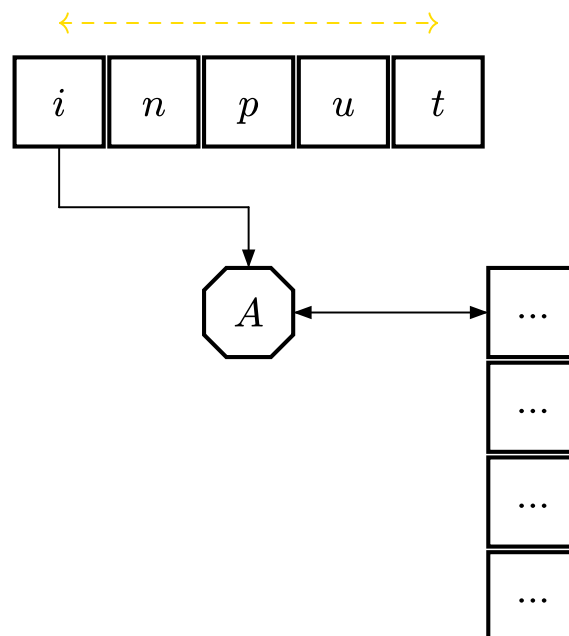
2.1.3. Memoria esterna

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

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

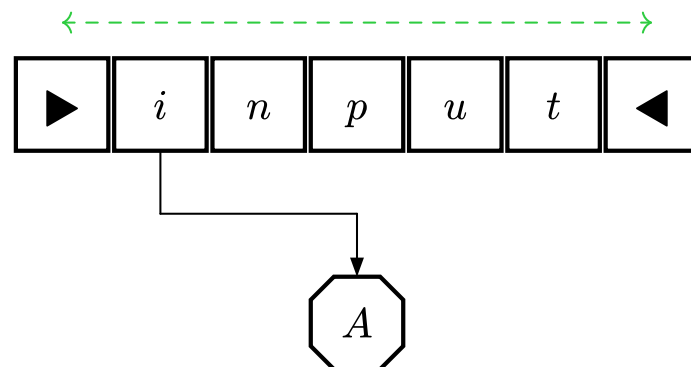
Abbiamo A un DFA che accetta L , come posso costruire un two-way per $\alpha(L)$? Potremmo leggere x la prima volta, ricordarci in che stato siamo arrivati, tornare indietro e poi ripartire a leggere x dallo stato nel quale eravamo arrivati e vedere se finiamo in uno stato finale.

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.

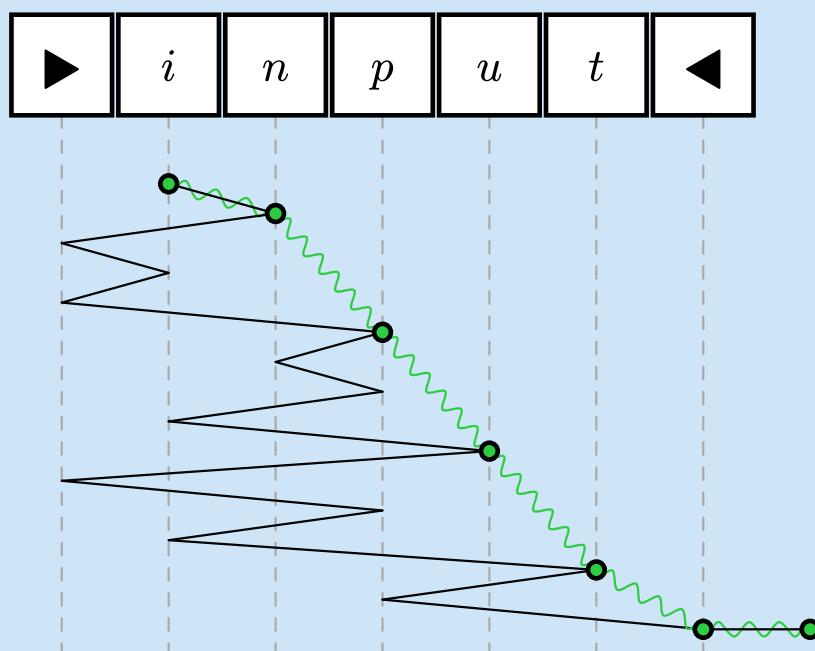
2.2.3. Potenza computazionale

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

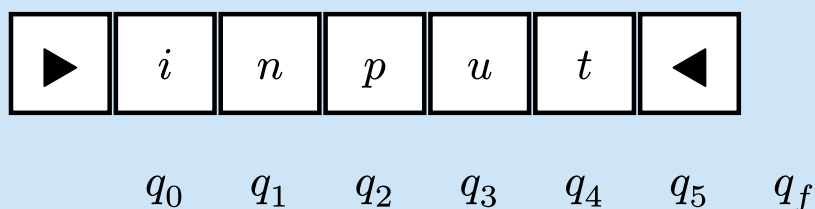
$$L(2DFA) = L(1DFA).$$

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

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

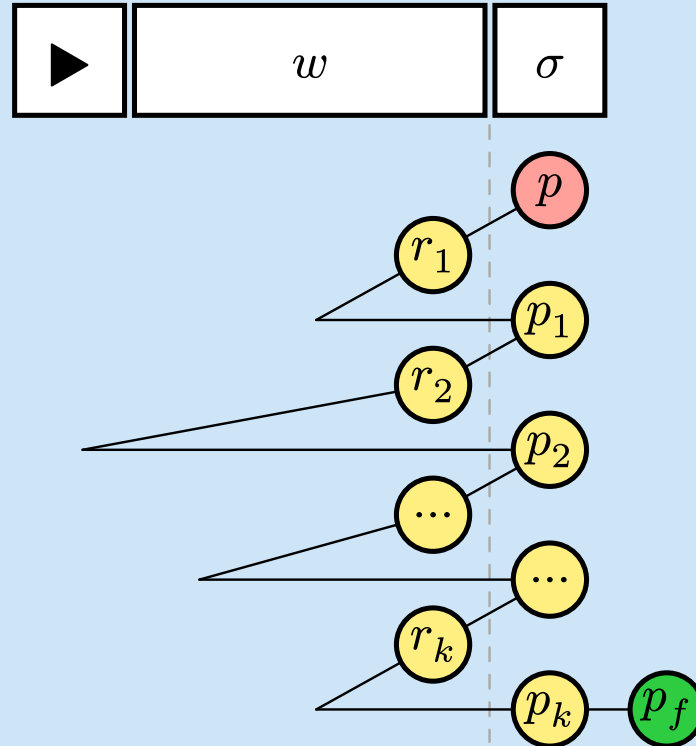
Supponiamo di aver calcolato la tabella di w , vediamo come costruire induttivamente la tabella di $w\sigma$, con $w \in \Sigma^*$ e $\sigma \in \Sigma$. Se vale

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

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 potrei 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

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

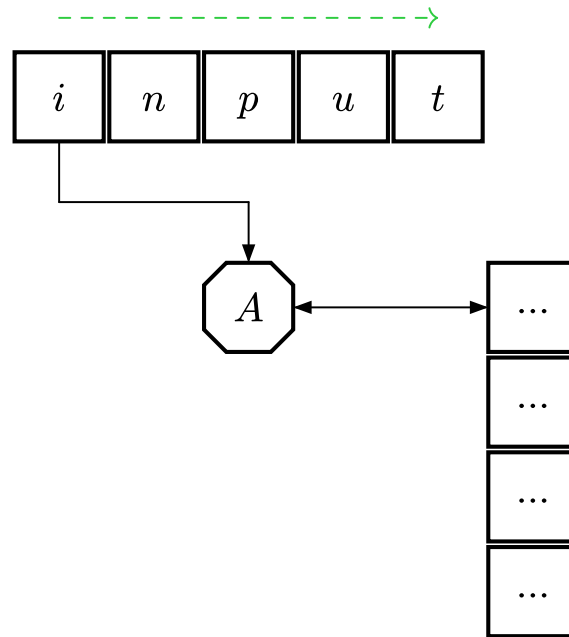
- ▶ se facciamo la trasformazione da 1NFA a 2DFA l'upper bound diventa esattamente n^2 , quindi la trasformazione fatta è ottimale.

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:

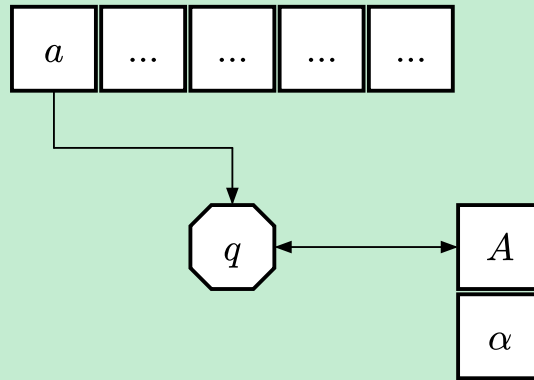
$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow \text{PF}(Q \times \Gamma^*).$$

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:

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

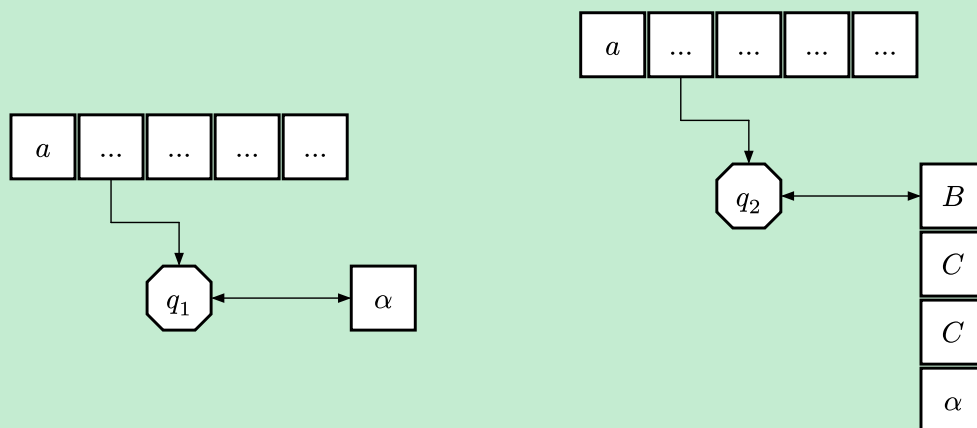


Con α nel disegno si intende una stringa in Γ^* perché oltre ad A potremmo avere altro.

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.

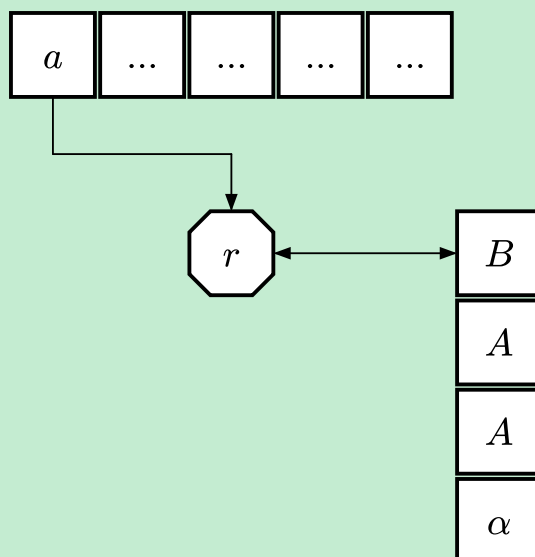


Per convenzione, quando inseriamo una stringa sulla pila, l'inserimento avviene da destra verso sinistra. In poche parole, se inseriamo la stringa $X \in \Gamma^*$ nella pila, se la togliessimo noi leggeremmo, in ordine, esattamente X . In altre parole ancora, quando leggiamo una stringa da inserire 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 pila 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'automata M andiamo ad inserire un carattere X in fondo alla pila, vedremo dopo perché;

- l'automata 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

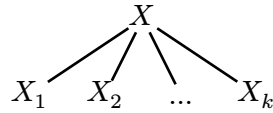
Una grammatica G di tipo 2 ha le **regole di produzione** nella forma

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

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

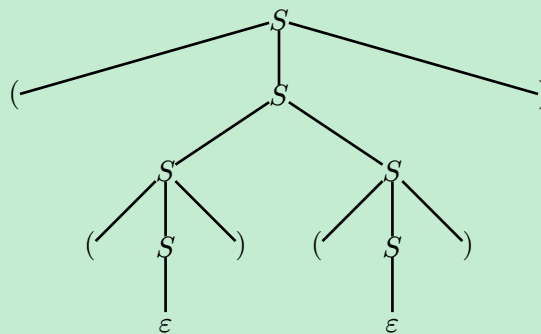
Il processo di derivazione nelle grammatiche di tipo 2 può essere espresso mediante **alberi di derivazione**: essi sono alberi che visualizzano l'applicazione delle regole di produzione.

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.

[illegible]

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

27

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	$abbbaa$	$abbUaa$	Avendo a disposizione il non determinismo, sono fortunato e scelgo la derivazione corretta e, come prima, non sposto la testina
U a a	$abbbaa$	$abbUaa$	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.