Teoria dei Linguaggi

Indice

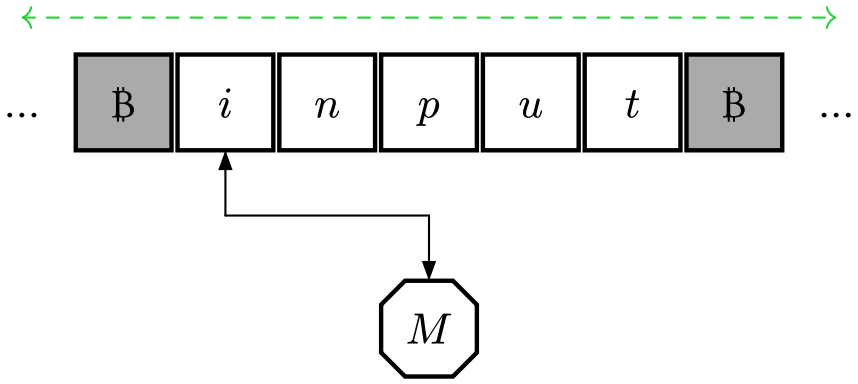
1. Lezione 23 [28/05]

Finiamo la gerarchia di Chomsky parlando finalmente di **macchine di Turing**.

1.1. Macchine di Turing

Come costruiamo una macchina di Turing?

Partiamo da un **automa a stati finiti**: questa macchina è one-way/two-way con un nastro che non può essere riscritto. Se aggiungiamo la possibilità di testina read-write allora otteniamo un **automa limitato linearmente**. Per ottenere finalmente una **macchina di Turing** dobbiamo rompere il vincolo di memoria pari alla lunghezza dell'input, togliendo i marcatori ▶◀ e inserire due porzioni di nastro potenzialmente infinite. Queste celle che non contengono l'input contengono il simbolo ฿.



A differenza degli LBA abbiamo **memoria illimitata**, quindi se ci serve dello spazio ce l'abbiamo sempre a disposizione per fare conti o per ricordarci qualcosa.

Questa memoria infinita ci permette di riconoscere i linguaggi di tipo 0. Infatti, le MdT sono **equivalenti** alle grammatiche di tipo 0. Vediamo perché.

Teorema 1.1.1: Le macchine di Turing sono equivalenti alle grammatiche di tipo 0.

Dimostrazione 1.1.1.1: Vediamo le due trasformazioni.

[**Grammatica → macchina di Turing**]

Nelle grammatiche di tipo 1 la condizione sulle produzioni $\alpha \longrightarrow \beta$ tale che

$$|\alpha| \leq |\beta|$$

ci permetteva di ricreare le produzioni al contrario sul nastro con la certezza di restare dentro il nastro per via della lunghezza della stringa che inserivamo.

Nelle grammatiche di tipo 0 invece non abbiamo vincoli, quindi ricreare le derivazioni al contrario potrebbe sforare il nastro, ma ora che abbiamo spazio a disposizione lo possiamo sfruttare per fare tutte le derivazioni possibili.

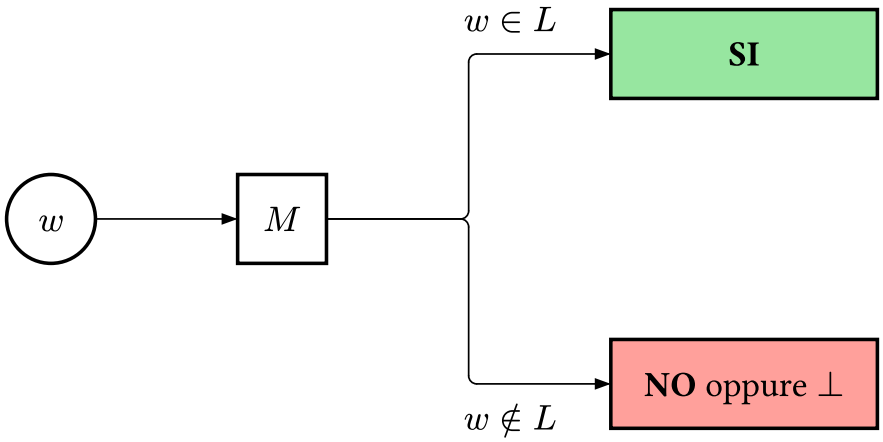
[**Macchina di Turing → grammatica**]

Si può fare, ci fidiamo. ■

Questa limitazione che non abbiamo più sulle grammatiche causa anche la **semi-decidibilità** dei linguaggi di tipo 0, o se vogliamo, il **riconoscimento parziale** dei linguaggi di tipo 0. Questo non succedeva nei linguaggi di tipo 1 perché la condizione sulle produzioni faceva terminare ad un certo punto le stringhe possibili generabili.

Con **semi-decidibilità** di un linguaggio L di tipo 0 intendiamo che:

- se una stringa appartiene a L allora la macchina si ferma e accetta;
- se una stringa non appartiene a L allora la macchina:
  - può fermarsi e non accettare;
  - può entrare in **loop infinito**.



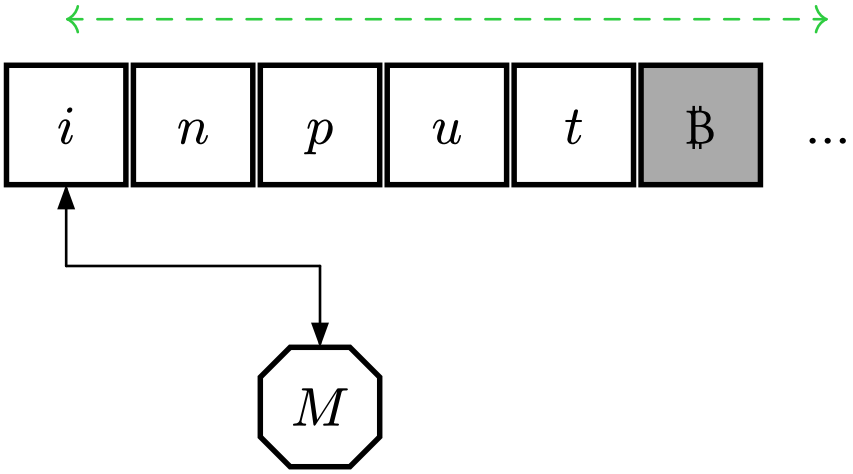
Le macchine di Turing sono un modello in grado di calcolare tutte le **funzioni calcolabili** da un computer. Queste funzioni sono dette **funzioni ricorsive**, ma con «ricorsive» non intendiamo le funzioni che chiamano sé stesse.

1.2. Varianti di MdT

Quella che abbiamo visto ora è una **MdT a 1 nastro**. Da questa nostra base possiamo tirare fuori molte **varianti** interessanti.

1.2.1. Nastro semi-infinito

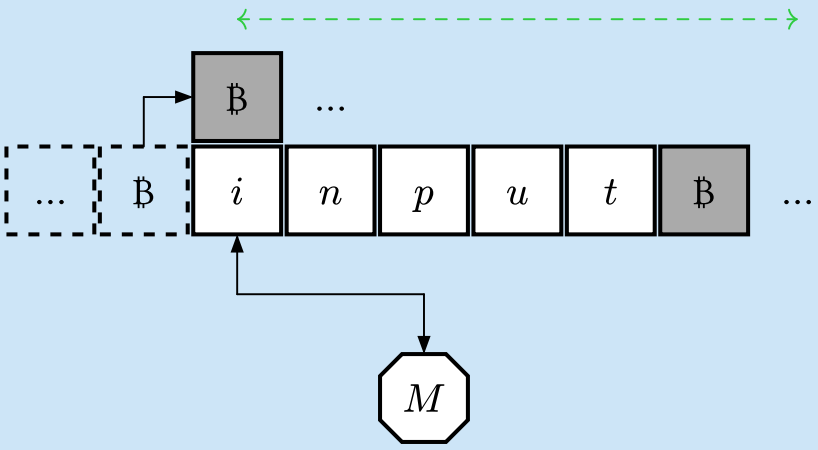
Abbiamo visto che limitando la lunghezza del nastro della macchina collassiamo nei linguaggi context-sensitive, ma se limitiamo il nastro da una parte sola otteniamo una MdT con **nastro semi-infinito**, o **infinito a destra**.



Lemma 1.2.1.1: Una MdT con nastro semi-infinito è equivalente ad una MdT a 1 nastro.

Dimostrazione 1.2.1.1.1: Data una MdT con nastro semi-infinito, è facilissimo simularla con una MdT a 1 nastro perché basta fare tutte le mosse che già fa la macchina data.

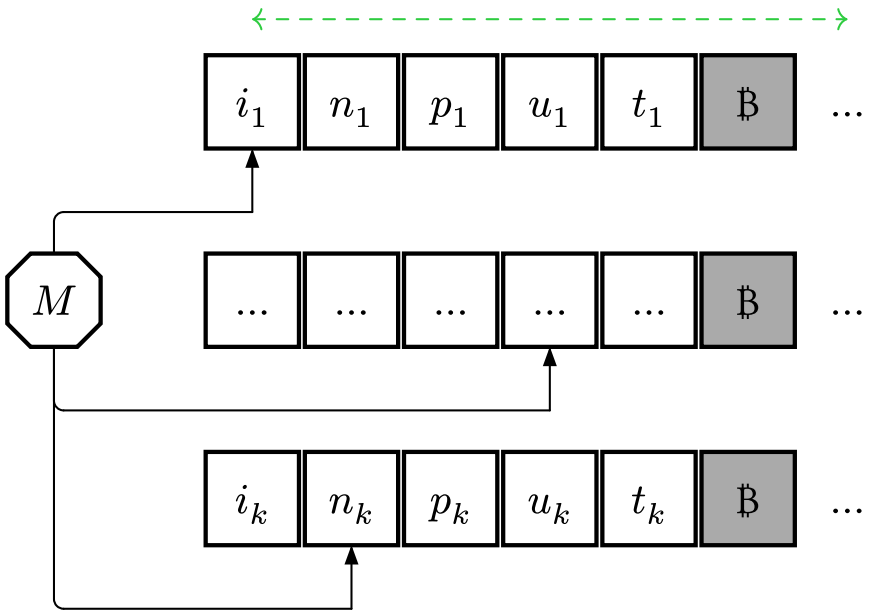
Dato invece una MdT a 1 nastro, la possiamo simulare con una MdT con nastro semi-infinito «incollando» la parte sinistra dell'input sopra al nastro, creando una nuova **traccia**.



■

1.2.2. Nastri multipli

Una **MdT con k nastri**, di cui almeno uno infinito semi-infinito, è una variante un pelo più complicata. Infatti, oltre ad avere k nastri diversi, ognuno di questi ha la propria testina, che può essere in punti diversi durante la computazione.



Lemma 1.2.2.1: Una MdT con k nastri è equivalente ad una MdT a 1 nastro.

Dimostrazione 1.2.2.1.1: Data una MdT con 1 nastro, è facilissimo simularla con una MdT a k nastri perché basta fare tutte le mosse su un nastro solo dei k a disposizione, ovviamente scegliendo uno di quelli infiniti o semi-infiniti.

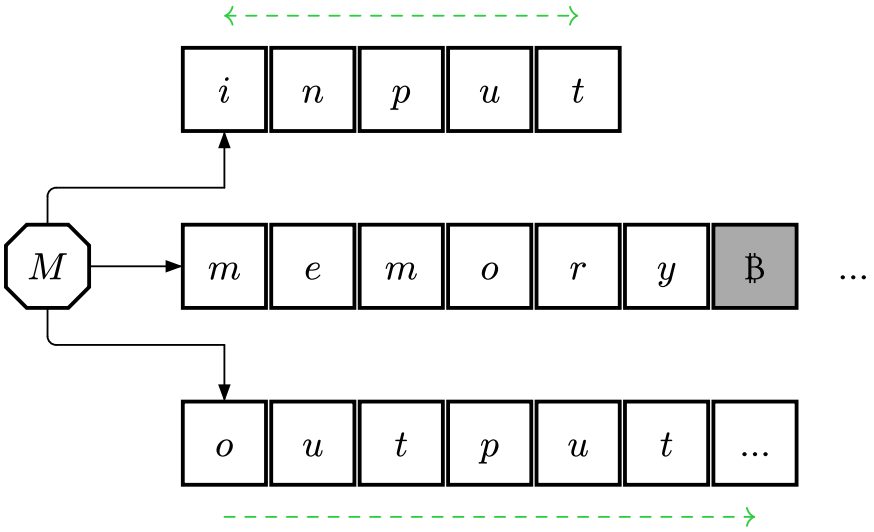
Dato invece una MdT a k nastri, la possiamo simulare con una MdT a 1 nastro rendendo i simboli delle k-tuple, come se avessimo appunto k tracce a disposizione. Per indicare dove sono le testine in ogni momento, marchiamo i caratteri corrispondenti con un puntino.

La simulazione con questo setup è poi possibile, parola di Roberto Carlino. ■

1.2.3. Nastri dedicati

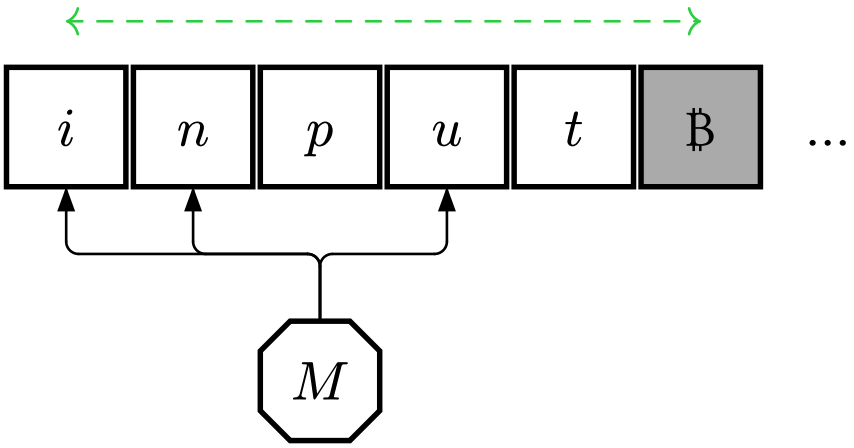
Una versione di MdT a più nastri è quella che **dedica un nastro al solo input**, che quindi ha la testina in sola lettura. Ci sono poi k nastri usati come **memoria**.

Una versione ancora migliorata ha un nastro per il solo **output**, in cui scriviamo e basta senza poter tornare indietro, perché ovviamente è un output.



1.2.4. Testine multiple

Rimaniamo sulle MdT a 1 nastro ma inseriamo **testine multiple**.



Lemma 1.2.4.1: Una MdT con k testine è equivalente ad una MdT a 1 nastro.

Dimostrazione 1.2.4.1.1: Data una MdT con 1 nastro, è facilissimo simularla con una MdT a k testine perché basta fare tutte le mosse con una testina delle k a disposizione.

Dato invece una MdT a k testine, la possiamo simulare con una MdT a 1 nastro marcando i simboli come nella variante precedente. ■

Se nelle MdT questa variante non aumenta la potenza, nei **linguaggi regolari** lo fa.

Esempio 1.2.4.1: Nei linguaggi regolari non riusciamo a riconoscere

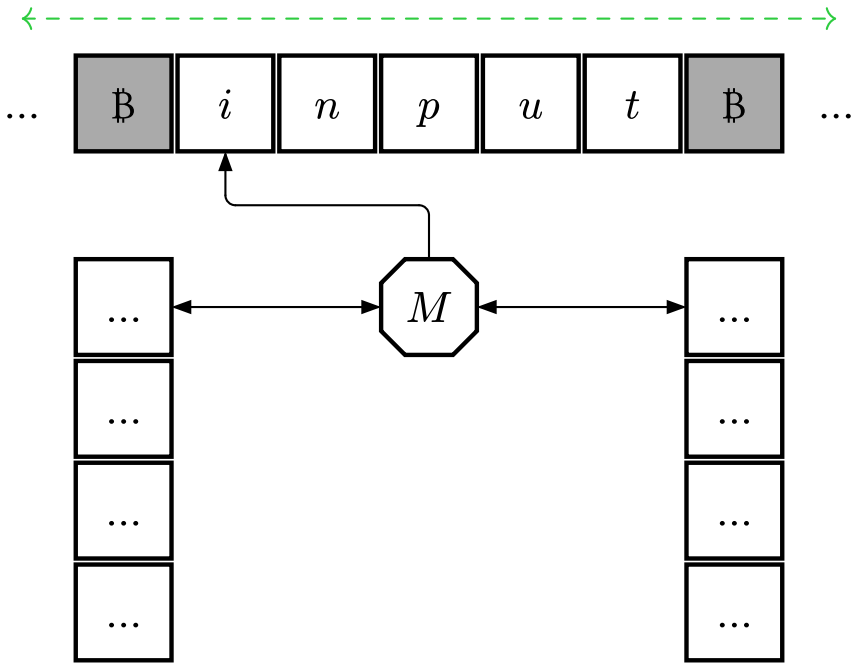
$$L = \{a^n b^n \mid n \geq 0\}.$$

Usando due testine, posizionandole una all'inizio delle a e una all'inizio delle b, riusciamo a riconoscere questo linguaggio contando di volta in volta una coppia di caratteri.

Ma allora in questo caso la potenza computazionale aumenta.

1.2.5. Automi a pila doppia

Negli automi a pila avevamo visto che non potevamo ricreare l'automa prodotto per l'intersezione e l'unione perché avevamo a disposizione una sola pila. Supponiamo di avere ora un **PDA con due pile**.



Con questa configurazione possiamo simulare una macchina di Turing. Prendiamo una MdT che ha sul nastro di lavoro α e β concatenati, con la testina sul primo carattere di β. Creiamo un automa a pila doppia che abbia α^R sulla prima pila e β sulla seconda. Con questa configurazione se leggiamo qualcosa da β lo facciamo dalla seconda pila, mentre se leggiamo α lo dobbiamo fare rovesciato.

1.3. Determinismo VS non determinismo

Vediamo adesso i modelli **deterministici** e **non deterministici**. Se consideriamo PDA e LBA, la classe di complessità di macchine deterministiche è diversa dalla classe di complessità di macchine non deterministiche. Questa distinzione non c'era invece nelle FSM perché con la **costruzione per sottoinsiemi** noi riuscivamo a simulare tutte le computazioni in parallelo usando degli insiemi di stati, pagando un costo in termini di stati.

Anche con le MdT non abbiamo questa distinzione.

Se consideriamo un **modello deterministico**, una computazione ha una sola strada da percorrere, mentre se consideriamo un **modello non deterministico**, una computazione può avere più strade possibili, Se anche solo una di queste strade dice **SI** allora accettiamo la stringa.

Teorema 1.3.1: Le Mdt deterministiche sono equivalenti alle MdT non deterministiche.

Dimostrazione 1.3.1.1: Ovviamente, una MdT non deterministica simula alla grande una MdT deterministica visto che ha una sola scelta disponibile.

Se abbiamo invece una MdT non deterministica, svogliamo un caso alla volta.

Supponiamo che per ora le computazioni siano **finite**, ovvero che terminano tutte. Eseguo ora la mia computazione in maniera deterministica:

- se arrivo in una configurazione accettante mi fermo e **accetto**;
- se arrivo in una configurazione non accettante posso usare la memoria infinita a mia disposizione per ricordarmi le scelte che ho fatto e fare quindi **backtracking**.

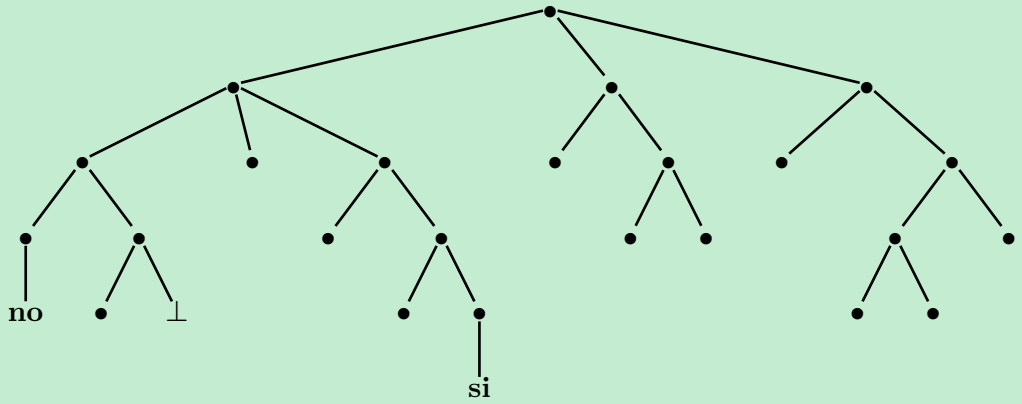
In poche parole, faccio un attraversamento dell'albero di computazione, e ogni volta che trovo un **SI** mi fermo e accetto, altrimenti torno indietro.

Supponiamo ora di avere anche computazioni che non terminano. La tecnica più semplice è usare un **clock**, ovvero dare un numero massimo di passi possibili. Se entro il clock non si trova un **SI** si raddoppia il clock e si rifà da capo la ricerca.

Se esiste una configurazione accettante la devo trovare in un numero finito di passi, altrimenti vado avanti all'infinito. ■

Vediamo un esempio di come funziona il **backtracking**.

Esempio 1.3.1: Ci viene dato il seguente albero di computazione.



Eseguendo una ricerca nel caso senza $\perp$, quando troviamo il $\perp$ di sinistra facciamo backtracking e torniamo indietro per cercare vie alternative. Se invece siamo nel caso con $\perp$, quando arriviamo a sforare il clock perché siamo finiti in $\perp$ allora raddoppiamo il timer e speriamo di trovarlo in quel numero di iterazioni.

1.4. Definizione formale

Vediamo finalmente la **definizione formale** di una MdT a 1 nastro.

Una MdT è una tupla

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

definita da:

- Q **insieme finito di stati**;
- Σ **alfabeto finito di input**;
- Γ **alfabeto finito di lavoro** che contiene sicuramente Σ visto che andiamo a scrivere sullo stesso nastro dove abbiamo l'input; contiene inoltre ฿, che però non sta in Σ, ovvero

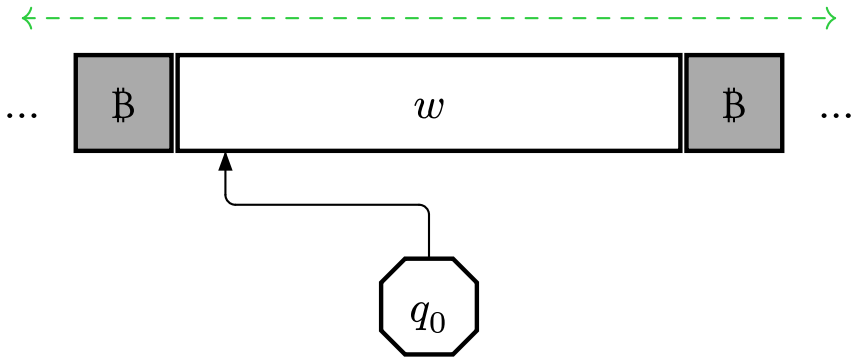
$$\Sigma \subseteq \Gamma \wedge ฿ \in \Gamma / \Sigma;$$

- δ **funzione di transizione** che permette di processare le stringhe date in input. Essa è la funzione

MdT deterministica	**MdT non deterministica**
$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{-1, 0, 1\}$	$\delta : Q \times \Gamma \longrightarrow 2^{Q \times \Gamma \times \{-1,0,1\}}$

  che, dati lo stato corrente e il simbolo sulla testina, calcola il nuovo stato, il nuovo simbolo da mettere sul nastro e la mossa da eseguire. La mossa con movimento 0 (nullo) può essere sostituita con una mossa a destra e una mossa a sinistra consecutiva (o viceversa). Assumiamo inoltre di non scrivere mai ฿ nella porzione dove c'è scritto l'input perché il simbolo ฿ lo usiamo un po' come separatore;
- q_0 **stato iniziale** della macchina;
- $F \subseteq Q$ **insieme finito di stati finali**.

Una **configurazione** è una foto della macchina in un dato istante di tempo. All'accensione della MdT la **configurazione iniziale** su input w contiene tutto l'input w sul nastro, la testina sul primo carattere di w e la macchina nello stato q_0.



Una **configurazione accettante** è una qualsiasi configurazione che si trova in uno stato finale, a prescindere da quello che troviamo sul nastro. Questo è abbastanza strano, ovvero **non siamo obbligati a leggere tutto l'input**, questo perché magari devo riconoscere solo una parte iniziale dell'input. Si può forzare tutta la lettura obbligando la macchina a scorrere tutto l'input prima di poter andare in uno stato finale. Inoltre, assumiamo per semplicità che quando la macchina entra in uno stato finale allora essa **si ferma** e **accetta** l'input fornito.

Come possiamo scrivere queste configurazioni? Come descriviamo lo stato di una MdT?

Per descrivere a pieno lo stato di una MdT in un dato momento dobbiamo sapere:

- l'**input** presente sul nastro, e questo lo ricaviamo vedendo la porzione di nastro racchiusa tra ฿;
- la posizione della **testina**;
- lo **stato** nel quale ci troviamo.

La posizione della testina è la più «difficile» da indicare perché non abbiamo una numerazione delle celle, quindi dobbiamo trovare una soluzione alternativa. Chiamiamo x la parte di stringa che si trova **prima** della testina e y la parte di stringa che contiene il **carattere indicato dalla testina** e il **resto** della stringa fino ai ฿ di destra.

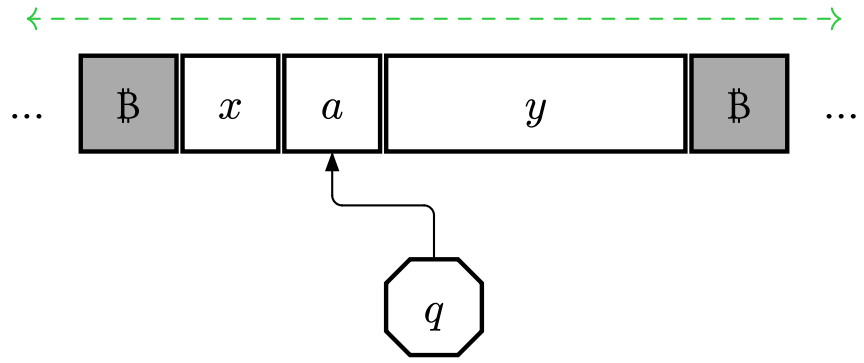
Con questo truccaccio riusciamo a fare una fotografia completa:
- l'**input** l'abbiamo spezzato in xy ma ce l'abbiamo per intero;
- la posizione della **testina** l'abbiamo implicitamente segnata indicando la divisione della stringa di input;
- lo **stato** basta indicarlo e siamo a posto.

La configurazione di una MdT nello stato q con input $w = xy$ è quindi una **tripla**

$$xqy \mid x, y \in \Gamma^* \wedge q \in Q.$$

È molto comodo di solito **evidenziare** il primo simbolo a di y perché lo utilizza la funzione di transizione, quindi ogni tanto configurazioni diamo la forma

$$xqay \mid x, y \in \Gamma^* \wedge a \in \Gamma \wedge q \in Q.$$



Esempio 1.4.1: Se la configurazione è

$$wq$$

vuol dire che mi trovo nello stato q avendo letto tutto l'input, ovvero mi trovo con la testina sul primo ฿ di destra.

La **configurazione iniziale** su input w è

$$q_0 w.$$

Una **configurazione accettante** è una qualsiasi configurazione che si trovi in uno stato finale, ovvero

$$xqy \mid x, y \in \Gamma^* \wedge q \in F.$$

Come passiamo da una configurazione all'altra usando la funzione di transizione? Sia

$$C = xqy \mid x = x'c \wedge y = ay'$$

la **configurazione corrente**. La configurazione che segue C si calcola con la **funzione di transizione** e dipende dalla mossa che questa funzione ci ritorna, ovvero:

- se
$$\delta(q, a) = (p, b, 0)$$
vuol dire abbiamo cambiato la a sulla testina con una b, abbiamo cambiato stato da q a p e non ci siamo spostati, ovvero
$$xqay' \vdash xpby';$$
- se
$$\delta(q, a) = (p, b, +1)$$
vuol dire abbiamo cambiato la a sulla testina con una b, abbiamo cambiato stato da q a p e ci siamo spostati avanti di una posizione, ovvero
$$xqay' \vdash xbpy';$$
- se
$$\delta(q, a) = (p, b, -1)$$
vuol dire abbiamo cambiato la a sulla testina con una b, abbiamo cambiato stato da q a p e ci siamo spostati indietro di una posizione, ovvero
$$xqay' \vdash x'pcby'.$$

Il **linguaggio riconosciuto** dalla MdT M è l'insieme

$$L(M) = \left\{ w \in \Sigma^* \mid \exists q \in F \wedge \exists x, y \in \Gamma^* \mid q_0 w \overset{*}{\vdash} xqy \right\}.$$

1.5. Linguaggi utili

Come vediamo, quando passiamo da una configurazione alla successiva, l'area in cui facciamo i cambiamenti è ristretta: di solito quello che cambia è il simbolo sulla testina, con uno spostamento opzionale di quest'ultima. Questo ci servirà per riconoscere dei linguaggi molto particolari che sono **basati sulle configurazioni**.

Definiamo un nuovo alfabeto

$$\Upsilon = \Gamma \cup Q \cup \{\#\} \mid \# \notin \Gamma \cup Q.$$

Definiamo due linguaggi che sono basati su questo nuovo alfabeto Υ.

Il primo linguaggio che definiamo è il **primo linguaggio successore**

$$L'_{\text{succ}} = \{\alpha \# \beta^R \mid \alpha, \beta \text{ configurazioni} \wedge \alpha \vdash \beta\}.$$

Esempio 1.5.1: Prendiamo due configurazioni α e β con $\alpha \vdash \beta$ e β ottenuta dalla funzione di transizione con una mossa che sposta la testina avanti di una posizione.

Questo vuol dire che

$$xqay \vdash xbpy$$

e quindi che dentro L'_{succ} abbiamo la stringa

$$xqay\#y^R pbx^R.$$

In che posizione della gerarchia posizioniamo questo linguaggio? Dobbiamo controllare due aspetti:

- α e β^R devono essere due configurazioni, e questo lo possiamo fare con un banale **controllo a stati finiti** che controlli che ci siano dei simboli di Γ, un solo stato, e altri simboli di Γ;
- $\alpha \vdash \beta$, ma questo si può fare con un **automa a pila**:
  - durante la prima passata carichiamo sulla pila i vari caratteri e capiamo, usando la funzione di transizione, quale parte della configurazione verrà modificata;
  - leggiamo il separatore #;
  - iniziamo a fare il check della seconda configurazione, che, essendo scritta al contrario, è possibile fare con la pila. Inoltre, sapendo quali porzioni sono cambiate della configurazione, riusciamo a fare un controllo preciso.

Ma allora riusciamo a riconoscere L'_{succ} con un **automa a pila**, addirittura **deterministico**, quindi

$$L'_{\text{succ}} \in \text{DCFL} .$$

Il secondo linguaggio che definiamo è il **secondo linguaggio successore**

$$L''_{\text{succ}} = \{\alpha^R \# \beta \mid \alpha, \beta \text{ configurazioni} \wedge \alpha \vdash \beta\}.$$

Anche in questo caso il linguaggio può essere riconosciuto da un automa a pila deterministico quindi

$$L''_{\text{succ}} \in \text{DCFL} .$$

Abbiamo quindi due linguaggi DCFL per i quali riusciamo a costruire due automi a pila deterministici che li riconoscono, conoscendo ovviamente la MdT che definisce la configurazioni. In poche parole, abbiamo un algoritmo che costruisce questi due automi a pila.

Fissiamo ora una MdT M e definiamo il **linguaggio delle computazioni valide**

$$\text{valid}(M) = \left\{ \alpha_1 \# \alpha_2^R \# ... \# \alpha_k^{(R)} \mid \begin{array}{ll} \text{a.} & \forall i \in \{1, ..., k\} \quad \alpha_i \in \Gamma^* Q \Gamma^* \text{ configurazione di } M \\ \text{b.} & \alpha_1 = q_0 w \text{ configurazione iniziale di } M \\ \text{c.} & \alpha_k = xqy \text{ configurazione finale di } M \\ \text{d.} & \forall i \in \{2, ..., k\} \quad \alpha_{i-1} \vdash \alpha_i \end{array} \right\}.$$

In poche parole abbiamo «esteso» i due linguaggi precedenti, formando una **catena di configurazioni** ove la prima è una configurazione iniziale, l'ultima è una configurazione finale e in ogni coppia di configurazioni consecutive la seconda segue la prima. Questa catena di configurazioni le alterna dritte, in **posizione dispari**, e rovesciate, in **posizione pari**.

Come possiamo riconoscere questo linguaggio? Vediamo le singole condizioni:

a. per la **prima condizione** ci basta una **FSM** che controlli che tra ogni coppia di # ci sia un solo stato, e che anche la prima e l'ultima configurazione abbiano un solo stato nella

loro stringa. In poche parole, adattiamo il controllo a stati finiti che avevamo definito per L'_{succ};

b. per la **seconda condizione** ci basta adattare leggermente la **FSM** che abbiamo costruito per la prima condizione;
c. per la **terza condizione** facciamo lo stesso della seconda condizione;
d. per la **quarta condizione** possiamo riciclare il **PDA** che avevamo per L'_{succ} o per L''_{succ} ma solo quando abbiamo due configurazioni e basta, perché quando ne abbiamo di più e controlliamo due configurazioni consecutive usiamo la pila a disposizione ma perdiamo l'informazione per controllare quella ancora successiva.

Quindi sicuramente serve almeno un **LBA** per riconoscere questo linguaggio.

Come possiamo fare? Proviamo a **scomporre** l'ultima condizione in due parti:

- condizione d' che definisce la consecutività di due configurazioni ove la seconda ha un **indice pari** nella sequenza, ovvero
$$\forall i \text{ pari} \quad \alpha_{i-1} \vdash \alpha_i;$$
- condizione d'' che fa lo stesso ma lo fa per tutti gli **indici dispari**, ovvero
$$\forall i \text{ dispari} \quad \alpha_{i-1} \vdash \alpha_i.$$

Queste singole condizioni possono essere verificate con i **DPDA** che abbiamo costruito prima per i linguaggi successore. Definiamo i linguaggi

$$L' = \{(a) \wedge (b) \wedge (c) \wedge (d')\}$$
$$L'' = \{(a) \wedge (b) \wedge (c) \wedge (d'')\}$$

che **rispettano le condizioni** indicate. Abbiamo detto prima che questi due linguaggi sono **DCFL**. Se imponiamo che entrambe le condizioni siano verificate otteniamo il linguaggio delle computazioni valide, ovvero

$$\text{valid}(M) = L' \cap L''.$$

Per le proprietà di (non) chiusura dei DCFL, questa intersezione **non è DCFL**.

1.6. Problemi di decisione

Le MdT sono molto utili come **strumento** per dimostrare che alcuni **problemi di decisione CFL** non possono essere risolti, perché se lo fossero allora potremmo risolvere alcuni **problemi di decisione sulle MdT** che non possono invece essere risolti.

Visto che una MdT ogni tanto può entrare in **loop** esistono molti problemi di decisione che non possiamo purtroppo decidere. Vediamo due famosi problemi di (non) decisione sulle MdT.

Definizione 1.6.1 (Problema dell'arresto o HALT): Il **problema dell'arresto** ha come input una MdT M e una stringa $w \in \Sigma^*$ e si chiede se M termina su input w.

Per risolvere questo problema, visto che una MdT è una macchina universale, possiamo eseguire con una MdT la macchina M che ci viene fornita su input w e vedere se termina, ma questo non lo possiamo sapere: infatti, se la macchina non ci dà risposta è perché sta ancora facendo dei conti o è perché è andata in loop?

Definizione 1.6.2 (Vuotezza): Il **problema della vuotezza** ha come input una MdT M e si chiede se

$$L(M) = \varnothing.$$

Avevamo visto che questo problema era decidibile nelle tipo 3 e nelle tipo 2 grazie al pumping lemma, ma in questo caso non lo possiamo decidere.

Di questi due **problemi indecidibili** useremo praticamente sempre l'ultimo.

1.6.1. Intersezione vuota di DCFL

Riprendiamo il linguaggio valid(M). Abbiamo detto che, data una MdT M, riusciamo a costruire algoritmicamente due automi a pila che riconoscono L' e L''. Abbiamo poi detto che valid(M) è l'intersezione di questi due linguaggi.

Osservando questa intersezione, se la vediamo vuota possiamo concludere che allora il linguaggio valid(M) delle computazioni valide è vuoto, ovvero

$$L' \cap L'' = \varnothing \iff \text{valid}(M) = \varnothing \iff L(M) = \varnothing.$$

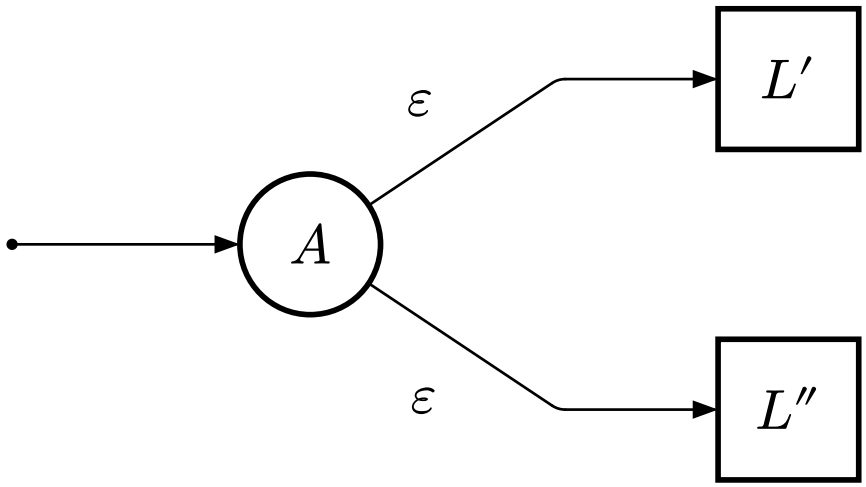
Ma abbiamo detto che questo non lo possiamo decidere in una MdT, quindi non è possibile decidere se l'intersezione di due linguaggi DCFL è vuota.

1.6.2. Linguaggio ambiguo

Con l'intersezione di due DCFL non ci è andata molto bene, quindi ora proviamo con l'**unione**. Dati allora i due linguaggi L' e L'' DCFL di prima definiamo il linguaggio

$$\Xi = L' \cup L''.$$

Per riconoscere questo linguaggio potremmo costruire un riconoscitore A non deterministico che all'inizio usa una ε-mossa per far partire in parallelo i due **DPDA** e vedere se almeno uno dei due riconosce la stringa che viene data in input.



I due linguaggi L' e L'' sono **DCFL**, quindi hanno una sola computazione accettante, ma con A potremmo invece avere **ambiguità** perché abbiamo inserito due ε-mosse e quindi riconoscere la stringa in due modi diversi.

Esistono delle **stringhe ambigue** nel linguaggio Ξ, o in altri termini, il linguaggio Ξ è **ambiguo**?

Vediamo la catena di implicazioni:

$$\Xi \text{ ambiguo} \iff \exists x \mid x \text{ ha due computazioni accettanti in } A \iff$$
$$\iff x \in L' \cap L'' \iff x \in \text{valid}(M) \iff \text{valid}(M) \neq \varnothing.$$

Ma siamo al punto di prima: non potendo decidere la vuotezza, e quindi anche la non vuotezza, non è possibile decidere se un linguaggio CFL è ambiguo o no. Questa proprietà si trasporta quindi di conseguenza anche sui DCFL.