Teoria dei Linguaggi

Indice

1. Lezione 21 [21/05]

1.1. 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 si che ogni automa per L sia ambiguo e quindi che L è non deterministico.

Esempio 1.1.1: 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 $L^C \notin$ CFL allora L non può essere DCFL perché questi ultimi sono chiusi rispetto al complemento, ed essendo $L^C \notin$ CFL allora vale anche $L^C \notin$ DCFL.

Esempio 1.1.2: 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

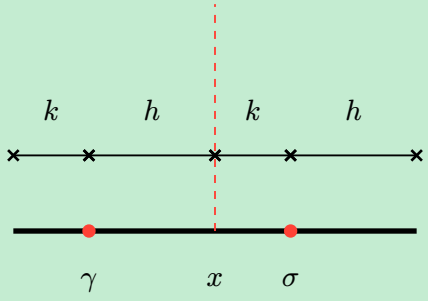
$$L^C = \{ww \mid w \in \{a, b\}^*\}.$$

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.



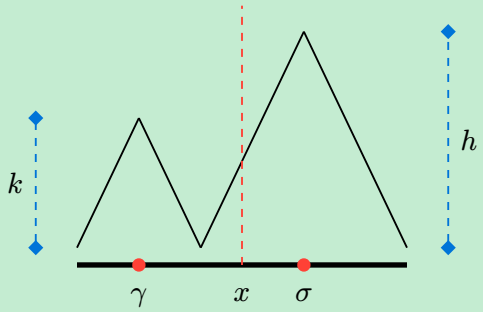
Abbiamo quindi un simbolo γ a distanza k dall'inizio che deve essere diverso da un simbolo σ a distanza $h + k = m$ da γ.

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 1.1.3: Vediamo una grammatica per il linguaggio precedente.

Le regole di produzione sono:

$$S \longrightarrow AB \mid BA \mid A \mid B$$
$$A \longrightarrow aAa \mid aAb \mid bAa \mid bAb \mid a$$
$$B \longrightarrow aBa \mid aBb \mid bBa \mid bBb \mid b.$$

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

$$A \overset{*}{\Rightarrow} xAy \Rightarrow xay \quad | \quad |x| = |y|$$
$$B \overset{*}{\Rightarrow} zBv \Rightarrow zbv \quad | \quad |z| = |v|.$$

Ma allora stiamo generando della stringhe

$$S \Rightarrow AB \overset{*}{\Rightarrow} \underbrace{xa\underline{\ }yz\underline{\ }bv} \quad | \quad |x| + |v| = |y| + |z|.$$

Stesso discorso lo possiamo fare per $S \Rightarrow BA$.

Esempio 1.1.4: 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 1.1.5: 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 1.1.1: 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 1.1.6: Definiamo di nuovo il linguaggio

$$\text{PAL} = \{x \in \{a,b\}^* \mid x = x^R\}.$$

Facciamo vedere che

$$x, y \in \{a,b\}^* \mid x \neq y \Longrightarrow (x,y) \notin R,$$

ovvero che ogni classe di equivalenza è formata da un solo elemento.

Prendiamo quindi due stringhe generiche

$$x = x_1 ... x_n$$
$$y = y_1 ... y_m$$

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

$$\exists i \in \{1, ..., n\} \mid x_i \neq y_i$$

allora scegliamo la stringa $z = x^R$ tale che

$$xz = xx^R \in \text{PAL}$$
$$yz = yx^R = y_1 ... y_m x_n ... x_1 \notin \text{PAL}$$

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

$$\forall i \in \{1, ..., n\} \quad x_i = y_i,$$

sapendo che $x \neq y$ possiamo dire che $m > n$. Possiamo scrivere y come

$$y = x_1 ... x_n y_{n+1} ... y_m.$$

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

$$xz = xcx^R \in \text{PAL}$$
$$yz = xy_{n+1}...y_m cx^R \notin \text{PAL}$$

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 accordato 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.

1.2. Ricorsione

Visto che i linguaggi DCFL ci consentono l'uso della **ricorsione** essi sono utili per definire i **linguaggi di programmazione**. Come gerarchia abbiamo

$$\text{LR}(k) \subseteq \text{DCFL} \subseteq \text{CFL},$$

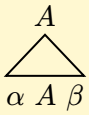
con la classe $\text{LR}(k)$ che indica degli oggetti molto tecnici, poco naturali, che sono usati nei **parser**. Se $k = 1$ allora stiamo considerando direttamente i DCFL.

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 1.2.1 (Grammatiche self-embedding): Prendiamo una grammatica $G = (V, \Sigma, P, S)$ context-free. Diciamo che G è **self-embedding** se

$$\exists A \in V \mid A \overset{*}{\Rightarrow} \alpha A \beta \mid \alpha, \beta \in (\Sigma \cup V)^+.$$

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 1.2.1: 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 1.2.1.1: Se L è un linguaggio CFL e non regolare allora ogni G per L è self-embedding.

1.3. Linguaggio di Dyck

Per finire, vediamo un risultato che secondo me è veramente fuori di testa.

Definizione 1.3.1 (Linguaggio di Dyck): Definiamo l'alfabeto

$$\Omega_k = \{(_1, (_2, ...(_k,)_1,)_2, ...,)_k\}$$

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 1.3.1 (Teorema di Chomsky-Schutzenberger): Dato $L \subseteq \Sigma^*$ un CFL, allora:

- $\exists k > 0$ numero intero,
- $\exists$morfismo $h : \Omega_k \longrightarrow \Sigma^*$,
- $\exists R \subseteq \Omega_k^*$ 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 1.3.1: 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

$$(_1 \longrightarrow a \qquad\qquad)_1 \longrightarrow b$$

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 1.3.2: Se prendiamo L il linguaggio delle parentesi bilanciate, allora scegliamo l'identità come morfismo e come linguaggio regolare quello che fa passare tutto.

Esempio 1.3.3: 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

$$(_1 \longrightarrow a \qquad)_1 \longrightarrow a \qquad (_2 \longrightarrow b \qquad)_2 \longrightarrow b$$

Come espressione regolare ci ispiriamo a quella di prima, quindi scegliamo

$$R = [(_1 + (_2]^*[)_1 +)_2]^*.$$

Esempio 1.3.4: Definiamo infine il linguaggio PAL delle stringhe palindrome di lunghezza anche dispari. Qua dobbiamo modificare leggermente la soluzione precedente

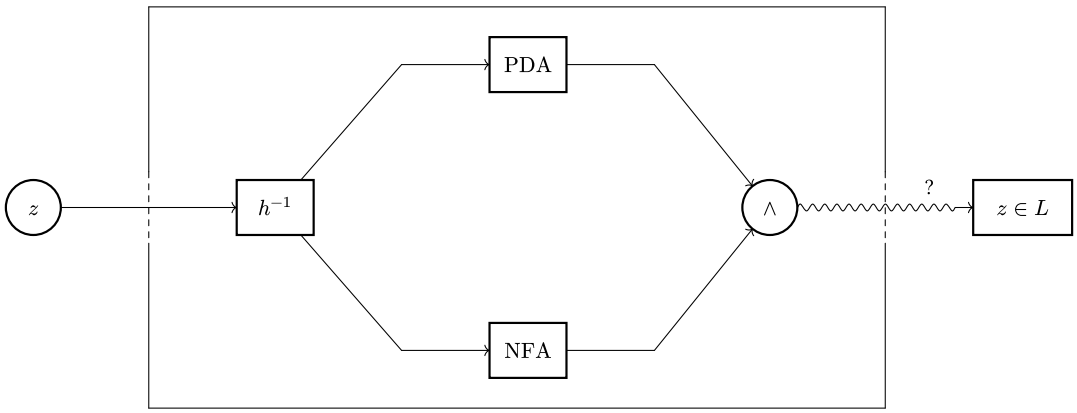
Scegliamo $k = 4$ usando le parentesi definite prima, alle quali aggiungiamo due parentesi, che usiamo per codificare l'eventuale simbolo centrale, che può essere una a o una b, quindi:

$$(_3 \longrightarrow a \qquad)_3 \longrightarrow \varepsilon \qquad (_4 \longrightarrow b \qquad)_4 \longrightarrow \varepsilon$$

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

$$R = [(_1 + (_2]^*[\varepsilon + (_3)_3 + (_4)_4][)_1 +)_2]^*.$$

Se non abbiamo a disposizione un riconoscitore per L, ma conosciamo tutto ciò che serve per costruirlo con il Teorema 1.3.1, 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.

2. Lezione 22 [23/05]

In questa lezione parleremo di troppe cose: toccheremo tutta la gerarchia di Chomsky, esclusi i linguaggi di tipo 0, considerando alfabeti particolari e macchine riconoscitrici diverse dal solito. Andremo poi avanti anche con le grammatiche di tipo 2.

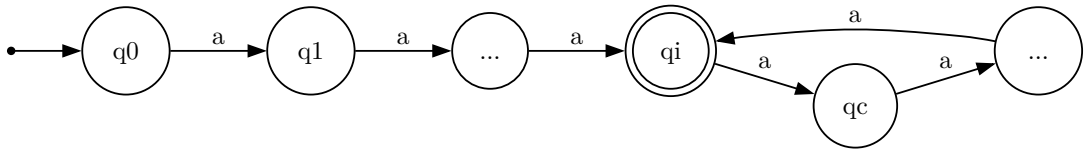
2.1. Alfabeti unari

Partiamo con gli **alfabeti unari**: questi sono alfabeti molto particolari formati da un solo carattere, ovvero sono nella forma

$$\Sigma = \{a\}.$$

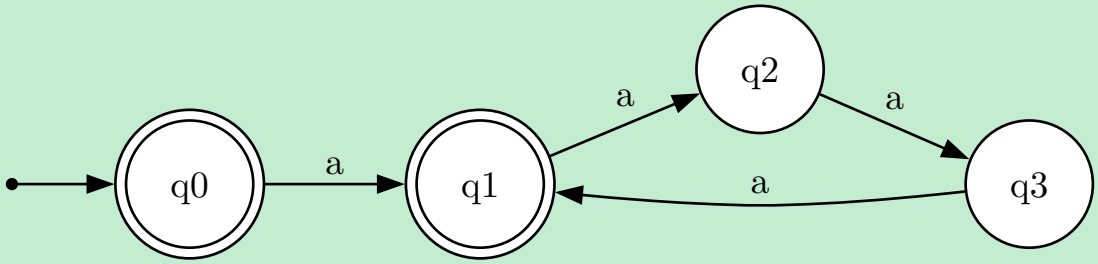
2.1.1. Linguaggi regolari

Riprendiamo in mano, dopo tanto tempo, gli **automi a stati finiti**. Se rimaniamo nel caso deterministico, da ogni stato di un **DFA** può uscire un solo arco con una certa etichetta, ovvero non posso avere più di 2 archi uscenti con la stessa etichetta. Avendo ora un solo carattere in Σ quello che abbiamo è una sequenza (opzionale) di stati che prima o poi sfocia in un **ciclo** (opzionale).



Notiamo come l'informazione sulle parole diventa **informazione sulla lunghezza** di esse, visto che possiamo riconoscere delle stringhe che seguono un certo pattern di lunghezze.

Esempio 2.1.1.1: Vediamo un esempio di automa a stati finiti unario.



Con questo automa riconosciamo ε, a e poi quest'ultima a cui aggiungiamo un numero di a uguali alla lunghezza del ciclo, ovvero

$$L = \{\varepsilon\} \cup \{a^{1+3k} \mid k \geq 0\} = \varepsilon + a(a^3)^*.$$

Dal punto di vista matematico, possiamo vedere questi automi come delle **successioni numeriche/aritmetiche**, ovvero delle successioni che hanno una parte iniziale e poi un periodo che viene ripetuto.

Nel caso di **NFA** invece abbiamo un grafo arbitrario, che per essere trasformato in DFA richiede meno dei 2^n classici della **costruzione per sottoinsiemi**, ovvero ci costa

$$e^{n \ln(n)}.$$

Come vediamo, è una quantità **subesponenziale** ma comunque **superpolinomiale**. Inoltre, questo bound non può essere migliorato, è la soluzione ottimale.

Esempio 2.1.1.2: Definiamo tre linguaggi

$$L_1 = a^{28}(a^3)^* \qquad L_2 = a^{11}(a^3)^* \qquad L_3 = a^{37}(a^3)^*$$

Cosa aggiungono questi linguaggi al linguaggio L dell'Esempio 2.1.1.1?

Se consideriamo L_1 notiamo che a^{28} può essere anche riconosciuto facendo uno step con una a e poi facendo 9 cicli da 3, quindi riusciamo a riconoscerlo anche con L. Possiamo fare un discorso praticamente simile con L_3. Ma allora questi due linguaggi non aggiungono niente.

Considerando invece L_2 questo aggiunge qualcosa ad L perché riusciamo a riconoscere la stringa $a^{10} = a(a^3)^3$ con L ma poi rimane una a fuori, che ci manda in q_2 e quindi ci fa accettare di più, o comunque qualcosa di diverso rispetto a L.

Avremmo aggiunto altre informazioni considerando un linguaggio

$$L = a^k(a^3)^* \mid k \bmod 3 = 0.$$

Notiamo che, fissato un periodo, non possiamo unire tanti linguaggi, ma solo quelli che rimangono all'interno delle **classi di resto del periodo**.

2.1.2. Equivalenza tra linguaggi regolari e CFL

Vediamo ora come si comportano i CFL. Sia L un **CFL unario**, ovvero

$$L \subseteq a^*.$$

Applichiamo il **pumping lemma** a questo linguaggio. Prendiamo N la **costante del pumping lemma** per i CF per L. Questo ci dice che

$$\forall z \in L \mid |z| \geq N$$

noi possiamo decomporre z come $z = uvwxy$ con:

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

Le stringhe di L sono formate da sole a, quindi se scambiamo dei fattori nella stringa non lo notiamo. Modifichiamo l'ultima condizione del pumping lemma con

$$\forall i \geq 0 \quad uwy(vx)^i \in L.$$

Mettendo insieme le prime due condizioni possiamo dire che

$$1 \leq |vx| \leq N.$$

La stringa z la possiamo dividere in una **parte fissa** e in una **parte pompabile**, ovvero

$$|z| = |uwy| + |vx| = s_z + t_z.$$

Grazie alla terza condizione sappiamo che

$$\forall i \geq 0 \quad a^{s_z}(a^{t_z})^i \in L \Longrightarrow a^{s_z}(a^{t_z})^* \in L.$$

Possiamo fare un'ulteriore divisione, stavolta sulle stringhe di L: infatti, possiamo scrivere L come unione di due insiemi L' e L'' tali che

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

Nell'insieme L' mettiamo tutte le stringhe che non fanno parte del pumping lemma, ovvero

$$L' = \{z \in L \mid |z| < N\}.$$

Nell'insieme L'' mettiamo invece tutte le stringhe pompate, ovvero

$$L'' = \{z \in L \mid |z| \geq N\} \subseteq \bigcup_{z \in L''} a^{s_z}(a^{t_z})^*.$$

Analizziamo separatamente i due insiemi:

- L' è un linguaggio **finito**, quindi lo possiamo riconoscere con un automa a stati finiti;
- L'' invece sembra un'unione infinita, ma abbiamo visto che il periodo t_z del pumping lemma è boundato con le classi di resto, ovvero

  $$1 \leq t_z \leq N,$$

  quindi questo linguaggio, che è unione finita di linguaggi regolari, è anch'esso **finito**.

Ma allora il linguaggio L è **regolare**.

Teorema 2.1.2.1: Sia $L \subseteq a^*$ un CFL. Allora L è regolare.

Questo va d'accordo con quello che abbiamo fatto la lezione scorsa: i CFL hanno la **ricorsione**, ma se abbiamo un solo carattere non possiamo aprire e chiudere le parentesi, quindi collassiamo nei linguaggi regolari.

2.1.3. Teorema di Parikh

Vediamo, per finire, una serie di concetti un po' strani e che non dimostreremo.

Definizione 2.1.3.1 (Immagine di Parikh sulle stringhe): Sia $\Sigma = \{\sigma_1, ..., \sigma_n\}$ un alfabeto. L'**immagine di Parikh** sulle stringhe è la funzione

$$\psi : \Sigma^* \longrightarrow \mathbb{N}^{|\Sigma|}$$

tale che

$$\psi(x) = \big(\#_{\sigma_1}(x), ..., \#_{\sigma_n}(x)\big).$$

In poche parole, questa funzione conta le **occorrenze** di ogni lettera di Σ dentro la stringa x.

Esempio 2.1.3.1: Definiamo $\Sigma = \{a, b\}$. Data $z = aababa$, calcoliamo

$$\psi(z) = (4, 2).$$

Con l'immagine di Parikh sulle stringhe possiamo definire un insieme di queste immagini.

Definizione 2.1.3.2 (Immagine di Parikh): Dato L un linguaggio generico, l'**immagine di Parikh** è l'insieme

$$\psi(L) = \{\psi(x) \mid x \in L\}.$$

In poche parole, l'immagine di Parikh è l'insieme di tutte le immagini di Parikh sulle stringhe di L.

Esempio 2.1.3.2: Vediamo tre linguaggi e le loro immagini di Parikh associate.

Linguaggio	**Immagine di Parikh**
$L = \{a^n b^n \mid n \geq 0\}$	$\{(n, n) \mid n \geq 0\}$
$L = a^* b^*$	$\{(i, j) \mid i, j \geq 0\}$
$L = (ab)^*$	$\{(n, n) \mid n \geq 0\}$

Notiamo come il primo e il terzo insieme sono uguali, anche se vengono generati da due linguaggi gerarchicamente diversi: il primo è un tipo 2, il terzo è un tipo 3.

L'ultima osservazione fatta genera quello che è il **teorema di Parikh**.

Teorema 2.1.3.1 (Teorema di Parikh): Se L è un CFL allora $\exists R$ regolare tale che

$$\psi(L) = \psi(R).$$

In poche parole, se non ci interessa l'**ordine** con cui scriviamo i caratteri di una stringa, allora i **linguaggi regolari** e i **CFL** sono la stessa cosa, collassano nella stessa classe.

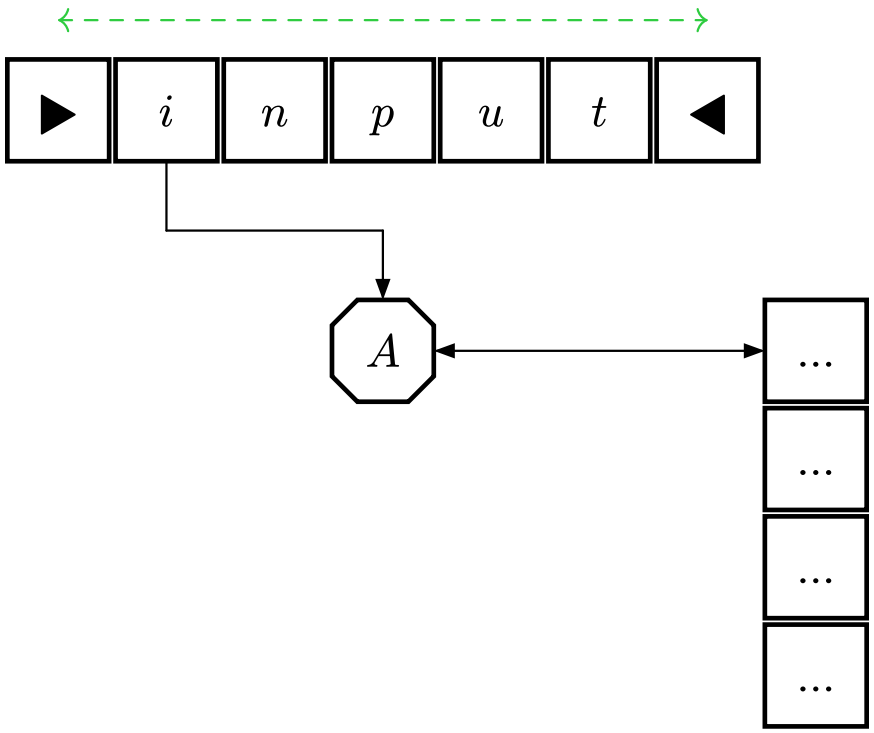
2.2. Automi a pila two-way

Modifichiamo un po' la macchina che stiamo usando da forse troppo tempo.

2.2.1. Definizione

Negli automi a stati finiti, il movimento **two-way** non aumentava la potenza computazionale del modello. Ma cosa succede negli **automi a pila two-way**?

Vediamo prima di tutto una rappresentazione del modello.



Come nei 2DFA, mettiamo degli **end marker** per marcare i bordi della stringa, perché ora la nostra testina di lettura può andare **avanti e indietro** sul nastro.

Con questo modello possiamo fare **molto di più** dei classici automi a pila.

2.2.2. Esempi

Vediamo una serie di linguaggi non CFL che riusciamo a riconoscere con questo modello.

Esempio 2.2.2.1: Definiamo il linguaggio

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

Questo linguaggio non è CFL perché una volta che controlliamo le b con le a perdiamo l'informazione su n. Con un **2DPDA** possiamo controllare le a con le b, poi tornare all'inizio delle b e controllare le b con le c.

Esempio 2.2.2.2: Definiamo il linguaggio

$$L = \{a^{2^n} \mid n \geq 0\}.$$

Con il **pumping lemma** avevamo mostrato che questo linguaggio non è CFL. Ora che abbiamo la definizione di sequenza algebrica, possiamo dire che questo linguaggio non è una **sequenza algebrica** perché le a si allontanano sempre di più tra loro.

Dobbiamo controllare se l'input è una potenza di 2: per fare ciò continuiamo a dividere per 2, verificando di avere sempre resto zero, salvo alla fine, dove abbiamo per forza resto 1.

Se k è la lunghezza dell'input, possiamo eseguire i seguenti passi:

1. leggiamo l'input per intero, e ogni due a carichiamo una lettera sulla pila, caricando in totale $\frac{k}{2}$ caratteri. Con questa passata controlliamo se le a sono pari o dispari;
2. svuotiamo la pila, spostandoci di una posizione a sinistra ogni volta che togliamo un carattere. Con questa mezza passata ci troviamo, appunto, a metà della stringa, sul carattere in posizione $\frac{k}{2}$;
3. ricominciamo dal primo punto fino a quando non rimaniamo con un carattere solo, che mi dà per forza resto 1.

Anche questo, come quello di prima, è un **2DPDA**.

Esempio 2.2.2.3: Definiamo il linguaggio

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

Anche questo linguaggio non è CFL, e lo avevamo mostrato con uno dei quattro criteri della scorsa lezione, non mi ricordo quale in questo momento.

Come prima, carichiamo nella pila un carattere ogni due caratteri letti dell'input completo. Con questa prima passata controlliamo anche se il numero di caratteri è pari o dispari, e in quest'ultimo caso ci fermiamo e rifiutiamo. Spostiamoci poi in mezzo alla stringa scaricando la pila fino al carattere iniziale che avevamo anche prima.

Chiamiamo w la parte a sinistra della posizione nella quale ci troviamo ora. Carichiamo w dal centro verso l'inizio: stiamo leggendo w^R, che caricata sulla pila diventa w.

Spostiamoci di nuovo a metà della stringa, mettendo un separatore # tra w e i caratteri che usiamo per spostarci. Ora che siamo a metà, togliamo # dal congelatore e, con w sulla pila, possiamo controllare se la seconda parte è uguale a w.

Anche questo è un fantastico **2DPDA**.

Esempio 2.2.2.4: Non lo facciamo vedere, ma il linguaggio

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

ha un **2DPDA** che lo riconosce in maniera molto simile a quelle precedenti.

Come vediamo, questo modello è **molto potente**, talmente potente che nessuno sa quanto sia potente: infatti, tutti gli esempi visti sono stati risolti con un **2DPDA**, quindi anche da un **2NPDA** che fa partire una sola computazione alla volta, ma non sappiamo se

$$\text{2NPDA} \stackrel{?}{=} \text{2DPDA} .$$

Inoltre, non si conosce la relazione che si ha con i linguaggi di tipo 1, che vediamo tra poco, ovvero non sappiamo se

$$\text{2DPDA} \stackrel{?}{=} \text{CS} .$$

2.3. Problemi di decisione dei CFL

Per finire questa lezione infinita, torniamo indietro ai linguaggi CFL e vediamo qualche **problema di decisione**. Per ora vedremo i problemi a cui sappiamo rispondere con quello che sappiamo, questo perché dei problemi di decisione richiedono conoscenze delle **macchine di Turing**, che per ora non abbiamo.

2.3.1. Appartenenza

Dato L un CFL e una stringa $x \in \Sigma^*$, ci chiediamo se $x \in L$.

Questo è molto facile: sappiamo che i CFL sono **decidibili** perché lo avevamo mostrato per i linguaggi di tipo 1. Come complessità come siamo messi?

Sia $n = |x|$. Esistono algoritmi semplici che permettono di decidere in tempo

$$T(n) = O(n^3).$$

L'**algoritmo di Valiant**, quasi incomprensibile, riconduce il problema di riconoscimento a quello di prodotto tra matrici $n \times n$, che con l'algoritmo di Strassen possiamo risolvere in tempo

$$T(n) = O\big(n^{\log_2(7)}\big) = O(n^{2.81\dots}).$$

L'algoritmo di Strassen in realtà poi è stato superato da altri algoritmi ben più sofisticati, che impiegano tempo quasi quadratico, ovvero

$$T(n) = O(n^{2.3\dots}).$$

Una domanda aperta si chiede se riusciamo ad abbassare questo bound al livello quadratico, e questo sarebbe molto comodo: infatti, negli algoritmi di parsing avere degli algoritmi quadratici è apprezzabile, e infatti spesso di considerano sottoclassi per avvicinarsi a complessità lineari.

2.3.2. Linguaggio vuoto e infinito

Sia L un CFL, ci chiediamo se $L \neq \varnothing$ oppure se $|L| = \infty$.

Vediamo un teorema praticamente identico a uno che avevamo già visto.

Teorema 2.3.2.1: Sia $L \subseteq \Sigma^*$ un CFL, e sia N la costante del pumping lemma per L. Allora:

1. $L \neq \varnothing \iff \exists z \in L \mid |z| < N$;
2. $|L| = \infty \iff \exists z \in L \mid N \leq |z| < 2N$.

Gli algoritmi per verificare la non vuotezza o l'infinità non sono molto efficienti: infatti, prima di tutto bisogna trovare N, e se ho una grammatica è facile (basta passare in tempo lineare per la FN di Chomsky), ma se non ce l'abbiamo è un po' una palla. Poi dobbiamo provare tutte le stringhe fino alla costante, che sono 2^N, e con questo rispondiamo alla non vuotezza. Per l'infinità è ancora peggio.

Si possono implementare delle tecniche che lavorano sul **grafo delle produzioni**, ma sono molto avanzate e (penso) difficili da utilizzare.

2.3.3. Universalità

Dato L un CFL, vogliamo sapere se $L = \Sigma^*$, ovvero vogliamo sapere se siamo in grado di generare tutte le stringhe su un certo alfabeto.

Nei linguaggi regolari passavamo per il complemento per vedere se il linguaggio era vuoto, ma nei CFL **non abbiamo il complemento**, quindi non lo possiamo utilizzare.

Infatti, questo problema **non si può decidere**: non esistono algoritmi che stabiliscono se un PDA riesce a riconoscere tutte le stringhe, o se una grammatica riesce a generare tutte le stringhe.