

Teoria dei Linguaggi

Indice

Introduzione	7
--------------------	---

Parte I — Gerarchia di Chomsky 8

1. Breve ripasso	9
2. Gerarchia di Chomsky	12
2.1. Grammatiche	12
2.2. Gerarchia di Chomsky	15
2.3. Decidibilità	17
2.4. Introduzione della parola vuota	19
2.5. Linguaggi non esprimibili tramite grammatiche finite	20

Parte II — Linguaggi regolari 22

1. Automi a stati finiti deterministici	23
1.1. Definizione	23
1.2. Esempi	24
2. Automi a stati finiti non deterministici	27
2.1. Definizione	27
2.2. Confronto tra DFA e NFA	28
2.3. Forme di non determinismo	29
3. Numero minimo di stati	32
3.1. Distinguibilità	32
3.2. Applicazioni del concetto di distinguibilità	33
3.3. Automa di Meyer-Fischer	39
3.4. Fooling set	43
3.5. Applicazioni del concetto di fooling set	44
4. Equivalenza tra linguaggi di tipo 3 e automi a stati finiti	46
4.1. Dall'automa alla grammatica	46
4.2. Dalla grammatica all'automa	46
4.3. Grammatiche lineari	47
4.3.1. Grammatiche lineari a destra	48
4.3.2. Grammatiche lineari a sinistra	48
4.3.3. Grammatiche lineari	48
5. Automa minimo	50
5.1. Introduzione matematica	50
5.2. Relazione R_M	52
5.3. Relazione R_L	54
5.4. Teorema di Myhill-Nerode	55
5.5. Automa minimo	57
5.6. Applicazioni agli NFA	57
6. Operazioni tra linguaggi	59
6.1. Operazioni insiemistiche	59
6.2. Operazioni tipiche	59

7. Espressioni regolari	62
7.1. Teorema di Kleene	62
7.2. Da automa ad espressione regolare	63
7.3. State complexity	65
7.4. Da espressione regolare ad automa	66
7.4.1. Espressioni base	66
7.4.2. Complemento	67
7.4.2.1. DFA	67
7.4.2.2. NFA	67
7.4.2.3. Costruzione per sottoinsiemi	69
7.4.3. Unione	69
7.4.3.1. DFA	70
7.4.3.2. Automa prodotto	71
7.4.3.3. NFA	72
7.4.4. Intersezione	72
7.4.5. Prodotto	73
7.4.5.1. DFA	73
7.4.5.2. Costruzione senza nome	74
7.4.5.3. NFA	74
7.4.6. Chiusura di Kleene	74
7.4.6.1. DFA	75
7.4.6.2. NFA	76
7.5. Codici	76
7.6. Star height	77
7.7. Espressioni regolari estese	79
8. Operazioni esotiche e avanzate	82
8.1. Reversal	82
8.1.1. DFA	82
8.1.2. NFA	83
8.1.3. Reversal di una grammatica	83
8.1.4. Algoritmo per l'automa minimo	85
8.2. Shuffle	88
8.2.1. Alfabeti disgiunti	89
8.2.2. Stesso alfabeto	90
8.2.3. Alfabeto unario	90
8.3. Raddoppio	90
8.4. Metà	91
8.5. Automi come matrici	92
8.5.1. Prima applicazione: raddoppio	93
8.5.2. Seconda applicazione: metà	93
8.5.3. Matrici su NFA	94
9. Pumping Lemma	96
9.1. Definizione	96
9.2. Applicazioni	98
10. Problemi di decisione per i linguaggi regolari	100
10.1. Linguaggio vuoto e infinito	100
10.2. Appartenenza	101

10.3. Universalità	101
10.4. Inclusione e uguaglianza	101
11. Automi two-way	102
11.1. Varianti di automi a stati finiti	102
11.1.1. Automi pesati e probabilistici	102
11.2. Varianti pesanti di automi a stati finiti	102
11.2.1. One-way VS two-way	103
11.2.2. Read-only VS read-write	103
11.2.3. Memoria esterna	104
11.3. Automi two-way	104
11.3.1. Esempi vari	105
11.3.2. Definizione formale	107
11.3.3. Potenza computazionale	107
11.3.4. Problema di Sakoda & Sipser	111

Parte III — Linguaggi context-free **114**

1. Automi a pila	115
1.1. Definizione	115
1.2. Accettazione	118
1.3. Determinismo VS non determinismo	120
1.4. Trasformazioni	121
1.5. Esempi	122
2. Equivalenza tra CFL e grammatiche di tipo 2	124
2.1. Ripasso e introduzione	124
2.2. Da grammatica di tipo 2 ad automa a pila	126
2.3. Da automa a pila a grammatica di tipo 2	129
2.3.1. Forma normale per gli automi a pila	129
2.3.2. Dimostrazione	132
3. Forme normali per le grammatiche di tipo 2	135
3.1. FN di Greibach	135
3.1.1. Definizione	135
3.1.2. Esempio	136
3.2. Forma normale di Chomsky	137
3.2.1. Definizione	137
3.2.2. Costruzione	137
3.2.2.1. Eliminazione delle ϵ -produzioni	138
3.2.2.2. Eliminazione delle produzioni unitarie	139
3.2.2.3. Eliminazione dei simboli inutili	140
3.2.2.4. Eliminazione dei terminali	140
3.2.2.5. Smontaggio delle produzioni	141
3.2.3. Esempio	141
4. Pumping lemma	143
4.1. Prerequisiti per il pumping lemma	143
4.2. Pumping lemma per i CFL	146
4.3. Applicazioni del pumping lemma	151

5. Lemma di Ogden	155
5.1. Fail del pumping lemma	155
5.2. Lemma di Ogden	156
5.3. Applicazioni	159
6. Ambiguità	161
6.1. Esempi	161
6.2. Definizione	161
6.3. Ambiguità e non determinismo	166
7. Operazioni tra linguaggi	169
7.1. Operazioni insiemistiche	169
7.1.1. Unione	169
7.1.1.1. CFL	169
7.1.1.2. DCFL	169
7.1.2. Intersezione	170
7.1.2.1. CFL	170
7.1.2.2. DCFL	170
7.1.3. Intersezione con un regolare	170
7.1.3.1. CFL	170
7.1.3.2. DCFL	170
7.1.4. Complemento	170
7.1.4.1. CFL	170
7.1.4.2. DCFL	171
7.1.5. Riassunto	174
7.2. Operazioni regolari	174
7.2.1. Prodotto	174
7.2.1.1. CFL	174
7.2.1.2. DCFL	174
7.2.2. Star	175
7.2.2.1. CFL	175
7.2.2.2. DCFL	175
7.2.3. Riassunto	175
7.3. Considerazioni	176
8. CFL VS DCFL	177
8.1. Pumping lemma	177
8.2. Linguaggio inerentemente ambiguo	177
8.3. Proprietà di chiusura	177
8.4. Relazione di Myhill-Nerode	180
9. Risultati particolari	182
9.1. Ricorsione	182
9.2. Linguaggio di Dyck	182
10. Alfabeti unari	186
10.1. Linguaggi regolari	186
10.2. Equivalenza tra linguaggi regolari e CFL	187
10.3. Teorema di Parikh	188
11. Automi a pila two-way	190
11.1. Definizione	190

11.2. Esempi	190
12. Problemi di decisione	193
12.1. Appartenenza	193
12.2. Linguaggio vuoto e infinito	193
12.3. Universalità	193
12.4. Altri problemi	194
 Parte IV — Linguaggi context-sensitive	 195
1. Automi limitati linearmente	196
1.1. Definizione	196
1.2. Classi di complessità associate	196
1.3. Esempi	197
2. Equivalenza tra LBA e grammatiche di tipo 1	200
2.1. Dimostrazione	200
2.2. Determinismo e non determinismo	200
3. Operazioni tra linguaggi	201
 Parte V — Linguaggi senza restrizioni	 202
1. Macchine di Turing	203
1.1. Introduzione	203
1.2. Varianti di MdT	204
1.2.1. Nastro semi-infinito	204
1.2.2. Nastri multipli	205
1.2.3. Nastri dedicati	206
1.2.4. Testine multiple	206
1.2.5. Automi a pila doppia	207
1.3. Determinismo VS non determinismo	208
1.4. Definizione formale	209
2. Problemi di decisione	213
2.1. Definizioni preliminari	213
2.2. Problemi di decisione	215
2.2.1. Intersezione vuota di DCFL	216
2.2.2. Linguaggio ambiguo	216
2.2.3. Universalità	217
2.2.4. Equivalenza	217
2.2.5. Contenimento	217
2.2.6. Regolarità	217
2.2.7. Regolarità ma peggio	219
2.2.8. Regolarità ma ancora peggio	219
2.3. Riassunto	220

Introduzione

In queste dispense vengono presentati i **sistemi formali** che descrivono i linguaggi: ci chiederemo cosa sono in grado di fare, ovvero cosa **descrivono** in termini di **linguaggi**.

Ci occuperemo anche delle **risorse utilizzate**, come il **numero di mosse** eseguite da una macchina che deve riconoscere un linguaggio, oppure il **numero di stati** che sono necessari per descrivere una macchina a stati finiti, oppure ancora lo **spazio utilizzato** da una macchina di Turing.

Un **linguaggio** è «*uno strumento di comunicazione usato da membri di una stessa comunità*», ed è composto da due elementi:

- **sintassi**: insieme di simboli (*o parole*) che devono essere combinati con una serie di regole;
- **semantica**: associazione frase-significato.

Per i linguaggi naturali è difficile dare delle regole sintattiche: vista questa difficoltà, nel 1956 **Noam Chomsky** introduce il concetto di **grammatiche formali**, che si servono di regole matematiche per la definizione della sintassi di un linguaggio.

Il primo utilizzo dei linguaggi formali risale agli stessi anni con il **compilatore Fortran**. Anche se ci hanno messo l'equivalente di 18 anni/uomo, questa è la prima applicazione dei linguaggi formali. Con l'avvento, negli anni successivi, dei linguaggi Algol, ovvero linguaggi con strutture di controllo, la teoria dei linguaggi formali è diventata sempre più importante.

Oggi la teoria dei linguaggi formali è usata nei compilatori di compilatori, dei tool usati per generare dei compilatori per un dato linguaggio fornendo la descrizione di quest'ultimo.

Parte I — Gerarchia di Chomsky

1. Breve ripasso

Prima di addentrarci nello studio della gerarchia di Chomsky facciamo un breve **ripasso** delle basi che ci serviranno durante lo studio dei linguaggi formali.

Partiamo proprio dalle basi, quindi prima di tutto diamo la definizione di **alfabeto**.

Definizione 1.1 (*Alfabeto*): Un **alfabeto** è un **insieme non vuoto e finito di simboli**, di solito indicato con le lettere greche maiuscole

$$\Sigma \quad | \quad \Gamma.$$

Dato un alfabeto, sopra di esso ci possiamo costruire delle **stringhe** con varie proprietà.

Definizione 1.2 (*Stringa*): Una **stringa**, o **parola**, è una **sequenza finita** di simboli appartenenti all'alfabeto Σ . Viene indicata con la lettera x e la possiamo scrivere come

$$x = a_1 \dots a_n \quad | \quad a_i \in \Sigma.$$

Definizione 1.3 (*Lunghezza di una stringa*): Data una stringa x , indichiamo con

$$|x|$$

la sua **lunghezza**, ovvero il **numero di caratteri** contenuti in x .

Definizione 1.4 (*Numero di occorrenze*): Data una stringa x e un carattere a , indichiamo con

$$|x|_a \quad \text{oppure} \quad \#_a(x)$$

il **numero di occorrenze** del carattere a in x .

Una stringa/parola molto importante è la **parola vuota**, che possiamo indicare in vari modi:

$$\varepsilon \quad | \quad \lambda \quad | \quad \Lambda.$$

Come dice il nome, questa parola non ha simboli, ovvero è l'unica parola tale che

$$|\varepsilon| = 0.$$

Dato un alfabeto Σ , l'insieme di tutte le possibili parole che possiamo formare si indica con Σ^* . Questo insieme, ovviamente, è un **insieme infinito**, visto che possiamo concatenare infinite volte i caratteri presenti nell'alfabeto dato.

Con le parole possiamo definire una serie di **operazioni**, ma la più importante è la **concatenazione**, o **prodotto**. Date due stringhe

$$x, y \in \Sigma^* \quad | \quad x = x_1 \dots x_n \wedge y = y_1 \dots y_m$$

allora la concatenazione di x e y è la stringa

$$w = x \cdot y = x_1 \dots x_n y_1 \dots y_m.$$

L'operazione di concatenazione **non è commutativa** ma è **associativa**, quindi la struttura

$$(\Sigma^*, \cdot, \varepsilon)$$

è un **monoide libero** generato da Σ .

Vediamo, per (quasi) finire, alcune proprietà che possiamo dare alle stringhe/parole.

Definizione 1.5 (*Prefisso*): La stringa $x \in \Sigma^*$ si dice **prefisso** di w se

$$\exists y \in \Sigma^* \mid w = xy.$$

In poche parole, x è prefisso di w se riusciamo a scomporre la stringa w in due parti, dove x è la prima di queste due. Abbiamo due tipi di prefisso:

- **proprio** se $y \neq \varepsilon$;
- **non banale** se $x \neq \varepsilon$.

Il **numero** di prefissi di una stringa w è $|w| + 1$.

Definizione 1.6 (*Suffisso*): La stringa $y \in \Sigma^*$ si dice **suffisso** di w se

$$\exists x \in \Sigma^* \mid w = xy.$$

In poche parole, vale quanto scritto prima, ma in questo caso y è la seconda delle due parti. Anche qui abbiamo tipi di suffisso:

- **proprio** se $x \neq \varepsilon$;
- **non banale** se $y \neq \varepsilon$.

Anche il **numero** di suffissi di una stringa w è $|w| + 1$.

Definizione 1.7 (*Fattore*): La stringa $y \in \Sigma^*$ si dice **fattore** di w se

$$\exists x, z \in \Sigma^* \mid w = xyz.$$

In poche parole, vale quanto scritto prima, ma in questo caso dividiamo la stringa w in tre parti e y è la centrale di queste.

Il **numero** di fattori di una stringa w è

$$\leq \frac{|w|(|w| + 1)}{2} + 1$$

per via dei possibili doppioni che possiamo trovare.

Definizione 1.8 (*Sottosequenza*): La stringa $x \in \Sigma^*$ si dice **sottosequenza** di w se x è ottenuta eliminando 0 o più caratteri da w . L'eliminazione può avvenire in maniera non contigua: posso eliminare qualsiasi carattere, ma la stringa risultante che leggiamo deve contenere i caratteri nello stesso ordine di partenza.

Possiamo dire che un **fattore** è una **sottosequenza contigua**.

Per finire veramente, diamo forse la definizione più importante, quella di **linguaggio**.

Definizione 1.9 (*Linguaggio*): Un **linguaggio** L , definito su un alfabeto Σ , è un qualunque sottoinsieme di Σ^* , ovvero

$$L \subseteq \Sigma^*.$$

Ora che abbiamo fatto un ripasso siamo pronti per vedere la gerarchia di Chomsky nella sua interezza.

2. Gerarchia di Chomsky

Vogliamo cercare di rappresentare in maniera **finita** un oggetto potenzialmente **infinito**, come ad esempio un **linguaggio**. Per fare ciò, abbiamo a nostra disposizione due modelli potenti:

- il **modello generativo** fornisce delle regole che, applicate da un certo punto di partenza, generano tutte le parole di un linguaggio;
- il **modello riconoscitivo** utilizza un modello di calcolo che prende in input una parola e ci dice se appartiene o meno al linguaggio.

Esempio 2.1: Consideriamo il linguaggio sull'alfabeto $\Sigma = \{ (,) \}$ delle parole ben bilanciate.

Un **modello generativo** per questo linguaggio deve applicare delle regole a partire da una **sorgente** S per derivare tutte le parole del linguaggio. Le regole potrebbero essere:

- la parola vuota ε è ben bilanciata;
- se x è ben bilanciata, allora anche (x) è ben bilanciata;
- se x e y sono ben bilanciate, allora anche xy è ben bilanciata.

Un **modello riconoscitivo** per questo linguaggio è una black-box che, presa una parola, ci dice se essa appartiene o meno al linguaggio. In realtà questa macchina potrebbe **non terminare** mai, ma ne parleremo più in fondo in questo capitolo. Una macchina per questo linguaggio deve verificare i seguenti fatti:

- il numero di parentesi aperte è uguale al numero di parentesi chiuse, quindi

$$\#_((x) = \#_)(x);$$

- considerato ogni prefisso, il numero di parentesi aperte non deve superare il numero di parentesi chiuse, quindi

$$\forall y \in \Sigma^* \mid x = yz \quad \#_((y) \leq \#_)(y).$$

2.1. Grammatiche

Le **grammatiche** sono un modello generativo molto potente: vediamo come sono definite.

Definizione 2.1.1 (*Grammatica*): Una **grammatica** è una quadrupla (V, Σ, P, S) definita da:

- V **insieme finito e non vuoto di variabili**. Sono anche dette **simboli non terminali** (o meta-simboli) e sono usate durante il processo di generazione delle parole del linguaggio;
- Σ **insieme finito e non vuoto di simboli terminali**. Sono chiamati così perché appaiono nelle parole generate, a differenza delle variabili che invece non possono essere presenti;
- P **insieme finito e non vuoto di regole di produzione**;
- $S \in V$ **simbolo iniziale o assioma**, il punto di partenza della generazione.

Soffermiamoci brevemente sulle **regole di produzione**: esse sono nella forma

$$\alpha \longrightarrow \beta \mid \alpha \in (V \cup \Sigma)^+ \wedge \beta \in (V \cup \Sigma)^*.$$

La notazione $()^+$ è praticamente l'insieme $()^*$ senza la parola vuota.

Una **regola di produzione** viene letta come «se ho α allora posso sostituirlo con β ».

L'applicazione delle regole di produzione è alla base del **processo di derivazione**: esso è formato infatti da una serie di **passi di derivazione**, che permettono di generare una parola del linguaggio.

Definizione 2.1.2 (*Derivazione in un passo*): Date le stringhe $x, y \in (V \cup \Sigma)^*$ diciamo che x **deriva y in un passo**, e si indica con

$$x \Rightarrow y$$

se e solo se

$$\exists(\alpha \rightarrow \beta) \in P \quad \exists \eta, \delta \in (V \cup \Sigma)^* \mid x = \eta\alpha\delta \wedge y = \eta\beta\delta.$$

Possiamo estendere questa definizione ad un numero definito o arbitrario di passi.

Definizione 2.1.3 (*Derivazione in k passi*): Date le stringhe $x, y \in (V \cup \Sigma)^*$ diciamo che x **deriva y in $k \geq 0$ passi**, e si indica con

$$x \xRightarrow{k} y$$

se e solo se

$$\exists x_0, \dots, x_k \in (V \cup \Sigma)^* \mid x = x_0 \wedge y = x_k \wedge (\forall i \in \{1, \dots, k\} \quad x_{i-1} \Rightarrow x_i).$$

Con questa definizione, se contiamo $k = 0$ andiamo a derivare x da sé stessa, ma questo caso lo usiamo solo per comodità, non ha una vera e propria applicazione pratica.

Quando **non abbiamo indicazioni** sul numero di passi possiamo:

- usare la notazione

$$x \xRightarrow{*} y$$

per indicare un processo di derivazione che avviene in un **numero generico di passi**, e questo vale se e solo se

$$\exists k \geq 0 \mid x \xRightarrow{k} y;$$

- usare la notazione

$$x \xRightarrow{+} y$$

per indicare un processo di derivazione che avviene in **almeno un passo**, e questo vale se e solo se

$$\exists k > 0 \mid x \xRightarrow{k} y.$$

Definizione 2.1.4 (*Linguaggio generato da una grammatica*): Il **linguaggio generato** dalla grammatica G è l'insieme

$$L(G) = \{w \in \Sigma^* \mid S \xRightarrow{*} w\}.$$

In poche parole, $L(G)$ è l'insieme di tutte le stringhe w che si possono ottenere in un certo numero di passi di derivazione a partire dall'assioma S della grammatica. Notiamo che le stringhe che otteniamo sono formate dai soli **caratteri terminali**. Le stringhe intermedie che utilizziamo invece nei vari passi di derivazione sono dette **forme sentenziali**.

Definizione 2.1.5 (*Grammatiche equivalenti*): Due grammatiche G_1 e G_2 sono **equivalenti** se e solo se

$$L(G_1) = L(G_2).$$

Vediamo qualche grammatica come esempio.

Esempio 2.1.1: Riprendiamo il linguaggio delle parentesi tonde ben bilanciate.

Possiamo definire una grammatica che ha le seguenti regole di produzione:

$$S \longrightarrow \varepsilon \qquad S \longrightarrow (S) \qquad S \longrightarrow SS$$

Esempio 2.1.2: Sia $G = (\{S, A, B\}, \{a, b\}, P, S)$ una grammatica con le seguenti regole di produzione:

$$S \longrightarrow aB \mid bA \qquad A \longrightarrow a \mid aS \mid bAA \qquad B \longrightarrow b \mid bS \mid aBB$$

Questa grammatica genera il linguaggio

$$L(G) = \{w \in \Sigma^* \mid \#_a(w) = \#_b(w)\}.$$

Infatti, ogni forma sentenziale w che generiamo è tale che

$$\#_{\{a,A\}}(w) = \#_{\{b,B\}}(w)$$

e questa relazione viene poi mantenuta trasformando tutte le A e le B nei relativi simboli terminali a e b .

Esempio 2.1.3: Definiamo ora la grammatica $G = (\{S, A, B, C, D, E\}, \{a, b\}, P, S)$ che contiene le seguenti regole di produzione:

$$\begin{array}{lll} S \rightarrow ABC & AB \rightarrow \varepsilon \mid aAD \mid bAE & DC \rightarrow BaC \\ EC \rightarrow BbC & Da \rightarrow aD & Db \rightarrow bD \\ Ea \rightarrow aE & Eb \rightarrow bE & C \rightarrow \varepsilon \\ aB \rightarrow Ba & bB \rightarrow Bb & \end{array}$$

Generando qualche parola ci si accorge che questa grammatica genera il **linguaggio pappagallo**, definito come

$$L(G) = \{ww \mid w \in \Sigma^*\}.$$

2.2. Gerarchia di Chomsky

Negli anni '50 **Noam Chomsky** studia la generazione dei linguaggi formali e crea una **gerarchia di grammatiche formali** che si basa sulla forma delle **regole di produzione** che definiscono la grammatica.

Grammatica	Regole	Modello riconoscitivo
Tipo 0	Nessuna restrizione	Macchine di Turing
Tipo 1 , dette context-sensitive (o dipendenti dal contesto)	<p>Se $(\alpha \rightarrow \beta) \in P$ allora</p> $ \alpha \leq \beta ,$ <p>ovvero devo generare parole che non sono più corte di quella di partenza</p> <p>Sono dette dipendenti dal contesto perché ogni regola $(A \rightarrow B) \in P$ può essere scritta come</p> $\alpha_1 A \alpha_2 \rightarrow \alpha_1 B \alpha_2,$ <p>con $\alpha_1, \alpha_2 \in (V \cup \Sigma)^*$ che rappresentano il contesto, $A \in V$ e $B \in (V \cup \Sigma)^+$</p>	Automi limitati linearmente
Tipo 2 , dette context-free (o libere dal contesto)	Le regole in P sono del tipo $\alpha \rightarrow \beta$, con $\alpha \in V$ e $\beta \in (V \cup \Sigma)^+$	Automi a pila

Tipo 3 , dette grammatiche regolari	Le regole in P sono del tipo $A \rightarrow aB$ oppure $A \rightarrow a$, con $A, B \in V$ e $a \in \Sigma$	Automi a stati finiti
---	--	------------------------------

La gerarchia che ha definito Chomsky è **propria**, ovvero:

$$L_3 \subset L_2 \subset L_1 \subset L_0.$$

Come vedremo alla fine di questo capitolo, questa gerarchia **non esaurisce** tutti i linguaggi possibili: esistono infatti linguaggi che non sono descrivibili in maniera finita con le grammatiche.



Definizione 2.2.1 (*Tipo di una grammatica*): Sia $L \subseteq \Sigma^*$ un linguaggio. Allora L è di tipo i se e solo se esiste una grammatica G di tipo i tale che

$$L = L(G).$$

La gerarchia data considera dei **modelli deterministici**, ma come cambia considerando invece dei **modelli non deterministici**? Sappiamo che:

- le grammatiche di tipo 3 mantengono la stessa potenza computazionale, pagando un costo in termini di descrizione, quindi in **numero di stati**;
- le grammatiche di tipo 2 hanno il modello non deterministico strettamente più potente;
- le grammatiche di tipo 1 sono abbastanza complicate;
- le grammatiche di tipo 0, come quelle regolari, mantengono la stessa potenza computazionale.

Il non determinismo comunque è una nozione del **riconoscitore** che sto usando:

- nel determinismo il riconoscitore può fare una scelta alla volta;
- nel non determinismo può fare più scelte contemporaneamente.

Nelle grammatiche è difficile catturare questa nozione, perché esse lo hanno **intrinsecamente**, perché le derivazioni le applico tutte assieme per ottenere le stringhe del linguaggio.

2.3. Decidibilità

Se una grammatica è di tipo 1 allora possiamo costruire una macchina che sia in grado di dire, in tempo finito, se una parola appartiene o meno al linguaggio generato da quella grammatica. Questa macchina è detta **verificatore**, e si dice che le grammatiche di tipo 1 sono **decidibili**.

Teorema 2.3.1 (*Decidibilità dei linguaggi context-sensitive*): I linguaggi di tipo 1 sono **ricorsivi**.

Con **ricorsività** non intendiamo le procedure ricorsive, ma si intende una procedura che è calcolabile automaticamente. Nei linguaggi, un qualcosa di ricorsivo intende una macchina che, data una stringa x in input, riesce a rispondere a $x \in L$ terminando sempre dicendo **SI** o **NO**. Si usano i termini **ricorsivo** e **decidibile** come sinonimi.

Dimostrazione 2.3.1.1: In una grammatica di tipo 1 l'unico vincolo è sulla lunghezza delle produzioni, ovvero non possono mai accorciarsi.

In input ho una stringa $w \in \Sigma^*$ la cui lunghezza è $|w| = n$. Ho una grammatica G di tipo 1 e mi chiedo se $w \in L(G)$. Per rispondere a questo, devo cercare w nelle forme sentenziali, ma possiamo limitarci a quelle che non superano la lunghezza n : infatti, visto che la lunghezza aumenta sempre (o al massimo rimane uguale) posso arrivare al massimo alle stringhe di lunghezza n e controllare solo quelle.

Definiamo quindi gli insiemi

$$T_i = \left\{ \gamma \in (V \cup \Sigma)^{\leq n} \mid S \xRightarrow{\leq i} \gamma \right\} \quad \forall i \geq 0.$$

Calcoliamo induttivamente questi insiemi.

Se $i = 0$ non eseguo nessuna derivazione, quindi

$$T_0 = \{S\}.$$

Supponiamo di aver calcolato T_{i-1} . Ma allora

$$T_i = T_{i-1} \cup \{ \gamma \in (V \cup \Sigma)^{\leq n} \mid \exists \beta \in T_{i-1} \mid \beta \Rightarrow \gamma \}.$$

Noi partendo da T_0 calcoliamo tutti i vari insiemi ottenendo una serie di T_i . Per come abbiamo definito gli insiemi, sappiamo che

$$T_0 \subseteq T_1 \subseteq T_2 \subseteq \dots \subseteq (V \cup \Sigma)^{\leq n}$$

e l'ultima inclusione è vera perché ho fissato la lunghezza massima, non voglio considerare di più perché vogliamo w di lunghezza n .

La grandezza dell'insieme $(V \cup \Sigma)^{\leq n}$ è finita, quindi anche andando molto avanti con le computazioni prima o poi arrivo ad un certo punto dove non posso più aggiungere niente, ovvero vale che

$$\exists i \in \mathbb{N} \mid T_i = T_{i-1}.$$

Ora è inutile andare avanti, questo T_i è l'insieme di tutte le stringhe che riesco a generare nella grammatica. Ora mi chiedo se $w \in T_i$, che posso fare molto facilmente scorrendo l'insieme.

Ma allora G è decidibile. ■

Ci rendiamo conto che questa soluzione è **altamente inefficiente**: infatti, in tempo polinomiale non riusciamo a fare questo nelle tipo 1, ma è una soluzione che ci garantisce la decidibilità.

In che situazione si trovano invece i linguaggi di tipo 0?

Teorema 2.3.2 (*Semi-decidibilità dei linguaggi di tipo 0*): I linguaggi di tipo 0 sono **ricorsivamente enumerabili**.

Dimostrazione 2.3.2.1: In una grammatica di tipo 0 non abbiamo vincoli da considerare.

In input ho una stringa $w \in \Sigma^*$ la cui lunghezza è $|w| = n$. Ho una grammatica G di tipo 0 e, come prima, mi chiedo se $w \in L(G)$. Per rispondere a questo, devo cercare w nelle forme sentenziali, ma a differenza di prima non possiamo limitarci a quelle che non superano la lunghezza n : infatti, visto che le forme sentenziali si possono accorciare posso anche superare di molto la lunghezza n e poi sperare di tornare indietro in qualche modo.

Definiamo quindi gli insiemi

$$U_i = \left\{ \gamma \in (V \cup \Sigma)^* \mid S \xRightarrow{\leq i} \gamma \right\} \quad \forall i \geq 0.$$

Calcoliamo induttivamente questi insiemi.

Se $i = 0$ non eseguo nessuna derivazione, quindi

$$U_0 = \{S\}.$$

Supponiamo di aver calcolato U_{i-1} . Vogliamo calcolare

$$U_i = U_{i-1} \cup \{ \gamma \in (V \cup \Sigma)^* \mid \exists \beta \in U_{i-1} \mid \beta \Rightarrow \gamma \}.$$

Noi partendo da U_0 calcoliamo tutti i vari insiemi ottenendo una serie di U_i . Per come abbiamo definito gli insiemi, sappiamo che

$$U_0 \subseteq U_1 \subseteq U_2 \subseteq \dots \subseteq (V \cup \Sigma)^*.$$

A differenza di prima, la grandezza dell'insieme $(V \cup \Sigma)^*$ è infinita, quindi non ho più l'obbligo di stopparmi ad un certo punto per esaurimento delle stringhe generabili.

Come rispondiamo a $w \in L(G)$? Iniziamo a costruire i vari insiemi U_i e ogni volta che termino la costruzione mi chiedo se $w \in U_i$: se questo è vero allora rispondo **SI**, in caso contrario vado avanti con la costruzione.

Vista la cardinalità infinita dell'insieme che fa da container, potrei andare avanti all'infinito (*a meno di ottenere due insiemi consecutivi identici, in tale caso rispondo NO*).

Ma allora G è semi-decidibile. ■

Usiamo la dicitura **ricorsivamente enumerabile** perché ogni volta che costruisco un insieme U_i posso prendere le stringhe $w \in \Sigma^*$ appena generate ed elencarle, quindi **enumerarle** una per una.

2.4. Introduzione della parola vuota

Introduciamo il **problema della parola vuota**. Dalle grammatiche di tipo 1 a salire abbiamo il vincolo di non poter scendere di lunghezza con le derivazioni, quindi diciamo che la parola vuota ε non la vedremo mai come lato destro di una derivazione. Eppure, ogni tanto la parola vuota dovrebbe appartenere al linguaggio generato da una grammatica. Come possiamo risolvere questo problema, senza far crollare l'intera gerarchia?

Una possibile soluzione è **spezzare le regole di produzione**.

Partiamo da una grammatica $G = (V, \Sigma, P, S)$ di tipo 1 e definiamo una nuova grammatica

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

tale che $L(G) = L(G')$. Vediamo le componenti di questa grammatica:

- l'**insieme delle variabili** contiene un nuovo elemento, il **nuovo assioma** S' , ovvero

$$V' = V \cup \{S'\};$$

- l'**insieme delle produzioni** mantiene le regole vecchie ma ne aggiunge due nuove, ovvero

$$P' = P \cup \{S' \rightarrow \varepsilon\} \cup \{S' \rightarrow S\}$$

dove:

- la prima regola permette di generare la parola vuota ε ;
- la seconda regola permette di far partire la computazione della vecchia grammatica;
- l'**assioma** S' ci permette la generazione della parola vuota ma dobbiamo garantire che non appaia mai nel lato destro delle produzioni.

Con questi accorgimenti ora riusciamo a generare anche la **parola vuota**: infatti, questo lo possiamo fare all'inizio partendo da S' . Se non ci interessa la parola vuota facciamo partire, sempre da S' , la computazione della vecchia grammatica.

Come cambia la gerarchia considerando anche la parola vuota? Abbiamo che:

- le grammatiche di tipo 1 mantengono la clausola $|\alpha| \leq |\beta|$ ma è possibile ottenere ε da S' purché S' non appaia mai nel lato destro delle produzioni;
- le grammatiche di tipo 2 modificano la notazione $()^+$ in $()^*$ nel lato destro delle produzioni senza isolare in modo specifico ε perché questo non crea problemi;
- le grammatiche di tipo 3 seguono le precedenti.

Queste particolari produzioni che considerano la parola vuota sono dette **ε -produzioni**.

2.5. Linguaggi non esprimibili tramite grammatiche finite

Vediamo infine dei linguaggi che non possiamo esprimere tramite **grammatiche finite**. Per fare ciò useremo la famosissima **dimostrazione per diagonalizzazione** di Cantor.

Esempio 2.5.1: Sono più i numeri pari o i numeri dispari? Sono più i numeri pari o i numeri interi? Sono più le coppie di numeri naturali o i naturali stessi?

Per rispondere a queste domande si usa la definizione di **cardinalità**, e tutti questi insiemi che abbiamo citato ce l'hanno uguale. Anzi, diciamo di più: tutti questi insiemi sono grandi quanto i naturali, perché esistono funzioni biettive tra questi insiemi e l'insieme \mathbb{N} .

Esempio 2.5.2: Sono più i sottoinsiemi di naturali o i naturali stessi?

In questo caso, sono di più i sottoinsiemi, che hanno la **cardinalità del continuo**. Per dimostrare questo useremo una dimostrazione per diagonalizzazione.

Teorema 2.5.1: Vale

$$\mathbb{N} \sim 2^{\mathbb{N}}.$$

Dimostrazione 2.5.1.1: Per assurdo sia $\mathbb{N} \sim 2^{\mathbb{N}}$, ovvero ogni elemento di $2^{\mathbb{N}}$ è **listabile**.

Creiamo una tabella booleana M indicizzata sulle righe dai sottoinsiemi di naturali S_i e indicizzata sulle colonne dai numeri naturali. Per ogni insieme S_i abbiamo sulla riga la funzione caratteristica, ovvero

$$M[i, j] = \begin{cases} 1 & \text{se } j \in S_i \\ 0 & \text{se } j \notin S_i \end{cases}.$$

Creiamo l'insieme

$$S = \{x \in \mathbb{N} \mid x \notin S_x\},$$

ovvero l'insieme che prende tutti gli elementi 0 della diagonale di M . Questo insieme non è presente negli insiemi S_i listati perché esso è diverso da ogni S_i in almeno una posizione, ovvero la diagonale.

Abbiamo ottenuto un assurdo, ma allora $\mathbb{N} \not\sim 2^{\mathbb{N}}$. ■

Siamo quasi pronti per l'ultima dimostrazione di questo capitolo.

Esempio 2.5.3: Sono più le stringhe o i numeri naturali?

Questi insiemi hanno la stessa cardinalità perché possiamo trasformare ogni stringa data in un numero naturale usando una qualche codifica. Con questa nozione e tutte quelle precedenti siamo pronti per dimostrare un teorema molto importante.

Teorema 2.5.2: Esistono linguaggi che non sono descrivibili da grammatiche finite.

Dimostrazione 2.5.2.1: Prendiamo una grammatica $G = (V, \Sigma, P, S)$.

Per descriverla devo dire come sono formati i vari campi della tupla. Cosa uso per descriverla? Sto usando dei simboli come lettere, numeri, parentesi, eccetera, quindi la grammatica è una descrizione che possiamo fare sotto forma di stringa. Visto quello che abbiamo da poco «dimostrato», ogni grammatica la possiamo descrivere come una stringa, e quindi come un numero intero. Siano G_i tutte queste grammatiche, che sono appunto listabili.

Consideriamo ora per ogni grammatica G_i l'insieme $L(G_i)$ delle parole generate dalla grammatica G_i , ovvero il linguaggio generato da G_i . Mettiamo dentro L tutti questi linguaggi.

Per assurdo, siano tutti questi linguaggi listabili, ovvero $\mathbb{N} \sim L$.

Come prima, creiamo una tabella M indicizzata sulle righe dai linguaggi $L(G_i)$ e indicizzata sulle colonne dalle stringhe x_i che possiamo però considerare come naturali. La matrice M ha sulla riga i -esima la funzione caratteristica di $L(G_i)$, ovvero

$$M[i, j] = \begin{cases} 1 & \text{se } x_j \in L(G_i) \\ 0 & \text{se } x_j \notin L(G_i) \end{cases}.$$

In poche parole, abbiamo 1 nella cella $M[i, j]$ se e solo se la stringa x_j viene generata da G_i .

Costruiamo ora l'insieme

$$LG = \{x_i \in \mathbb{N} \mid x_i \notin L(G_i)\},$$

ovvero l'insieme di tutte le stringhe x_i che non sono generate dalla grammatica G_i con lo stesso indice i . Come prima, questo insieme non è presente in L perché differisce da ogni insieme presente in almeno una posizione, ovvero quello sulla diagonale.

Siamo ad un assurdo, ma allora $\mathbb{N} \not\sim L$. ■

Parte II — Linguaggi regolari

1. Automi a stati finiti deterministici

Nel contesto delle grammatiche di tipo 3 andiamo ad utilizzare le **macchine a stati finiti** per stabilire se, data una stringa x , essa appartiene o meno ad un dato linguaggio. Le macchine a stati finiti da ora le chiameremo anche **FSM** (*Finite State Machine*) o, nel caso delle macchine deterministiche, **DFA** (*Deterministic Finite Automata*).

1.1. Definizione

Un **DFA** A è un dispositivo formato da un **nastro**, che contiene l'input x da esaminare disposto carattere per carattere uno per cella del nastro da sinistra verso destra. Abbiamo anche una **testina** read-only che punta alle celle del nastro e un **controllo a stati finiti**. Il numero di stati, come si capisce, sono in **numero finito**, e soprattutto sono **fissati**, ovvero non dipendono dalla grandezza dell'input. Infine, il modello base che usiamo per ora è quello delle FSM **one-way**, ovvero quello che usa una testina che va sinistra verso destra senza poter tornare indietro.



All'accensione della macchina il controllo si trova nello **stato iniziale** q_0 con la testina sul primo carattere dell'input. Ad ogni passo della computazione la testina legge un carattere e, in base a questo e allo stato corrente, calcola lo stato prossimo. Questo spostamento avviene grazie alla **funzione di transizione**, che vedremo dopo. Arrivati alla fine dell'input grazie alla funzione di transizione, la macchina deve rispondere **SI** o **NO**.

Definizione 1.1.1 (DFA): Un **DFA** è una **quintupla**

$$A = (Q, \Sigma, \delta, q_0, F)$$

formata da:

- Q **insieme finito** di stati;
- Σ **alfabeto** di input;
- δ **funzione di transizione**;
- $q_0 \in Q$ **stato iniziale**;
- $F \subseteq Q$ **insiemi degli stati finali**.

La funzione di transizione, che non abbiamo ancora definito formalmente, è il **programma dell'automa**, il **motore** che manda avanti la macchina. Essa è una funzione

$$\delta : Q \times \Sigma \longrightarrow Q$$

che, dati il simbolo letto dalla testina e lo stato corrente, mi dice in che stato muovermi.

La funzione di transizione spesso è comodo scriverla in **forma tabellare**, con le righe indicizzate dagli stati, le colonne indicizzate dai simboli e nelle celle inseriamo gli stati prossimi.

Può essere comodo anche **disegnare** l'automa. Esso è un **grafo orientato**, con i **vertici** che rappresentano gli stati e gli **archi** che rappresentano le transizioni. Gli archi sono **etichettati** dai simboli di Σ che causano una certa transizione. Lo **stato iniziale** è indicato con una freccia che arriva dal nulla, mentre gli **stati finali** sono indicati con un doppio cerchio o con una freccia che va nel nulla, ma quest'ultima convenzione è **francese** quindi va automaticamente scartata.

Dobbiamo modificare leggermente la funzione di transizione: a noi piacerebbe averla definita sulle **stringhe** e non sui caratteri. Definiamo quindi l'**estensione** di δ come la funzione

$$\delta^* : Q \times \Sigma^* \longrightarrow Q$$

definita induttivamente come

$$\begin{aligned}\delta^*(q, \varepsilon) &= q \\ \delta^*(q, xa) &= \delta(\delta^*(q, x), a) \mid x \in \Sigma^* \wedge a \in \Sigma.\end{aligned}$$

Per non avere in giro troppo nomi usiamo δ^* con il nome δ anche per le stringhe, è la stessa cosa.

Noi **accettiamo** se finiamo in uno stato finale. Il **linguaggio accettato** da A è l'insieme

$$L(A) = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}.$$

1.2. Esempi

Vediamo un po' di esempi che ci permettono di introdurre anche una serie di situazioni particolari.

Diamo subito una distinzione dei problemi che abbiamo sugli automi:

- se abbiamo in mano un automa e dobbiamo descrivere il linguaggio che riconosce, siamo davanti ad un **problema di analisi**;
- se abbiamo in mano un linguaggio e dobbiamo scrivere un automa che lo riconosce, siamo davanti ad un **problema di sintesi**.

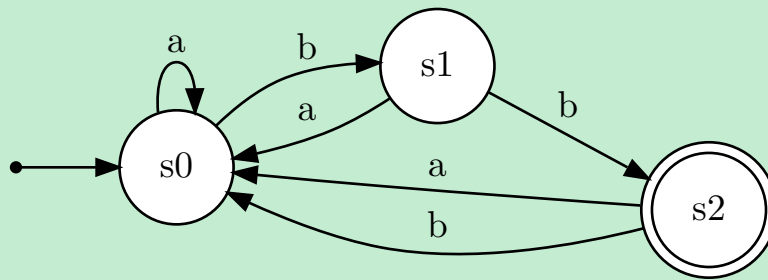
Esempio 1.2.1: Sia $A = (Q, \Sigma, \delta, q_0, F)$ tale che:

- $Q = \{s_0, s_1, s_2\}$;
- $\Sigma = \{a, b\}$;
- $q_0 = s_0$;
- $F = s_2$.

Diamo una **rappresentazione tabellare** della funzione di transizione δ . Essa è

$$\begin{array}{c|cc} & a & b \\ \hline s_0 & s_0 & s_1 \\ s_1 & s_0 & s_2 \\ s_2 & s_0 & s_0 \end{array}.$$

Disegniamo anche l'automa A avendo a disposizione la rappresentazione di δ .



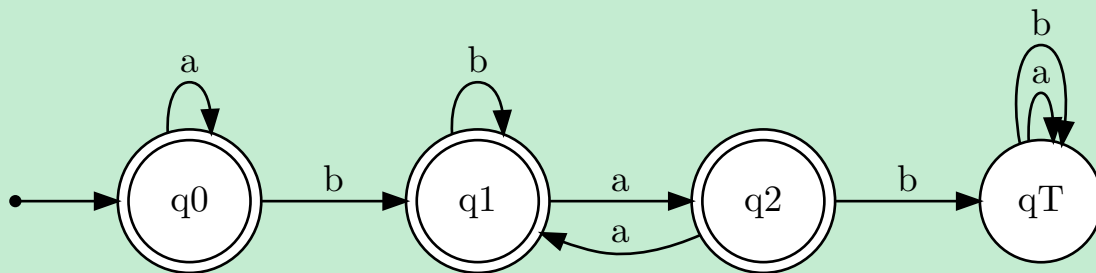
Il linguaggio che riconosce questo automa è

$$L = \{x \in \Sigma^* \mid \text{il più lungo suffisso di } x \text{ formato solo da } b \text{ è lungo } 3k + 2 \mid k \geq 0\}.$$

Esempio 1.2.2: Sia $\Sigma = \{a, b\}$, vogliamo trovare un automa per il linguaggio

$$L = \{x \in \Sigma^* \mid \text{tra ogni coppia di } b \text{ successive vi è un numero di } a \text{ pari}\}.$$

Costruiamo un automa deterministico per L .



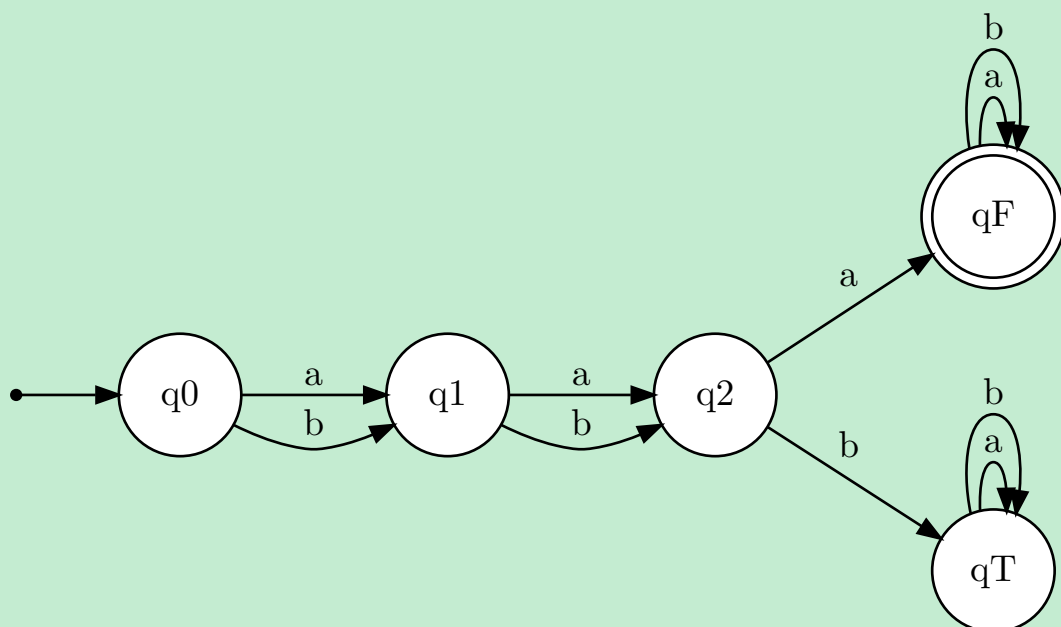
Come vediamo dall'esempio precedente, abbiamo uno stato particolare q_T che è detto **stato trappola**: esso viene utilizzato come «*punto di arrivo*» per esaurire la lettura dell'input e non accettare la stringa data in input. Finiamo in questo stato se, in uno stato q , leggiamo un carattere che rende la stringa non presente in L .

Lo stato trappola è **opzionale**: per semplicità, quando un automa **non è completo**, ovvero uno stato non ha un arco per un carattere, si assume che quell'arco vada a finire in uno stato trappola. Questa semplificazione permette di disegnare automi molto più **compatti**.

Esempio 1.2.3: Sia $\Sigma = \{a, b\}$, vogliamo trovare un automa per il linguaggio

$$L = \{x \in \Sigma^* \mid \text{il terzo simbolo di } x \text{ è una } a\}.$$

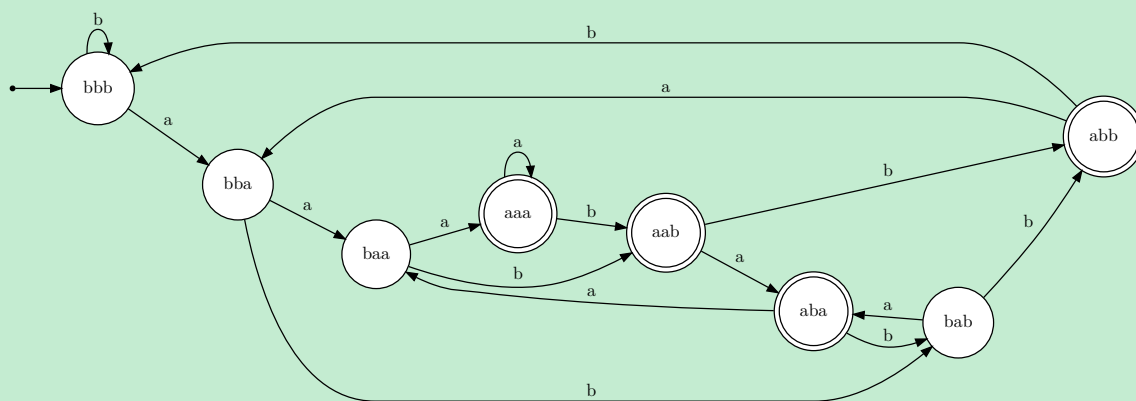
Costruiamo un automa deterministico per L .



Esempio 1.2.4: Sia $\Sigma = \{a, b\}$, vogliamo ora trovare un automa per il linguaggio

$$L = \{x \in \Sigma^* \mid \text{il terzo simbolo di } x \text{ da destra è una } a\}.$$

Costruiamo un automa deterministico per L . Qua l'idea è ricordarsi una finestra di 3 simboli e grazie a questa vediamo se il primo carattere che definisce lo stato è una a .



Ci servono per forza 8 stati o possiamo fare meglio? Abbiamo trovato la strada migliore? Quando introdurremo il concetto di **distinguibilità** nel [Capitolo 3](#) vedremo che questo è l'automa migliore che possiamo costruire per questo linguaggio.

2. Automi a stati finiti non deterministici

L'automa proposto nell'esempio [Esempio 1.2.4](#) del capitolo precedente ci ha richiesto 2^n stati. Abbiamo poi detto che con la nozione di **distinguibilità** dimostreremo che non ci sono DFA con meno stati di quello che abbiamo costruito. Ma se invece utilizzassimo degli **automi non deterministici**?

Esempio 2.1: Vediamo un automa non deterministico per il linguaggio appena discusso.



Abbiamo usato un numero di stati uguale a $n + 1$ (escluso quello trappola), dove n è la posizione da destra del carattere richiesto.

2.1. Definizione

Un **automa non deterministico** è un oggetto molto particolare. Analizzando l'[Esempio 2.1](#), notiamo che dallo stato q_0 , leggendo una a , noi abbiamo la possibilità di scegliere se restare in q_0 o andare in q_1 , ovvero abbiamo più scelte di transizioni in uno stesso stato. Che significato diamo a questo? Noi non sappiamo a che punto siamo della stringa, quindi usiamo il non determinismo come una **scommessa**: scommetto che, quando sono in q_0 , io sia nel terzultimo carattere, e che quindi riuscirò a finire nello stato q_3 . Ovviamente, con questa nuova nozione di «**parallelismo**» che si va a creare dobbiamo anche modificare alcune componenti dell'automa.

Gli **automi non deterministici**, o **NFA**, sono definiti ancora dalla **quintupla**

$$A = (Q, \Sigma, \delta, q_0, F)$$

che differisce da quella dei DFA solo nella **funzione di transizione**. Essa è la funzione

$$\delta : Q \times \Sigma \longrightarrow 2^Q$$

che, dati lo stato corrente e il carattere letto dalla testina, mi manda in un insieme di stati possibili.

Prima di definire formalmente l'accettazione di una stringa da parte di un automa non deterministico, definiamo l'**estensione** di δ come la funzione

$$\delta^* : Q \times \Sigma^* \longrightarrow 2^Q$$

definita induttivamente come

$$\begin{aligned} \delta^*(q, \varepsilon) &= \{q\} \\ \delta^*(q, xa) &= \bigcup_{p \in \delta^*(q, x)} \delta(p, a) \mid x \in \Sigma^* \wedge a \in \Sigma. \end{aligned}$$

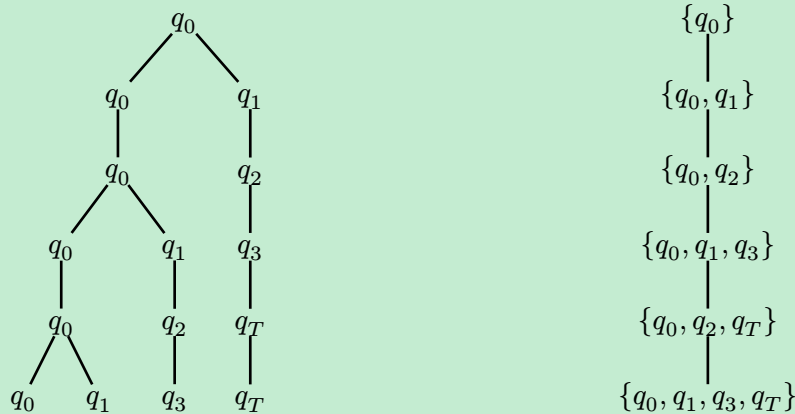
Come prima, per non avere in giro troppo nomi, usiamo δ^* con il nome δ anche per le stringhe.

Quando **accettiamo** una stringa? Avendo teoricamente la possibilità di fare **infinite computazioni parallele**, visto che ad ogni passo posso sdoppiare la mia computazione, ci basta avere **almeno** un percorso che finisce in uno stato finale.

Il **linguaggio riconosciuto** dall'automa A non deterministico è

$$L(A) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \emptyset\}.$$

Esempio 2.1.1: Considerando l'automa dell'Esempio 2.1, scriviamo l'albero di computazione che viene generato mentre cerca di riconoscere la stringa $x = ababa$.



Visto che raggiungiamo, all'ultimo livello dell'albero, almeno una volta lo stato finale q_3 , la stringa x viene accettata dall'automa.

2.2. Confronto tra DFA e NFA

Banalmente, ogni automa deterministico è anche un automa non deterministico nel quale abbiamo, per ogni stato, al massimo un arco uscente etichettato con lo stesso carattere. In poche parole, abbiamo sempre una sola scelta. Ma allora la classe dei linguaggi riconosciuti da DFA è inclusa nella classe dei linguaggi riconosciuti da NFA.

Ma vale anche il viceversa: ogni automa non deterministico può essere trasformato in un automa deterministico con una costruzione particolare, detta **costruzione per sottoinsiemi**.

Dato $A = (Q, \Sigma, \delta, q_0, F)$ un NFA, e costruisco

$$A' = (Q', \Sigma, \delta', q_0', F')$$

un DFA tale che:

- $Q' = 2^Q$, ovvero gli **stati** sono tutti i possibili sottoinsiemi;
- $\delta' : Q' \times \Sigma \rightarrow Q'$ è la nuova **funzione di transizione** che ci permette di navigare tra i possibili sottoinsiemi, ed è tale che

$$\delta'(\alpha, a) = \bigcup_{q \in \alpha} \delta(q, a);$$

- $q_0' = \{q_0\}$ nuovo **stato iniziale**;
- $F' = \{\alpha \in Q' \mid \alpha \cap F \neq \emptyset\}$ nuovo **insieme degli stati finali**.

Come vediamo, il non determinismo è estremamente comodo, perché ci permette di rendere molto **compatta** la rappresentazione degli automi, ma è irrealistico pensare di fare sempre la scelta giusta nelle scommesse.

2.3. Forme di non determinismo

Il non determinismo sulle **transizioni**, ovvero avere più opzioni per la stessa lettera a partire da uno stato, non è l'unica forma di non determinismo che abbiamo.

Infatti, un'altra forma di non determinismo è quella di avere **stati iniziali multipli**, ovvero poter scegliere più punti di partenza. Come potenza siamo uguali agli automi deterministici: basta fare una **costruzione per sottoinsiemi** e abbiamo sistemato tutto.

L'ultima forma di non determinismo che abbiamo è quella delle ε -**produzioni**, o ε -**mosse**: esse sono transizioni di stato etichettate dalla ε che permettono di spostarsi da uno stato all'altro senza leggere un carattere della stringa da riconoscere.

Che applicazioni può avere una forma del genere? Nei **compilatori** questo approccio è comodissimo per riconoscere dei numeri che possono essere indicati con o senza segno.

Esempio 2.3.1: Se $\Sigma = \{0, \dots, 9, +, -\}$ definiamo un numero come una sequenza non vuota di cifre, con un segno iniziale opzionale.



La epsilon mossa indica una opzionalità: potremmo leggere il prossimo carattere stando nello stato q_0 oppure nello stato q_s .

Questa soluzione aumenta la potenza dell'automa? **NO**: ogni sequenza nella forma

$$p \xrightarrow{\varepsilon} p' \xrightarrow{a} q' \xrightarrow{\varepsilon} q$$

può essere tradotta nella transizione

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

Esempio 2.3.2: Andiamo a rimuovere la ε -transizione usando le sequenze appena descritte.



Una soluzione analoga rimuove le ε -transizioni inserendo degli stati iniziali multipli, ma questo mantiene ancora la forma di non determinismo dell'automa e non migliora la potenza, visto che basta trasformare l'NFA in un DFA con la costruzione per sottoinsiemi e come stato iniziale si avranno più di due elementi.



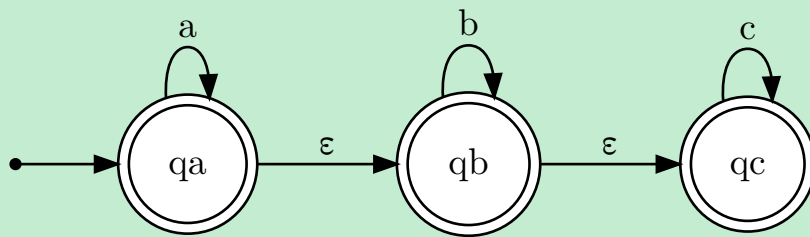
Vediamo altre applicazioni delle ε -produzioni.

Esempio 2.3.3: Ci vengono dati tre automi, che riconoscono sequenze di a , b e c arbitrarie.

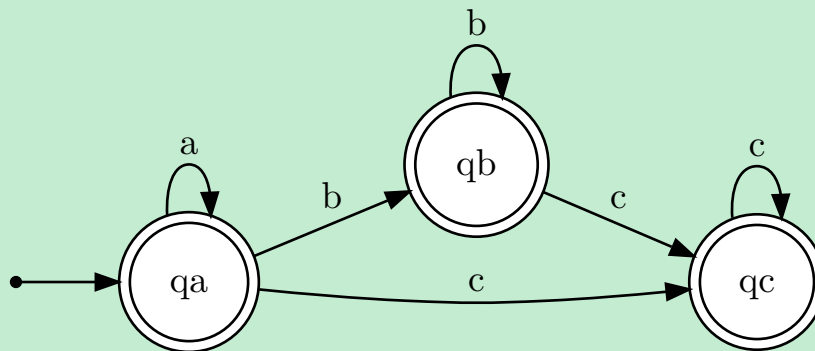


Vogliamo costruire un automa che utilizzi le ε -transizioni usando questi tre moduli per riconoscere il linguaggio

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

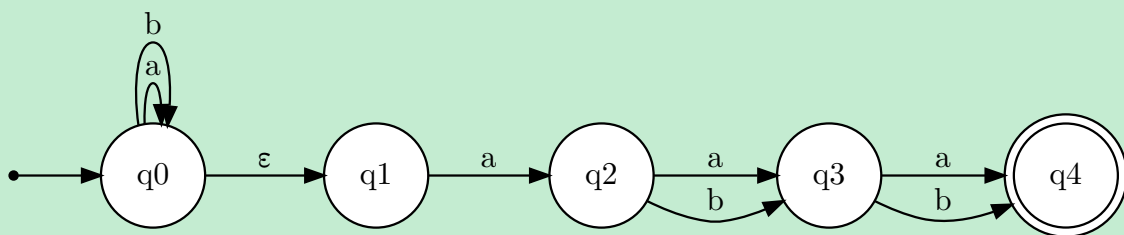


Come lo rendiamo deterministico? Sicuramente non andiamo ad utilizzare gli stati iniziali multipli, che qui ci starebbero molto bene, ma appunto vogliamo un comportamento deterministico.



Abbiamo un automa deterministico, ma quello di prima è molto più leggibile di questo.

Esempio 2.3.4: Riprendiamo il linguaggio L_n delle stringhe con l' n -esimo carattere da destra uguale ad una a . Avevamo visto un NFA sulle transizioni nell'[Esempio 2.1](#), vediamone uno non deterministico sulle ε -transizioni fissando il valore a $n = 3$.



La scommessa qua l'abbiamo messa nel primo stato, che cerca di indovinare se sia arrivato o meno al terzultimo carattere. Il numero di stati, per L_n generico, è $n + 2$.

3. Numero minimo di stati

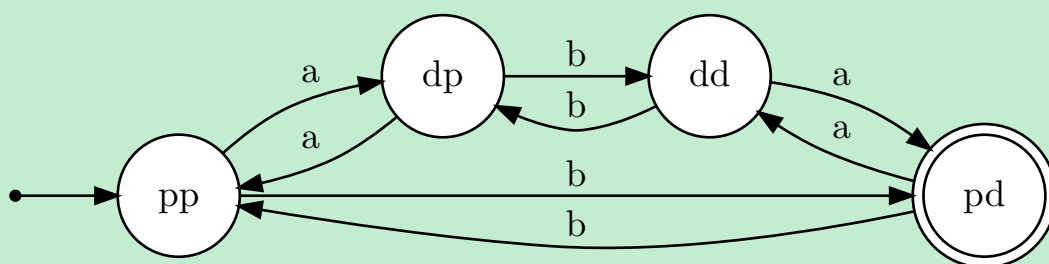
Come anticipato nell'Esempio 1.2.4 del Capitolo 1, in questo capitolo andiamo ad analizzare alcune **tecniche** per conoscere il **minimo numero di stati** che deve avere un automa per riconoscere un certo linguaggio.

3.1. Distinguibilità

Vediamo prima di tutto un esempio per introdurre alcuni concetti utili.

Esempio 3.1.1: Vogliamo trovare un automa che riconosca il linguaggio

$$L = \{x \in \{a, b\}^* \mid \#_a(x) \text{ pari} \wedge \#_b(x) \text{ dispari}\}$$



Ogni stato si ricorda il numero di a e b modulo 2 che ha incontrato.

Possiamo usare **meno stati** per descrivere un automa per questo linguaggio?

Per rispondere alla domanda dell'Esempio 3.1.1 ci serve un **criterio** da applicare facilmente e a qualsiasi linguaggio ci capiti davanti. Abbiamo detto «*qualsiasi linguaggio*» perché il criterio che andiamo a descrivere ora lavora sui **linguaggi** e non sugli automi.

Definizione 3.1.1 (*Distinguibilità*): Sia $L \subseteq \Sigma^*$ un linguaggio e siano $x, y \in \Sigma^*$ due stringhe. Allora x e y sono **distinguibili** per L se

$$\exists z \in \Sigma^* \mid (xz \in L \wedge yz \notin L) \vee (xz \notin L \wedge yz \in L).$$

In poche parole, due stringhe sono **distinguibili** se riesco a trovare una terza stringa z che, attaccata alle due stringhe x e y , da una parte mi fa stare in L mentre dall'altra mi manda fuori.

Teorema 3.1.1 (*Teorema della distinguibilità*): Sia $L \subseteq \Sigma^*$ e sia $X \subseteq \Sigma^*$ un insieme tale che tutte le coppie di stringhe $x, y \in X$, con $x \neq y$, sono distinguibili. Allora ogni automa deterministico che accetta L ha almeno $|X|$ stati.

Dimostrazione 3.1.1.1: Sia $X = \{x_1, \dots, x_n\}$ e sia $A = (Q, \Sigma, \delta, q_0, F)$ un DFA che accetta il linguaggio L . Definiamo gli stati

$$p_i = \delta(q_0, x_i) \quad \forall i = 1, \dots, n$$

che raggiungiamo dallo stato iniziale usando le stringhe x_i di X . Graficamente abbiamo

$$\begin{array}{c} x_0 \\ q_0 \rightsquigarrow p_0 \\ \dots \\ x_n \\ q_0 \rightsquigarrow p_n \end{array}$$

Per assurdo, supponiamo che $|Q| < n$. Ma allora esistono due stati tra i vari p_i che sono raggiunti da due stringhe diverse, ovvero

$$\exists i \neq j \mid p_i = p_j.$$

Per ipotesi x_i e x_j sono due stringhe distinguibili, quindi esiste una stringa $z \in \Sigma^*$ che le distingue. Ma partendo dallo stesso stato $p_i = p_j$ e applicando z vado per entrambe le stringhe in uno stato finale o in uno stato non finale.

Ma questo è un assurdo perché va contro la definizione di distinguibilità, quindi non può succedere che

$$|Q| < n \implies |Q| \geq n.$$

■

3.2. Applicazioni del concetto di distinguibilità

Andiamo ad applicare il [Teorema 3.1.1](#) a qualche linguaggio.

Esempio 3.2.1: Riprendiamo il linguaggio dell'[Esempio 3.1.1](#) e cerchiamo di costruire un insieme X di stringhe distinguibili.

	ε	a	b	ab
ε	—	b	b	b
a	b	—	ab	ab
b	b	ab	—	ε
ab	b	ab	ε	—

Un approccio comodo per creare un insieme di stringhe distinguibili è usare una stringa **per ogni stato** dell'automa a disposizione.

Esempio 3.2.2: Consideriamo il linguaggio che ha una a in terza posizione da destra e diamogli un nome:

$$L_3 = \{x \in \{a, b\}^* \mid \text{il terzo simbolo di } x \text{ da destra è una } a\}.$$

Avevamo visto un DFA per L_3 che prendeva una finestra di 3 simboli, usando 8 stati. Possiamo farlo con meno di 8 stati? Vediamo se troviamo dei bound al numero di stati.

Se scegliamo $X = \Sigma^3$, date due stringhe $\sigma, \gamma \in X$ tali che

$$\sigma = \sigma_1\sigma_2\sigma_3 \quad | \quad \gamma = \gamma_0\gamma_1\gamma_2$$

allora queste due stringhe le riusciamo a distinguere in base ad una delle posizioni nelle quali hanno un carattere diverso. Infatti, visto che

$$\exists i \mid \sigma_i \neq \gamma_i$$

possiamo affermare che:

- se $i = 1$ allora scelgo $z = \varepsilon$;
- se $i = 2$ allora scelgo $z \in \{a, b\}$;
- se $i = 3$ allora scelgo $z \in \{a, b\}^2$.

Con questa costruzione, noi «rimuoviamo» i caratteri prima della posizione i e aggiungiamo in fondo una qualsiasi sequenza della stessa lunghezza. Abbiamo ottenuto una stringa della stessa lunghezza che però ora ha in prima posizione i due caratteri diversi esattamente nella posizione dove dovremmo avere una a .

Cerchiamo di generalizzare questo concetto.

Esempio 3.2.3: Definiamo il linguaggio

$$L_n = \{x \in \{a, b\}^* \mid \text{l}'n\text{-esimo simbolo di } x \text{ da destra è una } a\}.$$

Come prima, definiamo $X = \Sigma^n$ insieme di stringhe nella forma $\sigma = \sigma_1 \dots \sigma_n$.

Date due stringhe $\sigma, \gamma \in \Sigma^n$ allora

$$\exists i \mid \sigma_i \neq \gamma_i.$$

Questa posizione può essere la prima o una a caso, è totalmente indifferente. Come stringa da attaccare scegliamo

$$z \in \Sigma^{i-1}.$$

Con questa stringa riusciamo a distinguere σ da γ : infatti, come prima, «isoliamo» i primi $i - 1$ caratteri, li «spostiamo» alla fine in un'altra forma e consideriamo solo gli n caratteri di destra. In questa nuova «configurazione» abbiamo l' n esimo carattere della stringa che è quello che era in posizione i , che in una stringa vale a e in una vale b , quindi le due stringhe sono distinguibili.

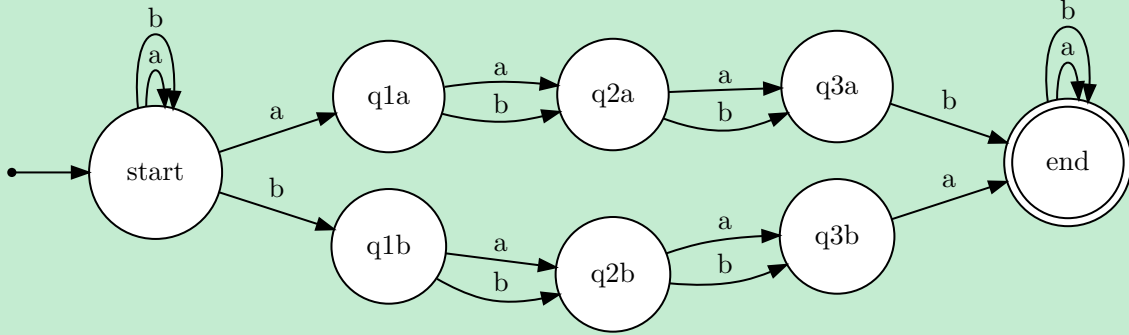
Ma allora ogni DFA per L_n usa almeno $2^{|X|} = 2^n$ stati.

Visto che siamo bravi con le scommesse, andiamo a fare un po' di sano **gambling**.

Esempio 3.2.4: Definiamo

$$D_n = \{x \in \{a, b\}^* \mid \exists \text{ due simboli di } x \text{ a distanza } n \text{ che sono diversi}\}.$$

Vediamo un NFA per D_3 , dove appunto viene fissato $n = 3$.



Una NFA per L_n utilizza $2n + 2$ stati, più un eventuale stato trappola.

Esempio 3.2.5: Riusciamo a trovare un bound al numero di stati di ogni DFA per il linguaggio dell'Esempio 3.2.4 precedente con un n generico?

Sia $X = \Sigma^n$ un insieme di stringhe distinguibili per D_n .

Prendiamo le stringhe $\sigma = \sigma_1 \dots \sigma_n$ e $\gamma = \gamma_1 \dots \gamma_n$ di X , e sia i la prima posizione nella quale le due stringhe sono diverse, ovvero $\sigma_i \neq \gamma_i$. Come stringa z scegliamo

$$z = \sigma_1 \dots \sigma_{i-1} \{a, b\}$$

Con questa scelta otteniamo le stringhe

$$\sigma z = \sigma_1 \dots \sigma_{i-1} \sigma_i \sigma_{i+1} \dots \sigma_n \sigma_1 \dots \sigma_{i-1} \{a, b\}$$

$$\gamma z = \gamma_1 \dots \gamma_{i-1} \gamma_i \gamma_{i+1} \dots \gamma_n \gamma_1 \dots \gamma_{i-1} \{a, b\}.$$

Tutti i simboli $\sigma_1 \dots \sigma_{i-1}$ sono uguali in entrambe le stringhe, mentre i simboli σ_i e $\{a, b\}$ saranno in una stringa uguali e in una diversi, quindi verrà accettata la prima o la seconda stringa e l'altra no.

Ma allora ogni DFA per D_n richiede almeno 2^n stati.

Vediamo ancora un esempio, ma teniamo a mente il linguaggio D_n che abbiamo appena visto.

Esempio 3.2.6: Dato l'alfabeto $\Sigma = \{a, b\}$, definiamo

$$D'_n = \{x \in \Sigma^* \mid \text{ogni coppia di simboli di } x \text{ a distanza } n \text{ contiene lo stesso simbolo}\}.$$

Notiamo che dopo che ho letto n simboli essi si iniziano a ripetere fino alla fine, ma allora

$$x \in D'_n \iff \exists w \in \Sigma^n \wedge \exists y \in \Sigma^{\leq n} \mid x = w^{m \geq 0} y \wedge y \text{ suffisso di } w.$$

Posso ripetere w quante volte voglio, ma poi la parte finale deve ripetere in parte w .

Notiamo inoltre che questo linguaggio è il complementare del precedente, ovvero

$$D'_n = D_n^C.$$

Vogliamo costruire un DFA per questo linguaggio: posso usare l'insieme X usato per D_n ma cambiare il valore di verità finale. Quindi ci servono ancora 2^n stati per il DFA.

Vediamo un esempio di automa con $n = 3$, un po' grossino, ma fa niente. Non viene inserito lo stato trappola per semplicità, ma ci dovrebbe essere anche quello per ogni transizione «sbagliata» nell'ultima parte dell'automata.

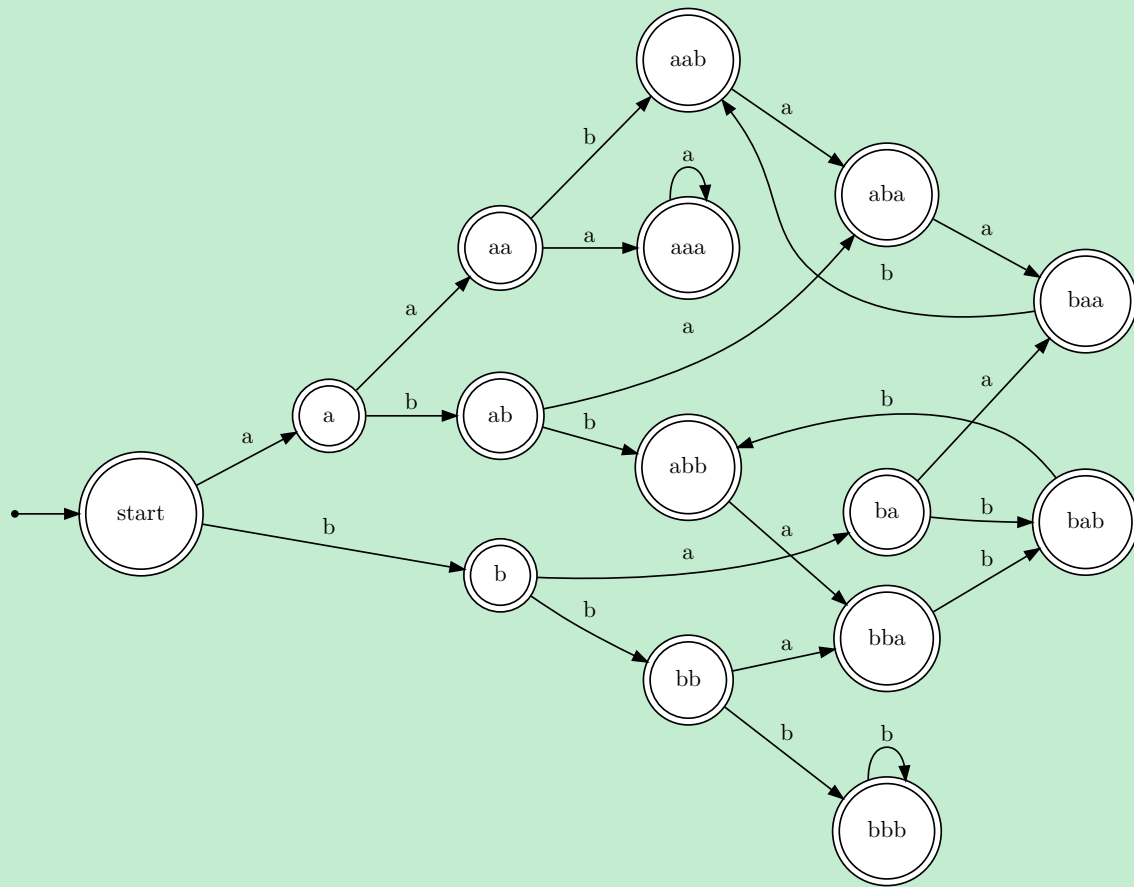


Per il linguaggio generico D'_n , l'albero usa un numero di stati pari a

$$2^{n+1} - 1 + -2^n(n - 1) + 1 = 2^{n+1} + 2^n(n - 1).$$

Una prima versione migliore dell'automa taglia via 4 stati facendo dei cappi negli stati $aaa1$ e $bbb1$, ma il numero rimane sempre esponenziale sotto steroidi.

Una seconda versione ancora migliore taglia tutti i $2^n(n-1)$ stati finali che fanno i cicli. Come mai? Possiamo usare tutte le foglie per mantenere comunque i cicli, abbastanza pesante da vedere però un bro è fortissimo e ha visto sta cosa.



Questa bellissima versione ha un numero di stati pari a

$$2^{n+1} - 1 + 1 = 2^{n+1}.$$

Come vediamo, in entrambi i casi abbiamo un numero esponenziale di stati, ma almeno abbiamo un automa deterministico da utilizzare.

Come vediamo, questo teorema è un'arma molto potente: oltre alla possibilità di dare dei **lower bound** al numero di stati di un automa, questo ci permette anche di dire se un linguaggio è di tipo 3 o meno. Infatti, se riusciamo a trovare un insieme X per un linguaggio L che ha un numero infinito di stringhe distinguibili, allora L non può essere riconosciuto da un automa a **STATI FINITI**.

Esempio 3.2.7: Definiamo il linguaggio

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

Se scegliamo $X = \{a^n \mid n \geq 0\}$, esso è un insieme di stringhe tutte distinguibili tra loro.

Infatti, date $x = a^i$ e $y = a^j$, con $i \neq j$, basta scegliere

$$z = b^i$$

per avere xz accettata e yz non accettata.

Ma allora L non può essere riconosciuto da un automa a stati finiti.

3.3. Automa di Meyer-Fischer

Abbiamo visto qualche applicazione del [Teorema 3.1.1](#). Osserviamo che spesso il numero di stringhe nell'insieme X (circa) corrisponde al numero di stati che otteniamo trasformando un NFA (minimo) in un DFA con la **costruzione per sottoinsiemi**. In un DFA costruito in questo modo però non servono sempre tutti i sottoinsiemi. Possiamo quindi dare un bound alla costruzione per sottoinsiemi, affermando quindi che il salto esponenziale è solo teorico? La risposta sarà **NO**, ma per capire perché vediamo qualche esempio.

Esempio 3.3.1: Riprendiamo il linguaggio dell'[Esempio 3.2.3](#) e costruiamo un NFA: per fare ciò basta adattare l'NFA che avevamo già costruito per L_3 nell'[Esempio 2.1](#) del [Capitolo 2](#) in cui venivano presentati gli NFA.

Usiamo quindi uno stato che fa la scommessa di essere arrivati all' n -esimo carattere da destra e uno stato che si ricorda di aver letto una a . Servono poi $n - 1$ stati per leggere i restanti $n - 1$ caratteri della stringa.



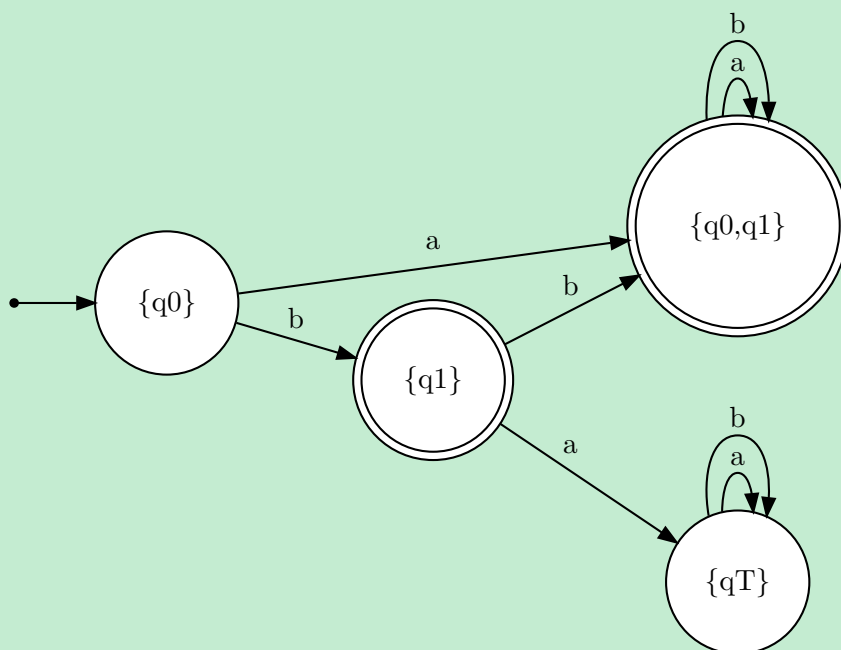
Il numero totale di stati è $n + 1$.

Per L_n abbiamo quindi visto che il numero di stati richiesti per un NFA è $n + 1$, mentre per un DFA è almeno 2^n grazie al [Teorema 3.1.1](#). Il salto che abbiamo fatto è quindi **esponenziale**.

Esempio 3.3.2: Dato il seguente NFA, costruire il DFA associato.



Usando la costruzione per sottoinsiemi otteniamo il seguente DFA.



Escludendo lo stato trappola siamo riusciti ad usare meno stati di quelli del salto $n \rightarrow 2^n$, quindi vuol dire che forse si riesce a fare meglio. E invece **NO**. Esiste un caso peggiore, un automa che esegue un salto preciso da n a 2^n preciso preciso.

Come per la teoria della complessità, dobbiamo considerare sempre il **caso peggiore**, quindi vedremo un salto da n a 2^n esaurendo completamente tutti i possibili sottoinsiemi di n . Poi si può fare di meglio, ma in generale si fa tutto il salto visto che esiste un controesempio.

L'automa che viene portato come sacrificio per questa causa è l'automa di Meyer-Fischer.

L'**automa di Meyer-Fischer**, ideato da questi due bro nel 1971, sarà il nostro NFA salvatore che ci permetterà di dimostrare quanto detto fino ad adesso.

Sia $M_n = (Q, \Sigma, \delta, q_0, F)$ tali che:

- $Q = \{0, \dots, n-1\}$ insieme di n stati;
- $\Sigma = \{a, b\}$;
- $q_0 = 0$ stato iniziale e anche unico stato finale.

La funzione di transizione è tale che

$$\delta(i, x) = \begin{cases} \{(i+1) \bmod n\} & \text{se } x = a \\ \{i, 0\} & \text{se } x = b \\ \emptyset & \text{se } x = b \wedge i = 0 \end{cases}.$$

L'automa M_n lo possiamo disegnare in questo modo.



Teorema 3.3.1: Ogni DFA equivalente a M_n deve avere almeno 2^n stati.

Dimostrazione 3.3.1.1: Sia $S \subseteq \{0, \dots, n-1\}$. Definiamo la stringa

$$w_S = \begin{cases} b & \text{se } S = \emptyset \\ a^i & \text{se } S = \{i\} \\ a^{e_k - e_{k-1}} b a^{e_{k-1} - e_{k-2}} b \dots b a^{e_2 - e_1} b a^{e_1} & \text{se } S = \{e_1, \dots, e_k\} \mid k > 1 \wedge e_1 < \dots < e_k \end{cases}.$$

Si può dimostrare (vedi [Lemma 3.3.1](#)) che per ogni $S \subseteq \{0, \dots, n-1\}$ vale

$$\delta(q_0, w_S) = S.$$

Si può dimostrare (vedi [Lemma 3.3.2](#)) inoltre che dati $S, T \subseteq \{0, \dots, n-1\}$, se $S \neq T$ allora w_S e w_T sono distinguibili per il linguaggio $L(M_n)$.

Viste queste due proprietà, l'insieme di tutte le stringhe w_S associate ai vari insiemi S è formato da stringhe indistinguibili tra loro a coppie. Definiamo quindi

$$X = \{w_S \mid S \subseteq \{0, \dots, n-1\}\}$$

insieme di stringhe distinguibili tra loro per $L(M_n)$.

Il numero di stringhe in X dipende dal numero di sottoinsiemi di $\{0, \dots, n-1\}$: questi sono esattamente 2^n , quindi anche $|X| = 2^n$. Ma allora, per il teorema sulla distinguibilità, ogni DFA per M_n deve usare almeno 2^n stati. ■

Formalizziamo un attimo le due proprietà utilizzate. Vediamo la prima.

Lemma 3.3.1: Per ogni $S \subseteq \{0, \dots, n-1\}$ vale

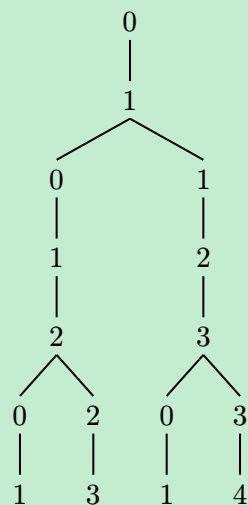
$$\delta(q_0, w_S) = S.$$

Esempio 3.3.3: Sia M_5 un'istanza dell'automa di Meyer-Fischer.

Se scegliamo $S = \{1, 3, 4\}$ allora

$$w_S = a^{4-3}ba^{3-1}ba^1 = abaaba.$$

Facciamo girare l'automa M_5 sulla stringa w_S .



Notiamo come l'insieme degli stati finali possibili sia esattamente S .

E ora vediamo la seconda e ultima proprietà.

Lemma 3.3.2: Dati $S, T \subseteq \{0, \dots, n-1\}$, se $S \neq T$ allora w_S e w_T sono distinguibili per il linguaggio $L(M_n)$.

Dimostrazione 3.3.2.1: Se $S \neq T$ allora sia $x \in S/T$ uno degli elementi che sta in S ma non in T . Vale anche il simmetrico, quindi consideriamo questo caso per ora.

Per il [Lemma 3.3.1](#), sappiamo che

$$\delta(q_0, w_S) = S \quad | \quad \delta(q_0, w_T) = T.$$

Se siamo nello stato x , se vogliamo finire nello stato finale basta leggere la stringa a^{n-x} . Infatti, dato l'insieme S che contiene x , allora

$$w_S a^{n-x} \in L(M_n)$$

perché lo stato x finisce nello stato finale.

Ora, visto che $x \notin T$, allora $w_T a^{n-x} \notin L(M_n)$ perché l'unico modo per finire in 0 leggendo a^{n-x} è essere nello stato x , come visto poco fa.

Ma allora w_S e w_T sono distinguibili. ■

3.4. Fooling set

Abbiamo visto un criterio di distinguibilità per i DFA, ma ne esiste uno anche per gli NFA.

Definizione 3.4.1 (*Fooling set*): Sia $L \subseteq \Sigma^*$. Definiamo

$$P = \{(x_i, y_i) \mid i = 1, \dots, N\} \subseteq \Sigma^* \times \Sigma^*$$

un insieme di N coppie formate da stringhe di Σ^* .

L'insieme P è un **fooling set** per L se:

1. $\forall i \in \{1, \dots, N\} \quad x_i y_i \in L$;
2. $\forall i, j \in \{1, \dots, N\} \mid i \neq j \quad x_i y_j \notin L$.

Cosa ci stanno dicendo queste due proprietà? La prima ci dice che la concatenazione degli elementi della stessa coppia forma una stringa che appartiene al linguaggio, mentre la seconda ci dice che la concatenazione della prima parte di una coppia con la seconda parte di un'altra coppia forma una stringa che non appartiene al linguaggio.

Noi useremo una versione leggermente diversa del fooling set.

Definizione 3.4.2 (*Extended fooling set*): Un **extended fooling set** è un fooling set nel quale viene modificata la seconda proprietà, ovvero:

1. $\forall i \in \{1, \dots, N\} \quad x_i y_i \in L$;
2. $\forall i, j \in \{1, \dots, N\} \mid i \neq j \quad x_i y_j \notin L \vee x_j y_i \notin L$.

Come vediamo, è una versione un pelo più rilassata: prima chiedevo che, presa ogni prima parte di indice i , ogni concatenazione con seconde parti di indice j mi desse una stringa fuori dal linguaggio. Ora invece me ne basta solo uno dei due versi.

Teorema 3.4.1 (*Teorema del fooling set*): Se P è un extended fooling set per il linguaggio L allora ogni NFA per L deve avere almeno $|P|$ stati.

Dimostrazione 3.4.1.1: Concentriamoci solo sui cammini accettanti che possiamo avere in un NFA per il linguaggio L . Grazie alla prima proprietà di P , sappiamo che le stringhe $z = x_i y_i$ stanno in L . Calcoliamo i cammini per ogni coppia di P , che sono N :

$$\begin{array}{ccc}
& x_1 & y_1 \\
q_0 & \rightsquigarrow & p_1 & \rightsquigarrow & f_1 \\
& \vdots & \\
& x_N & y_N \\
q_0 & \rightsquigarrow & p_N & \rightsquigarrow & f_N
\end{array}$$

Per assurdo sia A un NFA con meno di N stati. Ma allora esistono due stringhe $x_i \neq x_j$ che mi fanno andare in $p_i = p_j$. Sappiamo che:

- da p_i con y_i vado in uno stato finale;
- da p_j con y_j vado in uno stato finale.

Sappiamo che $p_i = p_j$, ma quindi $x_i y_j$ è una stringa che finisce in uno stato finale, ma questo è un assurdo perché contraddice la seconda proprietà del fooling set.

Quindi ogni NFA deve avere almeno N stati. ■

3.5. Applicazioni del concetto di fooling set

Usiamo il [Teorema 3.4.1](#) per valutare un NFA per il linguaggio precedente.

Esempio 3.5.1: Dato il linguaggio D'_n dell'[Esempio 3.2.6](#), definiamo l'insieme

$$P = \{(x, x) \mid x \in \Sigma^n\}$$

extended fooling set per D'_n . Infatti, ogni stringa $z = xx$ appartiene a D'_n , mentre ogni «stringa incrociata» $z = xy$, con $x \neq y$, non appartiene a D'_n perché in almeno una posizione a distanza n ho un carattere diverso.

Il numero di elementi di P è 2^n , che è il numero di configurazioni lunghe n di 2 caratteri, quindi ogni NFA per D'_n ha almeno 2^n stati.

Vediamo un mini **riassunto** dei due linguaggi visti di recente.

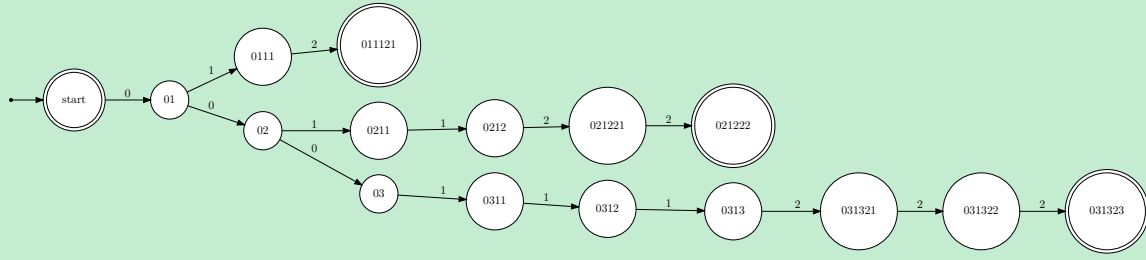
Linguaggio	DFA	NFA
D_n	$\geq 2^n$	$\leq 2n + 2$
D'_n	$\geq 2^n \wedge \leq 2^{n+1}$	$\geq 2^n$

Finiamo con un ultimo esempio.

Esempio 3.5.2: Dato il linguaggio $\Sigma = \{0, 1, 2\}$, definiamo il linguaggio

$$T_n = \{0^i 1^i 2^i \mid 0 \leq i \leq n\}.$$

Diamo un DFA per questo linguaggio, fissando $n = 3$.



Il numero di stati del linguaggio T_n generico è

$$\sum_{i=0}^n (2i + 1) = n^2.$$

Per finire diamo un NFA per il linguaggio T_n . Visto che non sappiamo su cosa scommettere, diamo un lower bound al numero di stati dei nostri NFA.

Creiamo un fooling set

$$P = \{(0^i 1^j, 1^{i-j} 2^i) \mid i = 1, \dots, n \wedge j = 1, \dots, i\}.$$

Questo è un fooling set per T_n :

- una coppia ci dà la stringa $z = 0^i 1^{j+i-j} 2^i = 0^i 1^i 2^i$ che appartiene al linguaggio;
- prendendo due elementi da due coppie diverse:
 - se sono diverse le i abbiamo un numero di 0 e 2 diversi;
 - se sono uguali le i allora sono diverse le j , ma allora la stringa $0^i 1^{j+i-j'} 2^i$ non appartiene al linguaggio perché $j + i - j' \neq i$.

Il numero di stati di P è ancora una somma di Gauss, quindi ogni NFA per T_n ha almeno un numero quadratico di stati.

4. Equivalenza tra linguaggi di tipo 3 e automi a stati finiti

In questo capitolo mostreremo l'**equivalenza** tra le grammatiche di tipo 3 e gli automi a stati finiti.

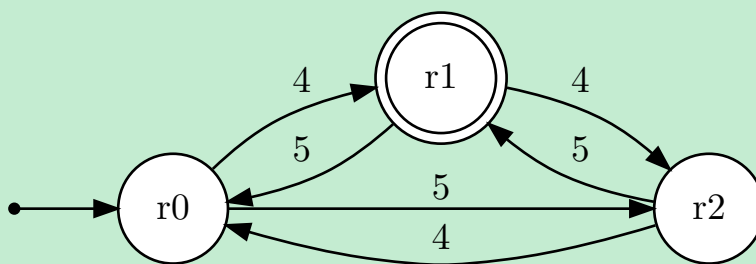
4.1. Dall'automata alla grammatica

Dato un automa $A = (Q, \Sigma, \delta, q_0, F)$ per il linguaggio L , costruiamo una grammatica G di tipo 3 che riconosca lo stesso linguaggio L .

Per fare ciò dobbiamo definire le variabili, l'assioma e le produzioni. Definiamo quindi G tale che:

- le **variabili** sono gli stati dell'automata, ovvero $V = Q$;
- l'**assioma** è lo stato iniziale dell'automata, ovvero $S = q_0$;
- le **produzioni** derivano dalle transizioni e sono nella forma:
 - $q \rightarrow ap$ se la funzione di transizione è tale che $\delta(q, a) = p$;
 - alla produzione precedente aggiungiamo la produzione $q \rightarrow a$ se p è uno stato finale, questo perché posso fermarmi in p .

Esempio 4.1.1: Sia $\Sigma = \{4, 5\}$. Ci viene fornito un automa che, date le stringhe sull'alfabeto Σ interpretate come numeri decimali, una volta divise per 3 ci danno 1 come resto.



Costruiamo una grammatica G di tipo 3 analoga a questo automa. Sia quindi G tale che:

- variabili $V = \{r_0, r_1, r_2\}$;
- assioma $S = r_0$;
- produzioni P :
 - $r_0 \rightarrow 4r_1 \mid 4 \mid 5r_2$;
 - $r_1 \rightarrow 4r_2 \mid 5r_0$;
 - $r_2 \rightarrow 4r_0 \mid 5r_1 \mid 5$.

Proviamo a derivare una stringa per vedere se effettivamente funziona:

$$r_0 \rightarrow 4r_1 \rightarrow 45r_0 \rightarrow 455r_2 \rightarrow 4555.$$

4.2. Dalla grammatica all'automata

In maniera analoga, data la grammatica G di tipo 3 creiamo un automa A tale che:

- **stati** $Q = V \cup \{q_F\}$;
- **stato iniziale** $q_0 = S$;
- **stati finali** $F = \{q_F\}$;
- **transizioni** della funzione di transizione derivano dalle regole di produzione, ovvero:
 - per ogni produzione $(A \rightarrow aB) \in P$ essa ci dice che dallo stato A leggendo una a andiamo a finire in B , ovvero $\delta(A, a) = B$;

- per ogni produzione $(A \rightarrow a) \in P$ essa ci dice che possiamo finire la derivazione, cioè che andiamo da A in uno stato finale tramite a , ovvero $\delta(A, a) = q_F$.

Per essere più precisi, definiamo i passi della funzione di transizione come

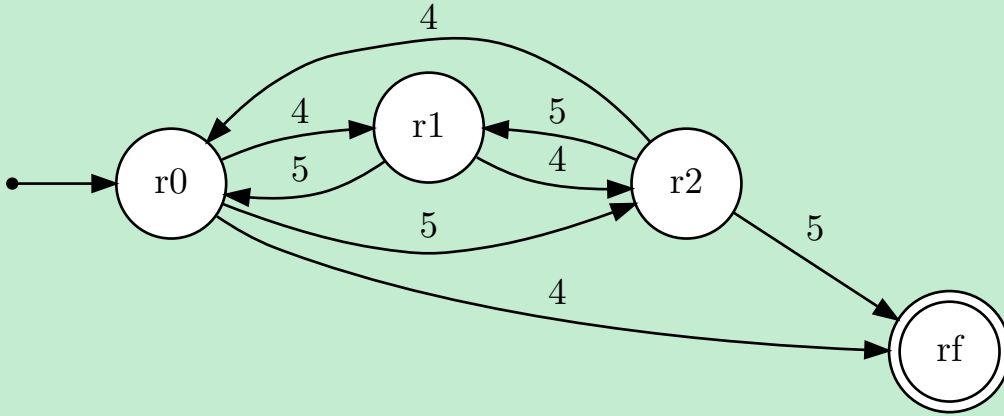
$$\delta(A, a) = \{B \mid (A \rightarrow aB) \in P\} \cup \{q_F \text{ se } (A \rightarrow a) \in P\}$$

Esempio 4.2.1: Data la grammatica $G = (V, \Sigma, P, S)$ tale che:

- $V = \{r_0, r_1, r_2\}$;
- $S = r_0$;
- produzioni P :
 - $r_0 \rightarrow 4r_1 \mid 4 \mid 5r_2$;
 - $r_1 \rightarrow 4r_2 \mid 5r_0$;
 - $r_2 \rightarrow 4r_0 \mid 5r_1 \mid 5$.

Ricaviamo un automa dalla grammatica G . Per fare ciò definiamo:

- $Q = \{r_0, r_1, r_2, r_f\}$;
- $q_0 = r_0$;
- $F = \{r_f\}$;
- funzione di transizione δ che ha il seguente comportamento:
 - $\delta(r_0, 4) = \{r_1, r_f\}$;
 - $\delta(r_0, 5) = \{r_2\}$;
 - $\delta(r_1, 4) = \{r_2\}$;
 - $\delta(r_1, 5) = \{r_0\}$;
 - $\delta(r_2, 4) = \{r_0\}$;
 - $\delta(r_2, 5) = \{r_1, r_f\}$.



Notiamo come l'automa ottenuto sia non deterministico e, soprattutto, non è l'automa minimo che avevamo invece nell'esempio precedente.

4.3. Grammatiche lineari

Stiamo parlando di grammatiche, quindi vediamo un tipo particolare di grammatiche che però incontreremo molto più avanti: le **grammatiche lineari**.

4.3.1. Grammatiche lineari a destra

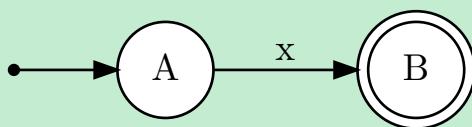
Potrebbero capitarci delle grammatiche che hanno una forma simile a quelle regolari, ma che in realtà non lo sono. Queste grammatiche hanno le produzioni nella forma

$$A \longrightarrow xB \mid x \text{ tale che } x \in \Sigma^*.$$

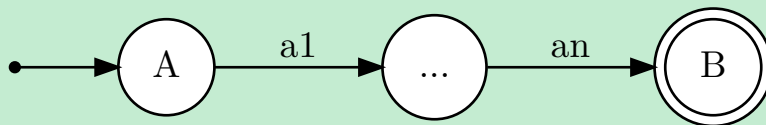
Non abbiamo più, come nelle grammatiche regolari, la stringa x formata da un solo terminale, ma possiamo averne un numero arbitrario.

Queste grammatiche sono dette **grammatiche lineari a destra**, ma nonostante questa aggiunta di terminali non aumentiamo la potenza del linguaggio: per generare quella sequenza di terminali x basta aggiungere una serie di regole che rispettano le grammatiche regolari che generino esattamente la stringa x .

Esempio 4.3.1.1: Dato l'automa in figura, andare a scrivere l'automa regolare corrispondente.



Nella grammatica avremmo una serie di regole che seguono la forma delle grammatiche regolari per generare la stringa x , ovvero:



Abbiamo quindi sostituito la stringa $x = a_1 \dots a_n$ con una serie di stati intermedi.

4.3.2. Grammatiche lineari a sinistra

Esistono anche le **grammatiche lineari a sinistra**, che hanno le produzioni nella forma

$$A \longrightarrow Bx \mid x \text{ tale che } x \in \Sigma^*.$$

Si dimostra che anche queste grammatiche non vanno oltre i linguaggi regolari, anche se è un accrocchio passare da queste grammatiche a quelle regolari.

4.3.3. Grammatiche lineari

E se facciamo un **mischione** delle due grammatiche precedenti?

Le produzioni di queste grammatiche sono nella forma

$$A \longrightarrow xB \mid Bx \mid x \text{ tale che } x \in \Sigma^* \wedge A, B \in V.$$

Queste grammatiche, che generano i cosiddetti **linguaggi lineari**, sono a cavallo tra le grammatiche di tipo 3 e le grammatiche di tipo 2. Quindi siamo un pelo più forti delle grammatiche regolari, ma non quanto le grammatiche CF.

Esempio 4.3.3.1: Definiamo una grammatica che utilizza le seguenti produzioni:

$$S \longrightarrow aA \mid \varepsilon$$

$$A \longrightarrow Sb$$

Con queste regole di una grammatica lineare stiamo generando il linguaggio

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

che non è un linguaggio di tipo 3.

La cosa che stiamo aggiungendo è una sorta di **ricorsione**, che mi permette di saltare fuori dai linguaggi regolari e catturare di più di prima.

5. Automa minimo

Nel capitolo [Sezione 3](#) abbiamo visto dei metodi che limitavano il numero di stati di DFA e NFA per un certo linguaggio. In questo capitolo vediamo invece un criterio che lavora direttamente sugli automi e non sui linguaggi.

5.1. Introduzione matematica

Definizione 5.1.1 (*Relazione binaria*): Sia S un insieme. Una **relazione binaria** sull'insieme S è definita come l'insieme

$$R \subseteq S \times S.$$

Come notazione useremo

$$x R y$$

oppure $(x, y) \in R$, ma molto di più la prima della seconda.

Ci interessiamo ad un tipo molto particolare di relazioni.

Definizione 5.1.2 (*Relazione di equivalenza*): La relazione R è una **relazione di equivalenza** se e solo se R è:

- **riflessiva**, ovvero $\forall x \in S \quad x R x$;
- **simmetrica**, ovvero $\forall x, y \in S \quad x R y \implies y R x$;
- **transitiva** $\forall x, y, z \in S \quad x R y \wedge y R z \implies x R z$.

Una relazione di equivalenza **induce** sull'insieme S una **partizione** formata da **classi di equivalenza**. Queste classi sono formate da elementi che sono equivalenti tra di loro. Le classi di equivalenza le indichiamo con $[x]_R$, dove $x \in S$ è detto **rappresentante**.

Se R è una relazione di equivalenza, l'**indice** di R è il numero di classi di equivalenza.

Esempio 5.1.1: Sia $S = \mathbb{N}$. Definiamo la relazione $R \subseteq \mathbb{N} \times \mathbb{N}$ tale che

$$x R y \iff x \bmod 3 = y \bmod 3.$$

Questa è una relazione di equivalenza (*non lo dimostriamo*) che ha tre classi di equivalenza:

- $[0]_R$ formata da tutti i multipli di 3;
- $[1]_R$ formata da tutti i multipli di 3 sommati a 1;
- $[2]_R$ formata da tutti i multipli di 3 sommati a 2.

L'indice di questa relazione è quindi 3.

Definizione 5.1.3 (*Relazione invariante a destra*): Sia \cdot un'operazione sull'insieme S . La relazione R è **invariante a destra** rispetto a \cdot se presi due elementi nella relazione

R , e applicando \cdot con uno stesso elemento, otteniamo ancora due elementi in relazione, ovvero

$$x R y \implies \forall z \in S \quad (x \cdot z) R (y \cdot z).$$

Esempio 5.1.2: Sia R la relazione dell'Esempio 5.1.1. Definiamo $\cdot = +$ l'operazione di **somma**. La relazione R è invariante a destra?

Dobbiamo verificare se

$$x R y \implies \forall z \in \mathbb{N} \quad (x + z) R (y + z),$$

ovvero se

$$x \bmod 3 = y \bmod 3 \implies \forall z \in \mathbb{N} \quad (x + z) \bmod 3 = (y + z) \bmod 3.$$

Questo è vero perché ce lo dice l'algebra modulare, quindi R è invariante a destra.

Ora vediamo una definizione che va contro la **semantica italiana**.

Definizione 5.1.4 (*Raffinamento*): Sia S un insieme e siano $R_1, R_2 \subseteq S \times S$ due relazioni di equivalenza su S .

Diciamo che R_1 è un **raffinamento** di R_2 se e solo se:

1. ogni classe di equivalenza di R_1 è contenuta in una classe di equivalenza di R_2
OPPURE
2. ogni classe di R_2 è l'unione di alcune classi di R_1 **OPPURE**
3. vale

$$\forall x, y \in S \quad (x, y) \in R_1 \implies (x, y) \in R_2.$$

Il primo punto è la definizione, gli altri due punti sono solo conseguenze.

Perché non rispecchia molto la semantica italiana? Perché un raffinamento di solito è qualcosa di migliore, in questo caso invece è il contrario: se R_1 è un raffinamento di R_2 allora R_1 è peggiore di R_2 in termini di classi di equivalenza.

Esempio 5.1.3: Presa ancora la relazione R dell'Esempio 5.1.1, definiamo ora la relazione R' tale che

$$x R' y \iff x \bmod 2 = y \bmod 2.$$

Le classi di equivalenza di questa relazione sono $[0]_{R'}$ e $[1]_{R'}$.

Come è messa R rispetto a R' ? E R' rispetto a R ?

Nessuna delle due è un raffinamento dell'altra: ci sono elementi sparsi un po' qua e là quindi non riusciamo a unire le classi di una nelle classi dell'altra.

Sia invece R'' la relazione tale che

$$x R'' y \iff x \bmod 6 = y \bmod 6.$$

La relazione R'' ha 6 classi di equivalenza con le varie classi di resto da 0 a 5.

Come è messa R' rispetto a R'' ? E R'' rispetto a R' ?

Possiamo dire che R'' è un raffinamento di R' : infatti, la classe $[0]_{R'}$ la possiamo scrivere come

$$\bigcup_{i \text{ pari}} [i]_{R''}$$

mentre la classe $[1]_{R'}$ la possiamo scrivere come

$$\bigcup_{i \text{ dispari}} [i]_{R''}.$$

Infine, come è messa R rispetto a R'' ? E R'' rispetto a R ?

Anche in questo caso, possiamo dire che R'' è un raffinamento di R : infatti, la classe $[0]_R$ la possiamo scrivere come

$$[0]_{R''} \cup [3]_{R''},$$

la classe $[1]_R$ la possiamo scrivere come

$$[1]_{R''} \cup [4]_{R''}$$

mentre la classe $[2]_R$ la possiamo scrivere come

$$[2]_{R''} \cup [5]_{R''}.$$

5.2. Relazione R_M

Sia $M = (Q, \Sigma, \delta, q_0, F)$ un DFA. Definiamo la relazione

$$R_M \subseteq \Sigma^* \times \Sigma^*$$

tale che

$$x R_M y \iff \delta(q_0, x) = \delta(q_0, y).$$

In poche parole, due stringhe sono in relazione se e solo se vanno a finire **nello stesso stato**.

Lemma 5.2.1: La relazione R_M è una relazione di equivalenza.

Dimostrazione 5.2.1.1: Facciamo vedere che R_M rispetta RST.

La relazione R_M è riflessiva: banale per la riflessività l'uguale.

La relazione R_M è simmetrica: banale per la simmetria dell'uguale.

La relazione R_M è transitiva: banale per la transitività dell'uguale.

Ma allora R_M è di equivalenza. ■

Lemma 5.2.2: La relazione R_M è invariante a destra rispetto all'operazione di concatenazione.

Dimostrazione 5.2.2.1: Dobbiamo dimostrare che

$$x R_M y \implies \forall z \in \Sigma^* \quad (xz) R_M (yz).$$

Ma questo è vero: con x e y vado nello stesso stato per ipotesi, quindi applicando z ad entrambe le stringhe finiamo nello stesso stato. ■

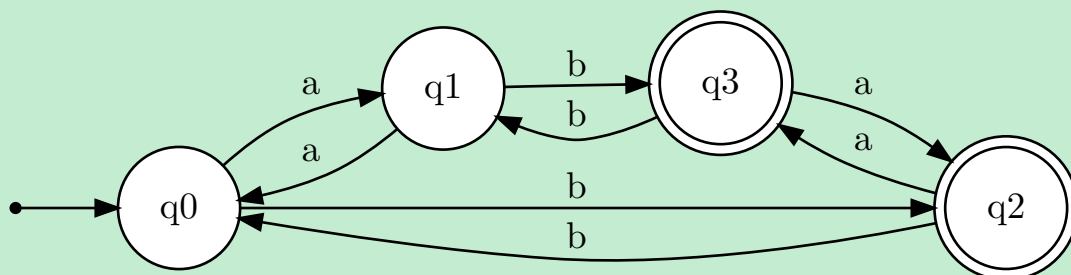
Quante classi di equivalenza abbiamo? Al massimo il numero di stati dell'automa. Come mai diciamo **AL MASSIMO** e non esattamente il numero di stati? Perché in un DFA potremmo avere degli stati che sono irraggiungibili e che quindi non vanno a creare nessuna classe di equivalenza.

In poche parole, R_M è una relazione di equivalenza, invariante a destra e di indice finito limitato dal numero di stati dell'automa M .

Notiamo inoltre che se $(x R_M y)$ allora x e y sono due stringhe non distinguibili per $L(M)$: infatti, esse vanno nello stato e , aggiungendo qualsiasi stringa $z \in \Sigma^*$ per l'invariante a destra, finisco sempre nello stesso stato. In particolare, se finiamo in uno stato finale accettiamo sia x che y , altrimenti entrambe non sono accettate da M .

Abbiamo appena dimostrato che $L(M)$ è l'**unione** di alcune classi di equivalenza di R_M , ovvero tutte le classi di equivalenza che contengono stringhe che mandano in stati finali.

Esempio 5.2.1: Dato il seguente automa deterministico, determinare le classi di equivalenza della relazione R_M appena studiata.



Abbiamo 4 classi di equivalenza, che sono tutte le varie combinazioni di a e b pari/dispari.

Questo automa accetta:

- stringhe con a dispari e b dispari;
- stringhe con a pari e b dispari.

Vedremo dopo come migliorare questo automa.

5.3. Relazione R_L

Dato un linguaggio $L \subseteq \Sigma^*$, ad esso ci associamo una relazione

$$R_L \subseteq \Sigma^* \times \Sigma^*$$

tale che

$$x R_L y \iff \forall z \in \Sigma^* \quad (xz \in L \iff yz \in L)$$

In poche parole, due stringhe x e y sono in relazione se, attaccando una qualsiasi stringa z a quelle date, allora entrambe sono accettate o entrambe sono rifiutate. È praticamente il contrario della **distinguibilità**, visto che partiamo da due stringhe diverse, ci attacchiamo la stessa cosa ma finiamo per essere entrambe accettate o rifiutate.

Lemma 5.3.1: La relazione R_L è una relazione di equivalenza.

Dimostrazione 5.3.1.1: Facciamo vedere che R_L rispetta RST.

La relazione R_L è riflessiva: banale perché sto valutando la stessa stringa.

La relazione R_L è simmetrica: banale per la simmetria del se e solo se.

La relazione R_L è transitiva: banale per la transitività del se e solo se.

Ma allora R_L è di equivalenza. ■

Lemma 5.3.2: La relazione R_L è invariante a destra rispetto all'operazione di concatenazione.

Dimostrazione 5.3.2.1: Dobbiamo dimostrare che

$$x R_L y \implies \forall w \in \Sigma^* \quad (xw) R_L (yw).$$

Se $(x R_L y)$ allora

$$\forall w \in \Sigma^* \quad (xw \in L \iff yw \in L).$$

Prendiamo ora una qualsiasi stringa $z \in \Sigma^*$ e aggiungiamola alle due stringhe, ottenendo xwz e ywz . Se chiamiamo $z' = wz$, con un semplice renaming quello che otteniamo è comunque una stringa di Σ^* che mantiene la relazione R_L , ma effettivamente abbiamo aggiunto qualcosa, la stringa z , quindi abbiamo dimostrato che R_L è invariante a destra. ■

Se prendiamo la stringa $z = \varepsilon$, le stringhe x e y che sono nella relazione R_L sono o entrambe dentro o entrambe fuori da L . Ma allora L è l'**unione** di alcune classi di equivalenza di R_L .

Esempio 5.3.1: Definiamo il linguaggio

$$L = \{x \in \{a, b\}^* \mid \#_a(x) = \text{dispari}\}.$$

Per questo linguaggio abbiamo due classi di equivalenza rispetto alla relazione R_L : una per le a pari e una per le a dispari.

Non abbiamo ancora parlato di **indice** per R_L . Ci sono linguaggi che hanno un numero di classi di equivalenza infinito: ad esempio il linguaggio

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

ha un numero di classi di equivalenza infinito perché non è un linguaggio regolare.

Se confrontiamo il linguaggio dell'Esempio 5.2.1 con il linguaggio dell'Esempio 5.3.1, notiamo che essi descrivono lo stesso linguaggio, ovvero quello delle stringhe con un numero di a dispari, ma abbiamo due situazioni diverse:

- nel primo esempio la relazione R_M ha 4 classi di equivalenza e il DFA ha 4 stati;
- nel secondo esempio la relazione R_L ha 2 classi di equivalenza e il DFA (*non disegnato*) ha 2 stati.

Ma allora R_M è un **raffinamento** di R_L . Questa cosa vale solo per questo esempio? **NO**.

5.4. Teorema di Myhill-Nerode

Teorema 5.4.1 (*Teorema di Myhill-Nerode*): Sia $L \subseteq \Sigma^*$ un linguaggio.

Le seguenti affermazioni sono equivalenti:

1. L è accettato da un DFA, ovvero L è regolare;
2. L è l'unione di alcune classi di equivalenza di una relazione E invariante a destra di indice finito;
3. la relazione R_L associata a L ha indice finito.

Queste relazioni che abbiamo visto fin'ora sono dette **relazioni di Nerode**.

Dimostrazione 5.4.1.1: Facciamo vedere $1 \implies 2 \implies 3 \implies 1$.

[1 \Rightarrow 2]

Sia M un DFA per L . Consideriamo la relazione R_M : abbiamo osservato che essa è:

- di equivalenza;
- invariante a destra;
- di indice finito.

Inoltre, rende L unione di alcune classi di equivalenza, quindi è esattamente quello che vogliamo dimostrare.

[2 \Rightarrow 3]

Supponiamo di avere una relazione

$$E \subseteq \Sigma^* \times \Sigma^*$$

di equivalenza, invariante a destra, di indice finito e che L è l'unione di alcune classi di E .

Sia $(x E y)$. Sappiamo che E è invariante a destra, ovvero vale che

$$\forall z \in \Sigma^* \quad (xz) E (yz).$$

Inoltre, vale che

$$\forall z \in \Sigma^* \quad (xz \in L \iff yz \in L)$$

perché L è unione di alcune classi di equivalenza di E . Ma allora

$$x R_L y$$

per tutta la catena che abbiamo costruito.

Inoltre, E è un raffinamento di R_L , quindi vuol dire che l'indice di E è maggiore di R_L , ovvero

$$\text{indice}(R_L) \leq \text{indice}(E).$$

Visto che E ha indice finito, anche R_L ha indice finito.

[3 \Rightarrow 1]

Sia R_L di indice finito, costruiamo l'automa M' che deve essere un DFA per L .

Definiamo quindi l'automa $M' = (Q', \Sigma, \delta', q'_0, F')$ tale che:

- Q' insieme degli stati formato dalle classi di equivalenza di R_L , ovvero

$$\{[x] \mid x \in \Sigma^*\};$$

- q'_0 stato iniziale che poniamo uguale alla classe di equivalenza che contiene la parola vuota, ovvero

$$q'_0 = [\varepsilon];$$

- δ funzione di transizione tale che

$$\forall \sigma \in \Sigma \quad \delta'([x], \sigma) = [x\sigma];$$

- F insieme degli stati finali formato dalle classi di equivalenza che contengono stringhe del linguaggio, ovvero

$$F' = \{[x] \mid x \in L\}.$$

Ma allora $L(M') = L(M)$ per costruzione. ■

Visto che abbiamo dimostrato questo teorema, possiamo porre E uguale a R_M : otteniamo

$$\text{indice}(R_L) \leq \text{indice}(R_M)$$

se L è un **linguaggio regolare**, altrimenti partiamo a ∞ con le classi di equivalenza di R_L .

5.5. Automa minimo

Finiamo con alcune nozioni che riguardano l'automa minimo. Con **automa minimo** intendiamo il DFA per L con il minimo numero di stati.

Teorema 5.5.1 (*Teorema dell'automa minimo*): Dato un linguaggio L accettato da automi, il DFA minimo per L è unico. Con unicità intendiamo la non esistenza di una configurazione diversa del grafo.

L'automa minimo contiene anche l'eventuale **stato trappola** dove mandiamo i pattern non accettanti. L'automa minimo M' è ottenuto grazie alla relazione R_L .

Per calcolare l'automa minimo abbiamo algoritmi per farlo in modo efficiente, che cercano le stringhe non distinguibili per abbassare il numero di stati. Se troviamo delle stringhe distinguibili siamo arrivati all'automa minimo.

5.6. Applicazioni agli NFA

Cosa succede se applichiamo tutti questi concetti sugli NFA?

Esempio 5.6.1: Costruiamo un po' di automi NFA per stringhe che finiscono in b .





Ovviamente non possiamo andare sotto i 2 stati perché almeno un carattere lo dobbiamo leggere, quindi tutti questi sono **automi minimi** ma **non sono unici**.

Inoltre, per i DFA abbiamo algoritmi polinomiali ben studiati negli anni '60, per gli NFA non abbiamo algoritmi efficienti perché esso è un problema difficile, estremamente difficile, che è ben oltre gli *NP*-completi, ovvero è un problema *PSPACE*-completo

Per fare un confronto, un problema NP-completo è CNF-SAT, un problema PSPACE-completo è CNF-SAT con una serie arbitraria di \forall e \exists posti davanti alla formula CNF.

6. Operazioni tra linguaggi

Supponiamo di avere in mano una serie di linguaggi. Vediamo un po' di **operazioni** che possiamo fare su essi per combinarli assieme e ottenere altri linguaggi importanti.

6.1. Operazioni insiemistiche

I linguaggi sono insiemi di stringhe, quindi perché non iniziare dalle **operazioni insiemistiche**?

Fissato un alfabeto Σ , siano $L', L'' \subseteq \Sigma^*$ due linguaggi definiti sullo stesso alfabeto Σ . Se i due alfabeti sono diversi allora si considera come alfabeto l'**unione** dei due alfabeti.

Partiamo con l'operazione di **unione**:

$$L' \cup L'' = \{x \in \Sigma^* \mid x \in L' \vee x \in L''\}.$$

Continuiamo con l'operazione di **intersezione**:

$$L' \cap L'' = \{x \in \Sigma^* \mid x \in L' \wedge x \in L''\}.$$

Terminiamo con l'operazione di **complemento**:

$$L'^C = \{x \in \Sigma^* \mid x \notin L'\}.$$

Per ora tutto facile, sono le classiche operazioni insiemistiche.

6.2. Operazioni tipiche

Passiamo alle **operazioni tipiche** dei linguaggi, definite comunque molto semplicemente.

Partiamo con l'operazione di **prodotto** (o *concatenazione*):

$$L' \cdot L'' = \{w \in \Sigma^* \mid \exists x \in L' \wedge \exists y \in L'' \mid w = xy\}.$$

In poche parole, concateniamo in tutti i modi possibili le stringhe del primo linguaggio con le stringhe del secondo linguaggio. Questa operazione, in generale, è **non commutativa**, e lo è se Σ è formato da una sola lettera.

Esempio 6.2.1: Vediamo due casi particolari e importanti di prodotto:

$$\begin{aligned} L \cdot \emptyset &= \emptyset \cdot L = \emptyset \\ L \cdot \{\varepsilon\} &= \{\varepsilon\} \cdot L = L. \end{aligned}$$

Andiamo avanti con l'operazione di **potenza**:

$$L^k = \underbrace{L \cdot \dots \cdot L}_{k \text{ volte}}.$$

In poche parole, stiamo prendendo k stringhe da L e le stiamo concatenando in ogni modo possibile. Possiamo dare anche una definizione induttiva di questa operazione, ovvero

$$L^k = \begin{cases} \{\varepsilon\} & \text{se } k = 0 \\ L^{k-1} \cdot L & \text{se } k > 0. \end{cases}$$

Infine, terminiamo con l'operazione di **chiusura di Kleene**, detta anche **STAR**. Questa operazione è molto simile alla potenza, ma in questo caso il numero k non è fissato e quindi

questa operazione di potenza viene ripetuta all'infinito. Vengono quindi concatenate un numero arbitrario di stringhe di L , e teniamo tutte le computazioni intermedie, ovvero

$$L^* = \bigcup_{k \geq 0} L^k.$$

Ecco perché scriviamo Σ^* : partendo dall'alfabeto Σ andiamo ad ottenere ogni stringa possibile.

Esiste anche la **chiusura positiva**, definita come

$$L^+ = \bigcup_{k \geq 1} L^k.$$

Che relazione abbiamo tra le due chiusure? Questo dipende da ε , ovvero:

- se $\varepsilon \in L$ allora $L^* = L^+$ perché $L^1 \subseteq L^+$ e visto che $\varepsilon \in L^1$ abbiamo gli stessi insiemi;
- se $\varepsilon \notin L$ allora $L^+ = L^*/\{\varepsilon\}$ perché l'unico modo di ottenere ε sarebbe con L^0 .

Esempio 6.2.2: Vediamo una cosa simpatica:

$$\mathbb{Q}^* = \{\varepsilon\}.$$

Abbiamo appena generato qualcosa dal nulla, fuori di testa. La generazione si blocca con la chiusura positiva, ovvero

$$\mathbb{Q}^+ = \mathbb{Q}.$$

Infine, vediamo una cosa abbastanza banale sull'insieme formato dalla sola ε , ovvero

$$\{\varepsilon\}^* = \{\varepsilon\}^+ = \{\varepsilon\}.$$

Con la chiusura di Kleene, partendo da un **linguaggio finito** L , otteniamo una chiusura L^* di cardinalità infinita, perché ogni volta andiamo a creare delle nuove stringhe.

Partendo invece da un **linguaggio infinito** L , otteniamo ancora una chiusura L^* di cardinalità infinita ma ci sono alcune situazioni particolari.

Esempio 6.2.3: Vediamo tre linguaggi infiniti che hanno comportamenti diversi.

Consideriamo il linguaggio

$$L_1 = \{a^n \mid n \geq 0\}.$$

Calcolando la chiusura L_1^* otteniamo lo stesso linguaggio L_1 perché stiamo concatenando stringhe che contengono solo a , che erano già presenti in L_1 .

Consideriamo ora il linguaggio

$$L_2 = \{a^{2k+1} \mid k \geq 0\}.$$

Calcolando la chiusura L_2^* otteniamo il linguaggio L_1 perché:

- in L_2^1 ho tutte le stringhe formate da a di lunghezza dispari;
- in L_2^2 ho tutte le stringhe formate da a di lunghezza pari (*dispari + dispari*).

Già solo con $L_2^0 \cup L_2^1 \cup L_2^2$ generiamo tutto il linguaggio L_1

Consideriamo infine

$$L_3 = \{a^n b \mid n \geq 0\}.$$

Proviamo a calcolare prima la potenza L_3^k di questo linguaggio, ovvero

$$L_3^k = \{a^{n_1} b \dots a^{n_k} b \mid n_1, \dots, n_k \geq 0\}.$$

La chiusura L_3^* conterrà stringhe in questa forma con k ogni volta variabili.

Abbiamo quindi visto diverse situazioni: nel primo linguaggio non abbiamo dovuto calcolare nessuna chiusura, nel secondo linguaggio abbiamo calcolato un paio di linguaggi, nel terzo linguaggio non ci siamo mai fermati.

7. Espressioni regolari

7.1. Teorema di Kleene

Con le operazioni che abbiamo visto possiamo creare dei nuovi linguaggi. Tra queste operazioni, possiamo raggruppare **unione**, **prodotto** e **chiusura di Kleene** sotto il cappello delle **operazioni regolari**. Come mai questo nome? Perché esse sono usate per definire i **linguaggi regolari**.

Vediamo tre versioni del seguente teorema, ma ci interesseremo solo della prima e della terza.

Teorema 7.1.1 (*Teorema di Kleene*): La classe dei linguaggi accettati da automi a stati finiti coincide con la più piccola classe contenente i linguaggi

$$\emptyset \quad | \quad \{\varepsilon\} \quad | \quad \{a\}$$

e chiusa rispetto alle operazioni di unione, prodotto e chiusura di Kleene.

Questa prima versione ci dice che possiamo costruire la classe dei linguaggi regolari partendo da tre linguaggi base e applicando in tutti i modi possibili le tre operazioni regolari.

Teorema 7.1.2 (*Teorema di Kleene*): La classe dei linguaggi accettati da automi a stati finiti coincide con la più piccola classe che contiene i linguaggi finiti.

Seconda versione carina, ma che non ci interessa.

Teorema 7.1.3 (*Teorema di Kleene*): La classe dei linguaggi accettati da automi a stati finiti coincide con la classe dei linguaggi espressi con le espressioni regolari.

Bella anche questa versione, ma cosa sono le **espressioni regolari**?

Simbolo/espressione	Linguaggio associato
\emptyset	\emptyset
ε	$\{\varepsilon\}$
a	$\{a\}$
$E_1 + E_2$	$L(E_1) \cup L(E_2)$
$E_1 \cdot E_2$	$L(E_1) \cdot L(E_2)$
E^*	$(L(E))^*$

Le **espressioni regolari** sono una **forma dichiarativa** usata per definire i linguaggi, ovvero grazie ad esse dichiariamo come sono fatte le stringhe di un certo linguaggio. Fin'ora avevamo usato gli automi (**forma riconoscitiva**) e le grammatiche (**forma generativa**).

Esempio 7.1.1: Vediamo un po' di espressioni regolari per alcuni linguaggi.

Linguaggio	Espressione regolare
$L = \{a^n \mid n \geq 0\}$	a^*
$L = \{a^{2k+1} \mid k \geq 0\}$	$a(aa)^* = (aa)^*a$
$L = \{a^n b \mid n \geq 0\}$	a^*b
$L = \{(a^n b)^k \mid n \geq 0 \wedge k > 0\}$	$(a^*b)^k$
L_3 terzultimo simbolo da destra è una a	$(a + b)^*a(a + b)(a + b)$

Il penultimo linguaggio ha una espressione regolare che non abbiamo visto: si tratta di una piccola **estensione algebrica** che ci permette di unire assieme una serie di fattori identici.

Andiamo ora a dimostrare il Teorema 7.1.3 nella sua forma con le espressioni regolari.

Dimostrazione 7.1.3.1: Dobbiamo mostrare una doppia implicazione.

[Automa \rightarrow RegExp]

Dobbiamo far vedere che, dato un automa per il linguaggio L , possiamo costruire una operazione regolare che denota esattamente L .

La «dimostrazione» si trova nell'Esempio 7.2.1 del Paragrafo 7.2.

[RegExp \rightarrow Automa]

Dobbiamo far vedere che, data una espressione regolare che denota un linguaggio L , possiamo costruire un automa A che riconosce esattamente L .

La dimostrazione si trova nel Paragrafo 7.4. ■

7.2. Da automa ad espressione regolare

Vediamo un esempio di come passare da un **automa** ad una **espressione regolare**.

Esempio 7.2.1: Per ricavare una espressione regolare da un automa si può usare un algoritmo di **programmazione dinamica** molto simile all'algoritmo Floyd-Warshall sui grafi, che cerca i cammini minimi imponendo una serie di vincoli.

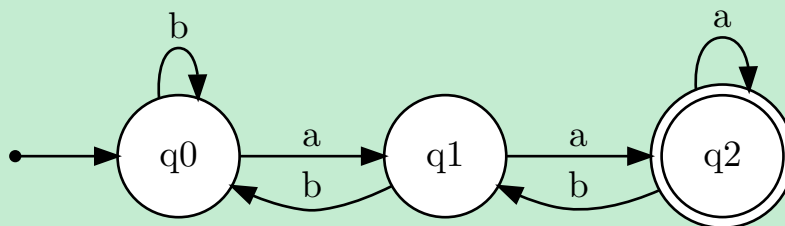
Un altro approccio invece cerca di risolvere un **sistema di equazioni** associato all'automa.

Dato un automa, costruiamo un sistema di n equazioni, dove n è il numero di stati dell'automa. Supponendo di numerare gli stati da 1 a n , la i -esima equazione descrive i cambiamenti di stato che possono avvenire partendo dallo stato i .

Ogni **cambiamento di stato** è nella forma aB , dove a è il carattere che causa una transizione e B è lo stato di arrivo. Tutti i cambiamenti di stato a partire da i vanno sommati tra loro. Inoltre, se lo stato i -esimo è uno stato finale si aggiunge anche ε all'equazione.

Questa somma di cambiamenti di stati va posta uguale allo stato i -esimo.

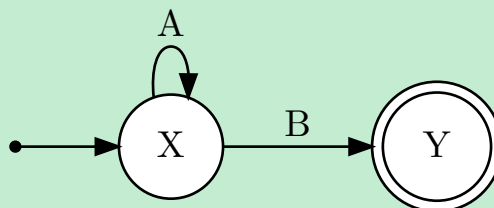
Vediamo con un esempio pratico. Ci viene dato il seguente automa.



Associamo ad esso un sistema di 3 equazioni, nel quale indichiamo gli stati con le variabili X_i e i caratteri sono quelli dell'alfabeto $\{a, b\}$. Il sistema è il seguente:

$$\begin{cases} X_0 = aX_1 + bX_0 \\ X_1 = aX_2 + bX_0 \\ X_2 = aX_2 + bX_1 + \varepsilon \end{cases}.$$

Ora dobbiamo risolvere questo sistema di equazioni. Per fare ciò, dobbiamo introdurre una **regola fondamentale** che ci permetterà di risolvere tutti i sistemi che vedremo.



Il sistema di equazioni per questo automa è

$$\begin{cases} X = AX + BY \\ Y = \varepsilon \end{cases}.$$

Sostituendo $Y = \varepsilon$ nella prima equazione otteniamo

$$X = AX + B.$$

L'espressione regolare per questo automa è

$$A^*B.$$

Visto che le due cose che abbiamo scritto devono essere identiche, ogni volta che abbiamo una equazione nella forma

$$X = AX + B$$

la possiamo sostituire con l'equazione

$$X = A^*B.$$

Riprendiamo il sistema dell'automa dell'esempio e andiamo a risolvere le nostre equazioni:

$$\begin{cases} X_0 = a(aX_2 + bX_0) + bX_0 \\ X_2 = aX_2 + b(aX_2 + bX_0) + \varepsilon \end{cases}$$

$$\begin{cases} X_0 = aaX_2 + abX_0 + bX_0 \\ X_2 = aX_2 + baX_2 + bbX_0 + \varepsilon \end{cases}$$

$$\begin{cases} X_0 = (ab + b)X_0 + aaX_2 \\ X_2 = (a + ba)X_2 + bbX_0 + \varepsilon \end{cases}$$

$$\begin{cases} X_0 = (ab + b)X_0 + aaX_2 \\ X_2 = (a + ba)^*(bbX_0 + \varepsilon) \end{cases}$$

$$X_0 = (ab + b)X_0 + aa((a + ba)^*(bbX_0 + \varepsilon))$$

$$X_0 = (ab + b)X_0 + aa(a + ba)^*bbX_0 + aa(a + ba)^*$$

$$X_0 = (ab + b + aa(a + ba)^*bb)X_0 + aa(a + ba)^*.$$

Applicando un'ultima volta la regola fondamentale otteniamo l'espressione regolare

$$(ab + b + aa(a + ba)^*bb)^*aa(a + ba)^*.$$

E pensare che l'algoritmo basato su Floyd-Warshall è anche più difficile...

Con questo esempio siamo riusciti a «dimostrare» la trasformazione da **automa** ad **espressione regolare**, anche se non è una vera e propria dimostrazione.

7.3. State complexity

Nella seconda parte della dimostrazione, ovvero la trasformazione da **espressioni regolari** ad **automi**, che troviamo nel [Paragrafo 7.4](#), vogliamo studiare anche il numero di stati che sono **necessari** per definire un automa. Vediamo due quantità che sono chiave in questo studio.

Definizione 7.3.1 (*State complexity deterministica*): Sia $L \subseteq \Sigma^*$. Indichiamo con

$$\text{sc}(L)$$

il **minimo numero di stati** di un DFA completo per L .

Abbiamo poi visto che l'automa con questo numero di stati è anche **unico**.

Definizione 7.3.2 (*State complexity non deterministica*): Sia $L \subseteq \Sigma^*$. Indichiamo con

$$\text{nsc}(L)$$

il **minimo numero di stati** di un NFA per L .

In questo caso abbiamo visto che l'NFA minimo **non è unico**. Inoltre, non abbiamo la nozione di **completo** perché la funzione di transizione associa ad ogni passo di computazione una serie di scelte, che può essere anche la scelta vuota.

Vediamo una prima applicazione di queste due quantità.

Lemma 7.3.1: Se L non è un linguaggio regolare allora

$$\text{sc}(L) = \text{nsc}(L) = \infty.$$

Lemma 7.3.2: Se L è un linguaggio regolare allora

$$\text{sc}(L) < \infty \wedge \text{nsc}(L) < \infty.$$

Avevamo inoltre il bound per passare da NFA a DFA, che nel caso peggiore trasformava n stati di un NFA in 2^n stati di un DFA con l'automa di **Meyer-Fischer**.

Esempio 7.3.1: Sia L_n il classico linguaggio dell' n -esimo simbolo da destra uguale ad a .

Avevamo visto un NFA che utilizzava $n + 1$ stati, quindi

$$\text{nsc}(L_n) \leq n + 1.$$

Si dimostra poi l'uguaglianza dei due valori utilizzando un fooling set.

Avevamo poi visto un DFA che utilizzava 2^n stati, quindi

$$\text{sc}(L_n) = 2^n.$$

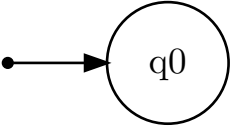
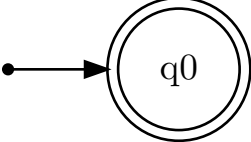
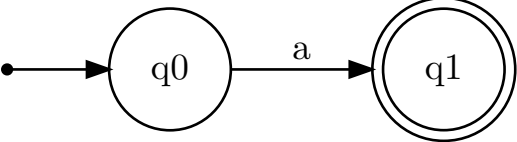
Con un insieme di stringhe distinguibili avevamo mostrato che servivano almeno 2^n stati, ma con la realizzazione effettiva abbiamo uguagliato il bound.

7.4. Da espressione regolare ad automa

Per dimostrare questa parte dell'implicazione dobbiamo costruire un automa per ogni espressione regolare possibile, quindi per le tre espressioni base e per le tre operazioni regolari. In realtà, in questa parte vedremo gli automi di un po' tutte le operazioni che abbiamo visto per adesso, studiando anche la complessità che otteniamo in termini di stati.

7.4.1. Espressioni base

Partiamo con le tre **espressioni regolari base**, che possiamo riconoscere con gli automi presenti nella Tabella 1.

Espressione regolare	Automa
\emptyset	
ε	
a	

Per essere precisi, dovremmo utilizzare dei **DFA completi**, quindi dovremmo considerare anche lo **stato trappola**. In questo caso vogliamo solo fare un **conto asintotico** quindi questo non ci interessa molto, ma se volessimo il numero preciso di stati allora quello stato diventa necessario.

7.4.2. Complemento

Dato il linguaggio L con $\text{sc}(L) = n$, vogliamo valutare la quantità $\text{sc}(L^C)$ del **complemento** di L .

7.4.2.1. DFA

Se abbiamo un DFA per L , passare a L^C è molto facile: tutte le stringhe che prima accettavo ora le devo rifiutare e viceversa. Parlando in termini di dell'automa, invertiamo ogni stato finale in non finale e viceversa, mantenendo intatte le transizioni.

Dato $A = (Q, \Sigma, \delta, q_0, F)$ un DFA per L , costruisco l'automa $A' = (Q, \Sigma, \delta, q_0, F')$ un DFA per L^C tale che

$$F' = Q/F.$$

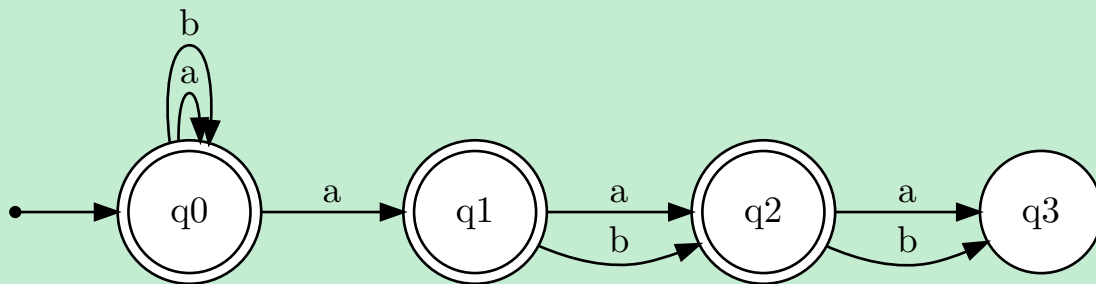
Dobbiamo imporre che A sia **completo** perché ciò che andava nello stato trappola ora deve essere accettato. Ma allora

$$\text{sc}(L^C) = \text{sc}(L).$$

7.4.2.2. NFA

Come ci comportiamo sugli NFA?

Esempio 7.4.2.2.1: Sia L_3 l'istanza del linguaggio L_n solito con $n = 3$. Andiamo a vedere un automa che cerca di calcolare L_3^C con la tecnica che abbiamo appena visto nei DFA.



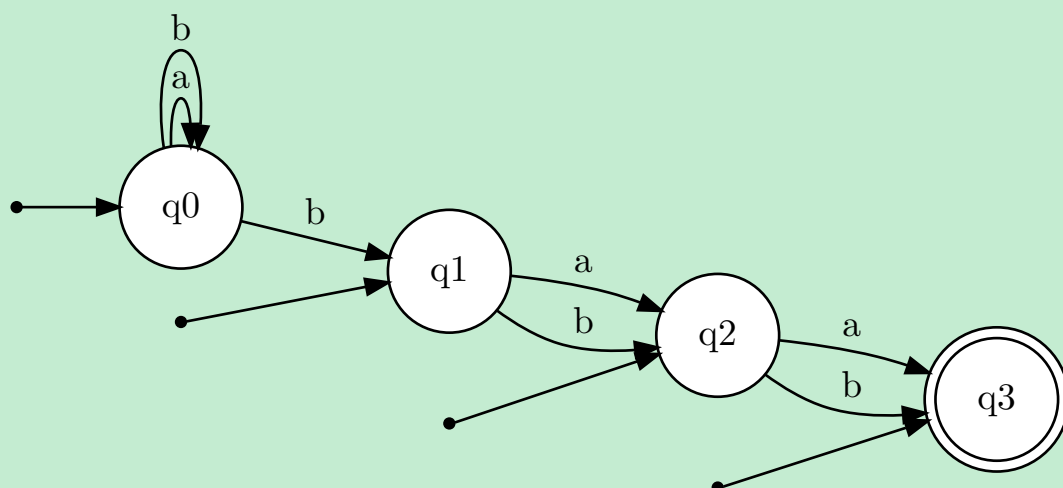
Abbiamo un problema: questo automa **accetta tutto**. Ma perché succede questo? Negli NFA accettiamo se esiste almeno un cammino accettante e rifiutiamo se ogni cammino è rifiutante. Quando accettiamo è molto probabile che ci sia, oltre al cammino accettante, anche qualche cammino rifiutante. Facendo il complemento, accettiamo ancora quando abbiamo almeno un cammino accettante, ma questo deriva da uno dei cammini rifiutanti precedenti.

È importantissimo avere il DFA, per via di questa **asimmetria** tra accettazione e non accettazione.

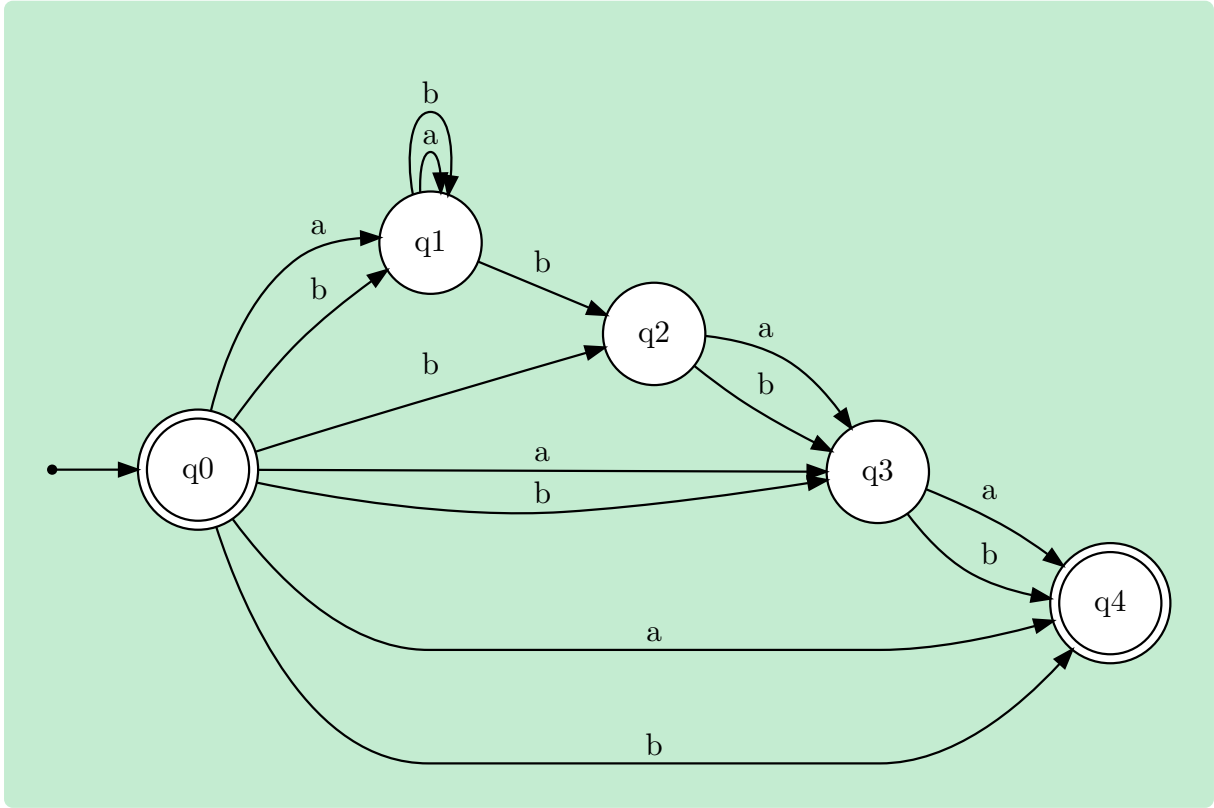
Ma se volessimo per forza un NFA per il complemento? Questo va molto a caso, dipende da linguaggio a linguaggio, potrebbe essere molto facile da trovare come molto difficile.

Esempio 7.4.2.2.2: Sempre per il linguaggio L_3 , diamo due NFA per riconoscere L_3^C .

Una prima soluzione utilizza una serie di stati iniziali multipli.



Una seconda soluzione utilizza invece il non determinismo puro.



Questo approccio di cercare a tutti i costi un NFA può essere difficoltoso.

7.4.2.3. Costruzione per sottoinsiemi

Vediamo un algoritmo che ci permette di avere un automa deterministico per L^C .

Sia $A = (Q, \Sigma, \delta, q_0, F)$ un NFA per L . Vogliamo costruire un automa per il linguaggio L^C . Un modo sistematico e ottimo per avere un DFA sotto mano è passare al DFA di A e poi eseguire la costruzione del complemento che abbiamo visto prima.

Quanti stati abbiamo? Sappiamo che abbiamo un salto esponenziale passando dall'NFA al DFA, e poi uno stesso numero di stati, quindi

$$\text{nsc}(L^C) \leq 2^{\text{nsc}(L)}.$$

Possiamo fare di meglio? Sicuramente esistono esempi di salti che non sono esattamente esponenziali, come i linguaggi delle coppie di elementi uguali/diversi a distanza n , che avevano un salto del tipo

$$2n + 2 \longrightarrow 2^n,$$

ma si può costruire un esempio che faccia un salto esponenziale perfetto.

Abbiamo quindi visto che del complemento negli NFA non ce ne facciamo niente, questo proprio per la natura asimmetrica del non determinismo.

7.4.3. Unione

Dati due linguaggi $L', L'' \subseteq \Sigma^*$ rispettivamente riconosciuti dagli automi $A' = (Q', \Sigma, \delta', q'_0, F')$ e $A'' = (Q'', \Sigma, \delta'', q''_0, F'')$, vogliamo costruire un automa per l'**unione**

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

Per risolvere questo problema pensiamo agli automi come se fossero delle scatole, che prendono l'input nello stato iniziale e poi arrivano alla fine nell'insieme degli stati finali.



L'idea per costruire l'automa l'unione è combinare i due automi A' e A'' usando il non determinismo per scegliere in quale automa finire con una ε -mossa.



Visto che il linguaggio dell'unione deve stare in almeno uno dei due, metto una scommessa all'inizio per vedere se andare nel primo o nel secondo automa. Bella soluzione, funziona, ma non ci piace tanto, come mai?

7.4.3.1. DFA

Non ci piace tanto questa soluzione perché se partiamo da **due DFA** andiamo a finire in un **NFA**. Infatti, la componente non deterministica viene inserita con le due ε -mosse iniziali. La stessa componente non deterministica l'avremmo inserita con gli stati iniziali multipli, che sarebbero stati in corrispondenza dei due stati iniziali q'_0 e q''_0 senza lo stato q_0 .

Se vogliamo rimanere nel mondo DFA dobbiamo unire i due automi con questa costruzione e poi passare al DFA con la costruzione per sottoinsiemi. Il numero di stati dell'NFA è

$$\text{nsc}(L' \cup L'') \leq 1 + \text{nsc}(L') + \text{nsc}(L''),$$

quindi con la costruzione per sottoinsiemi arriveremmo ad avere un numero di stati pari a

$$\text{sc}(L' \cup L'') \leq 2^{\text{nsc}(L' \cup L'')}.$$

Questa costruzione è altamente **inefficiente**. Si può fare molto meglio.

7.4.3.2. Automa prodotto

Utilizzando una costruzione particolare, la **costruzione dell'automa prodotto**, siamo in grado di **abbassare di brutto** la complessità in stati dei DFA per l'unione di linguaggi.

L'**automa prodotto** fa partire in parallelo i due automi, e alla fine controlla che almeno uno dei due abbia dato un cammino accettante. Definiamo quindi $A = (Q, \Sigma, \delta, q_0, F)$ tale che:

- gli **stati** rappresentano i due automi che viaggiano in parallelo, come se avessi due pc davanti, ognuno che lavora da solo. Gli stati sono quindi l'insieme

$$Q = Q' \times Q'';$$

- lo **stato iniziale** è la coppia di stati iniziali, ovvero

$$q_0 = (q'_0, q''_0);$$

- la **funzione di transizione** lavora ora sulle coppie di stati, che deve portare avanti in parallelo, quindi

$$\delta((q, p), a) = (\delta'(q, a), \delta''(p, a));$$

- gli **stati finali** sono tutte le coppie dove riesco a finire in almeno uno stato finale, ovvero

$$F = \{(q, p) \mid q \in F' \vee p \in F''\}.$$

Come cambia la complessità dell'automa rispetto alla costruzione per sottoinsiemi? Qua il numero di stati è

$$\text{sc}(L' \cup L'') \leq \text{sc}(L') \cdot \text{sc}(L''),$$

quindi abbiamo una soluzione notevolmente migliore. Inoltre, non si può fare meglio di così.

Esempio 7.4.3.2.1: Fissati due valori m, n positivi, definiamo i linguaggi

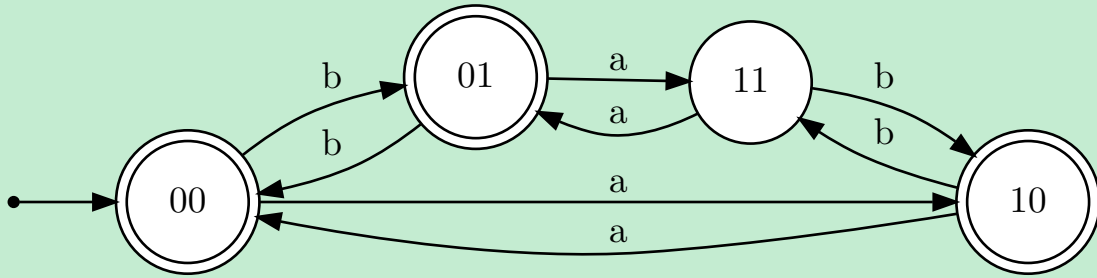
$$L' = \{x \in \{a, b\}^* \mid \#_a(x) \text{ è multiplo di } m\}$$

$$L'' = \{x \in \{a, b\}^* \mid \#_b(x) \text{ è multiplo di } n\}.$$

I due automi A' e A'' per L' e L'' sono molto semplici, devono solo contare il numero di a e b . Vediamo un esempio con $m = n = 2$.



Costruiamo l'automa prodotto per il linguaggio $L = L' \cup L''$.



7.4.3.3. NFA

Negli NFA non abbiamo nessun problema: partiamo da NFA e vogliamo restare in NFA, quindi non servono ulteriori costruzioni per avere un automa di questa classe. Il numero di stati è

$$\text{nsc}(L' \cup L'') \leq 1 + \text{nsc}(L') + \text{nsc}(L'').$$

Perdiamo il termine noto di questa quantità se non usiamo ε -mosse ma stati iniziali multipli.

7.4.4. Intersezione

Per l'**intersezione** di linguaggi non dobbiamo definire molto di nuovo.

Per i **DFA** possiamo utilizzare la costruzione dell'automa prodotto appena definita modificando l'insieme degli stati finali F rendendolo

$$F = \{(q, p) \mid q \in F' \wedge p \in F''\}.$$

Ma allora la state complexity vale

$$\text{sc}(L' \cap L'') \leq \text{sc}(L') \cdot \text{sc}(L'').$$

Esempio 7.4.4.1: Riprendendo i due linguaggi di prima, l'automa prodotto viene costruito nello stesso modo, ma cambia l'insieme degli stati finali, che si riduce al singleton $\{00\}$.



Per gli **NFA** possiamo riutilizzare la costruzione dell'automa prodotto per permetterci di navigare tutte le possibili coppie di cammini, e scommettendo bene su entrambi i cammini possiamo accettare la stringa. Va sistemata un pelo la definizione della funzione di transizione, ma la costruzione rimane uguale. Vale quindi

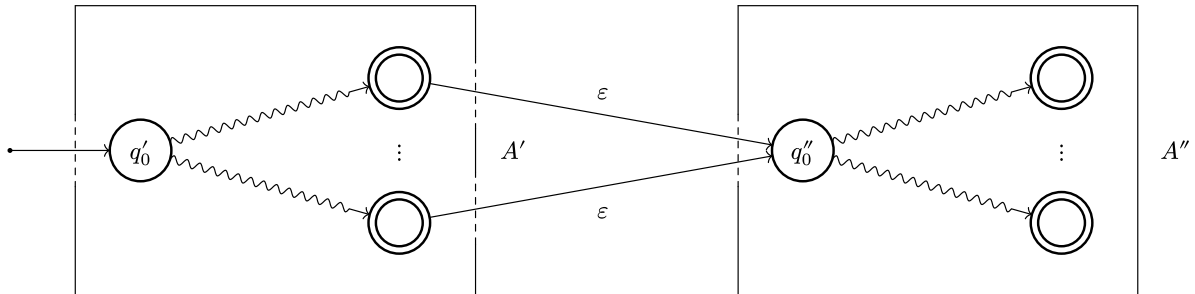
$$\text{nsc}(L' \cap L'') \leq \text{nsc}(L') \cdot \text{nsc}(L'').$$

7.4.5. Prodotto

Riprendiamo velocemente la definizione di **prodotto** di linguaggi. Dati due linguaggi L' e L'' , allora

$$L' \cdot L'' = \{w \mid \exists x \in L' \wedge \exists y \in L'' \mid w = xy\}.$$

Un possibile **automa** per il prodotto fa partire l'automa A' per L' , e quando si trova in uno stato finale fa un salto nell'automa A'' per L'' e lo fa partire.



In poche parole, ogni volta che arriviamo in uno stato finale di A' facciamo partire la computazione su A'' , ma in A' andiamo avanti a scandire la stringa. Stiamo scommettendo di essere arrivati alla fine della stringa x e di dover iniziare a leggere la stringa y .

Bella costruzione, ma ci va veramente bene una roba del genere?

7.4.5.1. DFA

La risposta, come prima, è **NO**: se partiamo da due DFA andiamo a finire in un NFA, che non ci va bene perché per poi tornare in un DFA ci costa un salto esponenziale. Visto che

$$\text{nsc}(L' \cdot L'') = \text{nsc}(L') + \text{nsc}(L''),$$

possiamo dire che

$$\text{sc}(L' \cdot L'') \leq 2^{\text{nsc}(L' \cdot L'')}.$$

Come prima, possiamo ottimizzare questa costruzione, anche se non di molto stavolta.

7.4.5.2. Costruzione senza nome

Il problema dell'esplosione del doppio esponenziale deriva dal fatto che, quando arrivo in uno stato finale del primo automa, devo far partire il secondo automa, ma il primo continua ancora a scandagliare la stringa perché deve scommettere.

La soluzione inefficiente di prima prendeva i due automi A' e A'' , li univa in un NFA ed effettuava la costruzione per sottoinsiemi. La soluzione che facciamo adesso **incorpora** i sottoinsiemi nei passi del DFA, così da evitare l'esecuzione non deterministica.

Costruiamo l'automa $A = (Q, \Sigma, \delta, q_0, F)$ che, ogni volta che A' finisce in uno stato finale, avvia anche A'' dal punto nel quale si trova. Esso è definito da:

- gli **stati** sono tutte le coppie di stati di A' con i sottoinsiemi di A'' , così da incorporare i sottoinsiemi nel DFA direttamente, ovvero

$$Q = Q' \times 2^{Q''};$$

- lo **stato iniziale** dipende se siamo già in una configurazione che permette lo start di A'' , ovvero

$$q_0 = \begin{cases} (q'_0, \emptyset) & \text{se } q'_0 \notin F' \\ (q'_0, \{q''_0\}) & \text{se } q'_0 \in F' \end{cases};$$

- la **funzione di transizione** deve lavorare sulla prima componente ma anche su tutte quelle presenti nella seconda componente, quindi essa è definita come

$$\delta((q, \alpha), a) = \begin{cases} (\delta'(q, a), \{\delta''(p, a) \mid p \in \alpha\}) & \text{se } \delta'(q, a) \notin F' \\ (\delta'(q, a), \{\delta''(p, a) \mid p \in \alpha\} \cup \{q''_0\}) & \text{se } \delta'(q, a) \in F' \end{cases};$$

- gli **stati finali** sono quelli nei quali riusciamo ad arrivare con il secondo automa, ovvero

$$F = \{(q, \alpha) \mid \alpha \cap F'' \neq \emptyset\}.$$

La prima componente la mandiamo avanti deterministicamente, ma la manteniamo sempre accesa per far partire la seconda computazione. Quest'ultima è anch'essa deterministica, ma simula un po' il comportamento non deterministico.

Il numero di stati massimo che abbiamo è

$$\text{sc}(L' \cdot L'') = \text{sc}(L') 2^{\text{sc}(L'')},$$

che rappresenta comunque un gap esponenziale ma abbiamo abbassato di un po' la complessità.

7.4.5.3. NFA

Come per l'unione, qua siamo molto tranquilli: partiamo da NFA e arriviamo in NFA, quindi a noi va tutto bene. La state complexity, come visto prima, è

$$\text{nsc}(L' \cdot L'') \leq \text{nsc}(L') + \text{nsc}(L'').$$

7.4.6. Chiusura di Kleene

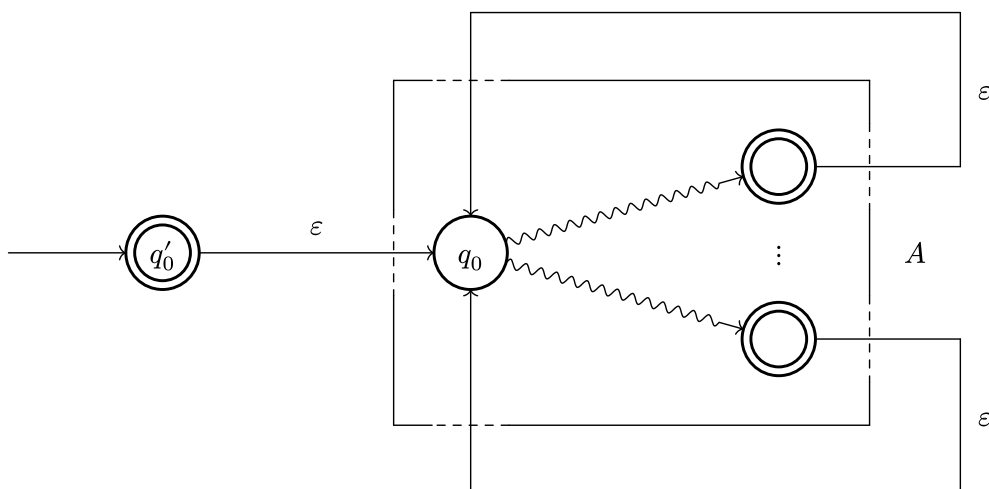
Con questa ultima operazione chiudiamo la dimostrazione del [Teorema 7.1.3](#).

Questa operazione la possiamo definire come

$$L^* = \bigcup_{k \geq 0} L^k = \{w \in \Sigma^* \mid \exists x_1, \dots, x_k \in L \mid k \geq 0 \mid w = x_1 \dots x_k\}.$$

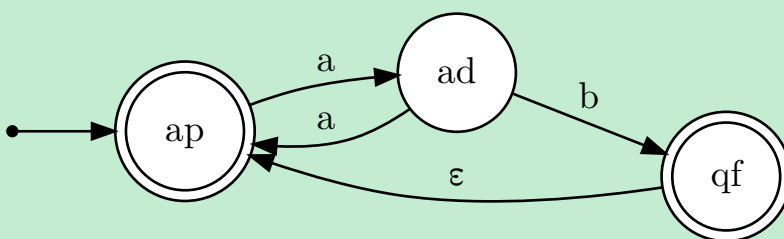
Un automa per la **star** deve cercare di scomporre la stringa in ingresso in più stringhe di L . Possiamo prendere spunto dall'automa per la concatenazione: facciamo partire l'automa per L

e ogni volta che arriviamo in uno stato finale lo facciamo ripartire dallo stato iniziale. Devo accettare anche la parola vuota, quindi aggiungiamo uno stato iniziale finale.



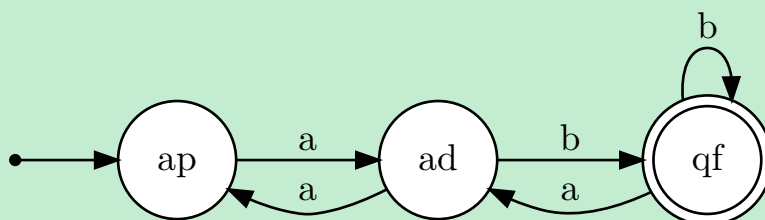
Abbiamo bisogno dello stato iniziale q'_0 perché dentro l'automa ci possono essere delle transizioni che mi fanno ritornare indietro e mi fanno accettare di più di quello che dovrei.

Esempio 7.4.6.1: Consideriamo il seguente automa **sbagliato** per la chiusura di Kleene del linguaggio delle sequenze dispari di a seguite da una b . Se vogliamo l'espressione regolare per questo linguaggio, essa è $(aa)^*ab$.



Non abbiamo inserito uno stato iniziale aggiuntivo. Che succede? La stringa aa adesso viene accettata, anche se palesemente non appartiene al linguaggio, così come la stringa $abaa$.

Un automa per la star di questo linguaggio, inoltre, è molto facile da trovare perché la lettera b fa da marcatore, e il numero di stati rimane quello dell'automa di partenza.



7.4.6.1. DFA

Ci piace questa soluzione, perché stiamo aumentando solo di uno il numero di stati dell'automa, ma siamo caduti nel non determinismo, e partire da DFA e finire in NFA non ci piace molto.

Purtroppo, come spesso ci succede, per tornare nei DFA dobbiamo, nel caso peggiore, applicare la costruzione per sottoinsiemi e fare un salto esponenziale nel numero degli stati. In poche parole

$$\text{sc}(L^*) \leq 2^{\text{nsc}(L^*)}.$$

7.4.6.2. NFA

Come solito, l'aggiunta di non determinismo agli NFA a noi non cambia, anzi ci piace, anche perché in questo caso l'automa lo otteniamo in modo semplice aggiungendo solo uno stato, ovvero

$$\text{nsc}(L^*) \leq \text{nsc}(L) + 1.$$

7.5. Codici

Abbiamo visto il linguaggio dell'Esempio 7.4.6.1, che era estremamente comodo da scomporre per via della presenza di una sola b nella stringa in input. Vediamo ora qualche altro esempio notevole.

Esempio 7.5.1: Definiamo il linguaggio $L = aaaba^*$. Vogliamo calcolare L^* .

Questo linguaggio è «facilmente» scomponibile: ogni volta che troviamo una b torniamo indietro di 3 caratteri e dividiamo la stringa in quel punto.

Ad esempio, la stringa

aaabaaaaaaabaabaaaab

viene suddivisa nel seguente modo:

aaabaaaa | aaab | aaaba | aaab.

L'automa comunque esegue un sacco di test non deterministici ogni volta che legge delle a dopo una b , perché rimaniamo sempre in uno stato finale.

Esempio 7.5.2: Definiamo ora il linguaggio $L = a(b + baab)a^*$. Vogliamo calcolare L^* .

Questo linguaggio invece è più difficile da scomporre. Ad esempio, data la stringa

abaabaaaaabaaba

essa la possiamo dividere in

aba | abaaaa | abaaba

oppure la possiamo dividere in

abaabaaaa | abaaba.

Per questa stringa abbiamo già due modi di scomposizione possibili.

Cambiamo la stringa: data la stringa

abaabaabaabaaaaba

essa la possiamo dividere in

$$aba \mid aba \mid aba \mid abaa \mid aba$$

oppure la possiamo dividere in

$$abaaba \mid abaabaa \mid aba.$$

In generale, si possono creare delle stringhe che hanno un numero di suddivisioni enorme.

A cosa servono questi esempi che abbiamo appena introdotto?

Definizione 7.5.1 (Codice): Dato $X \subseteq \Sigma^*$, diciamo che X è un **codice** se e solo se

$$\forall w \in X^* \quad \exists! \text{ decomposizione di } w \text{ come } x_1 \dots x_k \mid x_i \in L \wedge k \geq 0.$$

Dei tre linguaggi che abbiamo visto, quello dell'Esempio 7.4.6.1 e quello dell'Esempio 7.5.1 sono **codici**: è facile dimostrare che lo sono, soprattutto il primo per via della b che fa da delimitatore. L'Esempio 7.5.2, invece, abbiamo visto che ha un linguaggio con stringhe scomponibili in più modi, quindi non è un codice.

Tra tutti i codici a noi interessano quelli che possono essere **decomposti in tempo reale**: essi sono chiamati **codici prefissi**, e sono dei codici tali che

$$\forall i \neq j \quad x_i \text{ non è prefisso di } x_j.$$

In poche parole, il codice contiene parole che **non** sono prefisse di altre. Essi sono i più **efficienti**.

Dei due codici che abbiamo a disposizione, quello dell'Esempio 7.4.6.1 è un **codice prefisso**: ogni volta che troviamo una b sappiamo che dobbiamo dividere. Quello dell'Esempio 7.5.1, invece, deve aspettare una b e poi tornare indietro di 3 posizioni per avere la decomposizione.

7.6. Star height

Per definire le espressioni regolari abbiamo a disposizione le tre operazioni

$$+ \mid \cdot \mid ()^*$$

Concentriamoci un secondo sulla chiusura di Kleene e vediamo un esempio.

Esempio 7.6.1: Date le espressioni regolari

$$(a^*b^*)^*$$

$$(a + b)^*$$

$$(ab^*)^*$$

ci chiediamo che linguaggio stanno denotando. Questo è facile: $\{a, b\}^*$. Ognuna lo fa a modo proprio, in base a come ha risolto il sistema delle equazioni dell'automa associato.

Nell'esempio abbiamo visto diverse espressioni per lo stesso linguaggio, ma quante star mi servono per definire completamente un linguaggio?

Definizione 7.6.1 (*Star height*): La **star height** è il massimo numero di star innestate in una espressione regolare. Possiamo definire la quantità induttivamente, ovvero

$$\begin{aligned} h(\emptyset) &= h(\varepsilon) = h(a) = 0 \\ h(E' + E'') &= h(E' \cdot E'') = \max\{h(E'), h(E'')\} \\ h(E^*) &= 1 + h(E). \end{aligned}$$

Nell'Esempio 7.6.1, le espressioni regolari hanno star height rispettivamente 2, 1 e 2. A noi piacerebbe scrivere un'espressione regolare con la **minima star height**.

Sia L un linguaggio regolare. Definiamo con $h(L)$ la **minima altezza** delle espressioni regolari che definiscono L , ovvero

$$h(L) = \min\{h(E) \mid L = L(E)\}.$$

Questa quantità, nei **linguaggi infiniti**, è almeno 1, ovvero non posso usare meno star.

Teorema 7.6.1 (*Un bro nel 1966*): Vale

$$\forall q > 0 \quad \exists W_q \subseteq \{a, b\}^* \mid h(W_q) = q.$$

Il linguaggio W_q è definito come

$$W_q = \{w \in \{a, b\}^* \mid \#_a(w) \equiv \#_b(w) \pmod{2^q}\}.$$

Esempio 7.6.2: Proviamo a disegnare W_q per $q = 1$.



La sua espressione regolare, dopo un po' di conti, è la seguente:

$$L = (a(bb)^*a + a(bb)^*ba(aa + ab(bb)^*ba)^*ab(bb)^*a)^*(a(bb)^*ba(aa + ab(bb)^*ba)^*ab(bb)^*b).$$

A quanto pare ho sbagliato a risolvere il sistema, sono scarso scusate.

Abbiamo quindi fatto vedere che se fissiamo il numero $q > 0$ di star che vogliamo usare in una espressione regolare, riusciamo a trovare un linguaggio W_q che usa quel numero di star. Sincero? È un **teorema abbastanza inutile**.

Teorema 7.6.2: Se $|\Sigma| = 1$ è sufficiente una star innestata per definire completamente L , ovvero vale che

$$\forall L \in \text{Reg} \mid L \subseteq \{a\}^* \quad h(L) \leq 1.$$

7.7. Espressioni regolari estese

Cosa succede se nelle espressioni regolari, oltre alle operazioni di unione, concatenazione e chiusura, utilizziamo anche le operazioni di **intersezione** e **complemento**?

Otteniamo quelle che sono dette **espressioni regolari estese**: sono molto potenti ma devono essere usate con cautela. Vediamo il perché con qualche esempio.

Esempio 7.7.1: Sia

$$L = \{w \in \{a, b\}^* \mid \#_a(w) \text{ pari} \wedge \#_b(w) \text{ pari}\}.$$

Se mi chiedono l'espressione regolare di questo linguaggio mi mandano a quel paese, ma usando le espressioni regolari estese questo è molto facile.

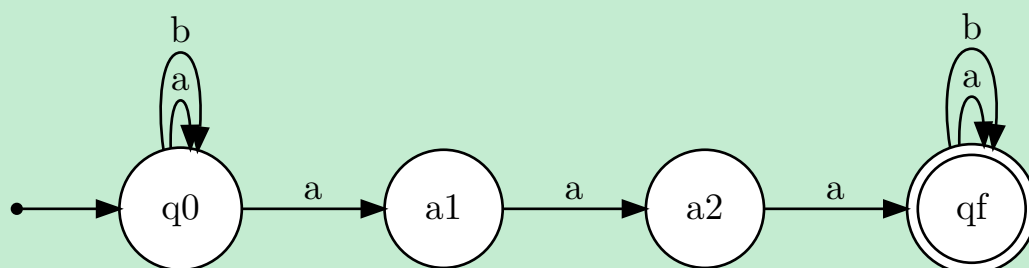
Per il linguaggio $L_a = \{w \in \{a, b\}^* \mid \#_a(w) \text{ pari}\}$ possiamo scrivere l'espressione regolare $(b + ab^*a)^*$, mentre per il linguaggio $L_b = \{w \in \{a, b\}^* \mid \#_b(w) \text{ pari}\}$ possiamo scrivere l'espressione regolare $(a + ba^*b)^*$.

Utilizzando l'intersezione possiamo banalmente concatenare le due espressioni, ovvero

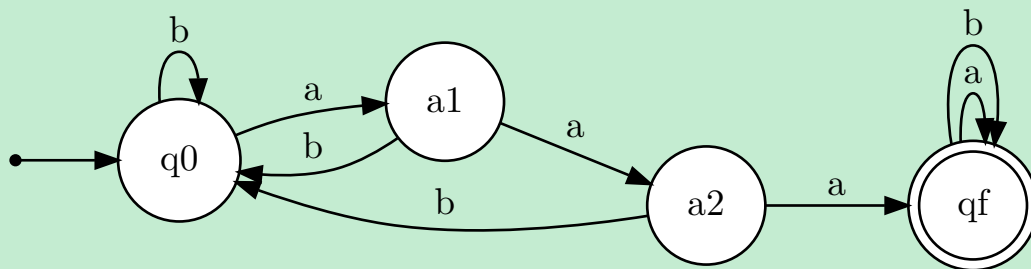
$$(b + ab^*a)^* \cap (a + ba^*b)^*.$$

Esempio 7.7.2: Vogliamo un'espressione regolare per le stringhe che **non** contengono tre a consecutive. Per fare ciò, costruiamo un automa e poi ricaviamo l'espressione regolare.

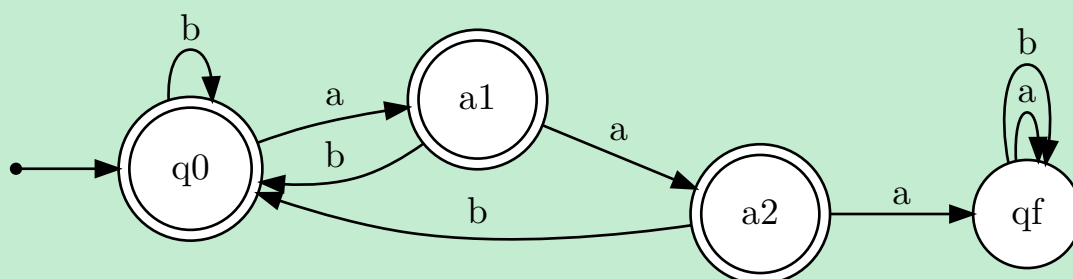
Partiamo da automi che accettano stringhe con tre a consecutive e poi complementiamo. In ordine, vediamo un NFA e un DFA per questo linguaggio di transizione.



Avevamo visto che il complemento non si comportava bene con gli NFA, quindi diamo un DFA, che invece funzionava bene con il complemento.



Siamo rimasti in un numero di stati accettabile. Andiamo a complementare questo DFA, che manterrà lo stesso numero di stati, fortunatamente.



Ora di questo dovrei farci l'espressione regolare. Sicuramente esce una bestiata enorme, con un po' di star qua e là visto che il linguaggio è infinito.

Con le espressioni regolari estese possiamo evitare l'uso delle star. Prima prendiamo tutte le stringhe che hanno tre a consecutive: esse sono nella forma

$$(a + b)^*aaa(a + b)^*.$$

Dobbiamo applicare il complemento a questa espressione per ottenere L , quindi

$$\overline{(a+b)^*aaa(a+b)^*}.$$

L'insieme di tutte le stringhe su un alfabeto lo possiamo vedere come il complemento dell'insieme vuoto rispetto all'insieme delle stringhe su quell'alfabeto, ovvero

$$(a + b)^* = \overline{\mathbb{Q}}.$$

Ma allora l'espressione regolare diventa

$$\overline{\overline{\mathbb{Q}}}aaa\overline{\overline{\mathbb{Q}}},$$

che come vediamo è un'espressione che non utilizza alcuna star.

Siamo stati in grado di **non usare le star** per un linguaggio infinito, cosa che nelle espressioni regolari classiche non è possibile. Bisogna stare attenti però: il complemento è molto comodo ma è anche molto insidioso perché fa saltare il numero di stati esponenzialmente se usiamo degli NFA.

Come nelle espressioni regolari, possiamo chiederci il **minimo numero di star** che sono necessarie per descrivere completamente un linguaggio.

Definizione 7.7.1 (*Star height generalizzata*): L'**altezza generalizzata**, o star height generalizzata, è il massimo numero di star innestate di una espressione regolare estesa. Possiamo definire la quantità induttivamente, ovvero

$$\begin{aligned} \text{gh}(\emptyset) &= \text{gh}(\varepsilon) = \text{gh}(a) = 0 \\ \text{gh}(E' + E'') &= \text{gh}(E' \cdot E'') = \text{gh}(E' \cap E'') = \max\{\text{gh}(E'), \text{gh}(E'')\} \\ \text{gh}(E^C) &= \text{gh}(E) \\ \text{gh}(E^*) &= 1 + \text{gh}(E). \end{aligned}$$

Come prima, dato un linguaggio L , possiamo definire la quantità

$$\text{gh}(L) = \min\{\text{gh}(E) \mid L = L(E)\}$$

come la **minima star height generalizzata** di tutte le espressioni regolari estese che generano L .

Le espressioni regolari estese sono molto comode, ma di queste non si sa quasi niente:

- si sa che esistono linguaggi di altezza 0, e l'abbiamo visto nell'[Esempio 7.7.2](#);
- si sa che esistono linguaggi di altezza 1;
- non si sa niente sui linguaggi di altezza almeno 2.

Quale è il cambio di marcia tra le espressioni regolari estese e quelle classiche? Il **complemento**: questa ci permette di dichiarare cosa non ci interessa, mentre nelle espressioni regolari classiche noi possiamo solo dire cosa vogliamo, avendo un **modello dichiarativo**.

8. Operazioni esotiche e avanzate

Riprendiamo il filo del [Capitolo 6](#) e andiamo avanti con alcune **operazioni avanzate**.

8.1. Reversal

Sia $L \subseteq \Sigma^*$ un linguaggio. Chiamiamo

$$L^R = \{w^R \mid w \in L\}$$

il linguaggio delle stringhe ottenute **ribaltando** tutte le stringhe di L , ovvero

$$w = a_1 \dots a_n \implies w^R = a_n \dots a_1.$$

Questa operazione sul linguaggio L viene detta **reversal**.

Lemma 8.1.1: L'operazione di reversal preserva la regolarità.

Dimostrazione 8.1.1.1: Dimostriamo questo lemma usando le espressioni regolari. Per fare ciò partiamo con il dare delle espressioni regolari per le espressioni base.

$$(\emptyset)^R = \emptyset$$

$$(\varepsilon)^R = \varepsilon$$

$$(a)^R = a$$

Ora vediamo le espressioni regolari induttive.

$$(E_1 + E_2)^R = E_1^R + E_2^R$$

$$(E_1 \cdot E_2)^R = E_2^R \cdot E_1^R$$

$$(E^*)^R = (E^R)^*$$

Queste espressioni sono ancora regolari, quindi il reversal preserva la regolarità. ■

Possiamo dimostrare questo lemma anche usando gli **automi**. Supponiamo di avere un DFA $A = (Q, \Sigma, \delta, q_0, F)$ che riconosce L . Vogliamo trovare un automa A' per L^R : questo è facile, basta mantenere la struttura dell'automato invertendo il senso delle transizioni, rendendo finale lo stato iniziale e rendendo iniziali tutti gli stati finali.

In poche parole, definiamo l'automato

$$A' = (Q, \Sigma, \delta', F, \{q_0\})$$

definito dalla **funzione di transizione** δ' tale che

$$\delta'(q, a) = \{p \mid \delta(p, a) = q\}.$$

In poche parole, visto che abbiamo reversato le transizioni, dobbiamo vedere tutti gli stati p che finivano in q con a per poter fare il **cammino inverso**.

8.1.1. DFA

Ci piace questa soluzione? **NO**: siamo partiti da un DFA e abbiamo ottenuto un NFA per via degli stati iniziali multipli F e per il fatto che la funzione di transizione può mappare in più stati con la stessa lettera letta.

Se voglio ottenere un DFA devo fare il classicissimo salto esponenziale con la **costruzione per sottoinsiemi**. Con il reversal dell'automa non abbiamo cambiato il numero degli stati, quindi

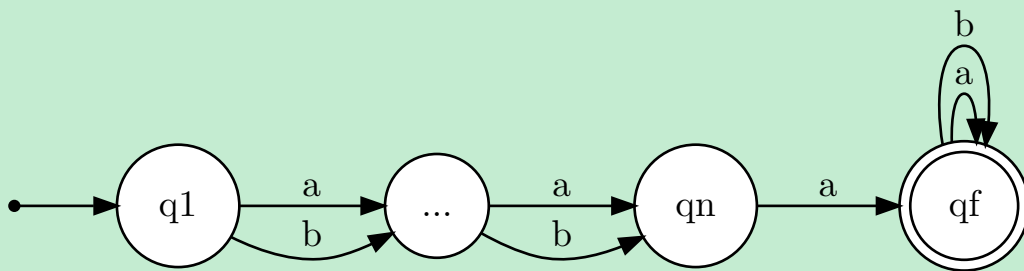
$$\text{sc}(L^R) \leq 2^{\text{nsc}(L^R)}.$$

Possiamo fare di meglio o nel caso peggiore abbiamo questo salto esponenziale?

Esempio 8.1.1.1: Prendiamo

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

il linguaggio che ha l' n -esimo simbolo da sinistra uguale ad una a , che riconosciamo con il seguente automa deterministico.



In questo caso, il numero di stati è $n + 1$, visto che ne abbiamo n all'inizio per eliminare gli $n - 1$ caratteri iniziali e poi uno stato per verificare di avere una a .

Il suo reversal è $L^R = L_n$, il solito linguaggio dell' n -esimo simbolo da destra uguale ad una a . Abbiamo visto che un NFA per L_n ha $n + 1$ stati mentre il DFA ha 2^n stati, visto che deve osservare finestre di n caratteri consecutivi.

Il **gap esponenziale** non riusciamo purtroppo ad evitarlo.

Esempio 8.1.1.2: Molto curiosa la situazione inversa: se partiamo da L_n riconosciuto da un DFA di 2^n stati noi otteniamo un reversal $L_n^R = L$ che, minimizzato, ha $n + 1$ stati (*escluso lo stato trappola*).

8.1.2. NFA

Se partiamo invece da un NFA, con la costruzione precedente manteniamo lo status di NFA, mantenendo inoltre il numero degli stati, quindi siamo contenti, ottenendo

$$\text{nsc}(L^R) = \text{nsc}(L).$$

8.1.3. Reversal di una grammatica

Abbiamo già visto l'operazione di reversal applicato ad un linguaggio, ovvero dato L regolare definiamo

$$L^R = \{w^R \mid w \in L\}$$

anch'esso regolare formato da tutte le stringhe di L lette in senso opposto. In poche parole, se $w = a_1 \dots a_n$ allora $w^R = a_n \dots a_1$. Ovviamente, il reversal è **autoinverso**, ovvero

$$(L^R)^R = L.$$

Cosa succede se applichiamo il reversal ad una **grammatica**?

Data una grammatica G di tipo 2 per L , come otteniamo una grammatica per L^R ? Abbiamo scelto una grammatica di tipo 2 perché le regole sono «*standard*» nella forma $A \rightarrow \alpha$ con $\alpha \in (V \cup \Sigma)^+$. Questa costruzione è facile: invertiamo ogni parte destra delle regole di derivazione, ovvero definiamo

$$\forall (A \rightarrow \alpha) \in P \quad \text{definiamo } A \rightarrow \alpha^R.$$

Esempio 8.1.3.1: Data una grammatica per L con regola di produzione

$$A \rightarrow aaAbBb,$$

una grammatica per L^R ha come regola di produzione

$$A \rightarrow bBbAaa.$$

Prendiamo ora una grammatica di tipo 3, che sappiamo essere capace di generare i linguaggi regolari, a meno della parola vuota. Le regole di produzione sono nella forma

$$A \rightarrow aB \mid a.$$

Avevamo visto due estensioni delle grammatiche regolari: una di queste è rappresentata dalle **grammatiche lineari a destra**, ovvero quelle grammatiche con regole di produzione

$$A \rightarrow wB \mid w \quad \text{tale che } w \in \Sigma^*.$$

Avevamo anche dimostrato che queste grammatiche sono equivalenti alle grammatiche di tipo 3 trasformando la stringa w in una serie di derivazioni che rispettino le grammatiche regolari.

L'altra estensione sono le grammatiche **lineari a sinistra**, con regole di produzione

$$A \rightarrow Bw \mid w \quad \text{tale che } w \in \Sigma^*.$$

Cosa possiamo dire di queste?

Lemma 8.1.3.1: Le grammatiche lineari a sinistra generano i linguaggi regolari.

Dimostrazione 8.1.3.1.1: Partiamo da una G lineare a sinistra e applichiamo il **reversal**: otteniamo una grammatica G' lineare a destra, visto che applichiamo la trasformazione

$$A \rightarrow Bw \mid w \implies A \rightarrow wB \mid w.$$

Possiamo quindi dire che

$$L(G') = (L(G))^R.$$

Sappiamo che i linguaggi regolari sono chiusi rispetto all'operazione di reversal, quindi essendo $L(G')$ regolare allora anche $L(G)$ lo deve essere. Quindi anche le grammatiche lineari a sinistra generano i linguaggi regolari. ■

Se prendiamo un linguaggio che è sia lineare a destra e a sinistra otteniamo le **grammatiche lineari**, che però generano di più rispetto alle grammatiche regolari.

8.1.4. Algoritmo per l'automa minimo

Dato $M = (Q, \Sigma, \delta, q_0, F)$ un DFA senza stati irraggiungibili, costruiamo l'automa per il reversal del linguaggio accettato da M . Definiamo quindi l'automa $M^R = (Q, \Sigma, \delta^R, F, \{q_0\})$ definito dalla **funzione di transizione**

$$\delta^R(p, a) = \{q \mid \delta(q, a) = p\}.$$

Questo automa, ovviamente, è NFA per via del non determinismo sulle transizioni e del non determinismo sugli stati iniziali multipli. A noi non piace, vogliamo ancora un DFA, quindi applichiamo la **costruzione per sottoinsiemi**.

Definiamo allora l'automa $N = \text{sub}(M^R)$ DFA ottenuto dalla **subset construction**, definito dalla tupla $(Q'', \Sigma, \delta'', q_0'', F'')$ tale che:

- $Q'' \leq 2^Q$ **insieme degli stati raggiungibili**, ovvero andiamo a rimuovere dall'automa tutti gli stati che sono irraggiungibili. Con **stati**, ovviamente, ci stiamo riferendo ai vari sottoinsiemi;
- δ'' **funzione di transizione** tale che

$$\delta''(\alpha, a) = \bigcup_{p \in \alpha} \delta^R(p, a);$$

- $q_0'' = F$ **stato iniziale** preso da M^R , che aveva già un insieme come stato iniziale;
- F'' **insieme degli stati finali** tale che

$$F'' = \{\alpha \in Q'' \mid q_0 \in \alpha\}$$

perché per il reversal lo stato iniziale diventava finale.

Vediamo adesso un risultato abbastanza strano di questa costruzione.

Lemma 8.1.4.1: N è il DFA minimo per il reversal del linguaggio riconosciuto da M .

Dimostrazione 8.1.4.1.1: Per prima cosa, dimostriamo che N **riconosce** il reversal del linguaggio riconosciuto da M . Ma questo è banale: l'abbiamo fatto per costruzione, usando prima il reversal e poi la costruzione per sottoinsiemi.

Dimostriamo quindi che N è **minimo**. In un automa minimo, tutti gli stati sono distinguibili tra loro. Analogamente, al posto di dimostrare questo, possiamo fare vedere che

$$\forall A, B \in Q'' \quad A, B \text{ non distinguibili} \implies A = B.$$

Assumiamo quindi che $A, B \in Q''$ siano due stati non distinguibili e che allora vale $A = B$. Questi due stati sono sottoinsiemi derivanti dalla costruzione per sottoinsiemi, quindi per dimostrare l'uguaglianza di insiemi devo dimostrare che

$$A \subseteq B \wedge B \subseteq A.$$

Partiamo con $A \subseteq B$. Sia $p \in A$, allora, visto che tutti gli stati sono raggiungibili, esiste una stringa $w \in \Sigma^*$ tale che, nell'automa M , vale

$$\delta(q_0, w) = p.$$

Per come abbiamo definito l'NFA per il reversal, allora

$$q_0 \in \delta^R(p, w^R).$$

Usando invece il DFA, abbiamo che

$$q_0 \in \delta''(A, w^R)$$

perché abbiamo assunto che $p \in A$. Ma allora w^R è accettata da N partendo da A .

Ora, visto che A e B non sono distinguibili per ipotesi, la stringa w^R è accettata da N partendo anche da B , cioè

$$q_0 \in \delta''(B, w^R)$$

quindi esiste un elemento $p' \in B$ tale che

$$q_0 \in \delta^R(p', w^R)$$

e quindi che

$$\delta(q_0, w) = p'.$$

L'automa è deterministico, quindi $p = p'$, e quindi che $p \in B$, e quindi

$$A \subseteq B.$$

La dimostrazione è analoga per la seconda inclusione. ■

Questa costruzione è un po' strana e contro quello che abbiamo sempre fatto: la costruzione per sottoinsiemi di solito fa esplodere il numero degli stati, mentre stavolta ci dà l'automa minimo per il reversal. Cosa possiamo fare con questo risultato?

Algoritmo di Brzozowski

Sia K un DFA per il linguaggio L senza stati irraggiungibili

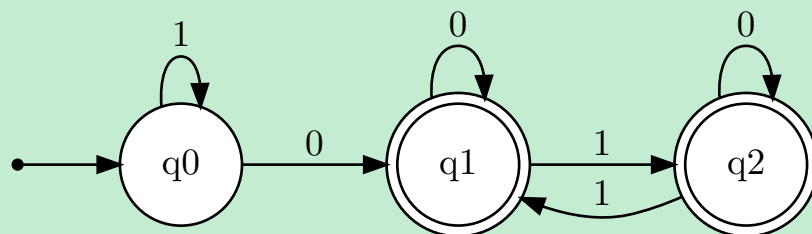
- 1: Costruiamo K^R NFA per L^R
 - 2: Costruiamo $M = \text{sub}(K^R)$ DFA per L^R
 - 3: Costruiamo M^R NFA per $(L^R)^R = L$
 - 4: Costruiamo $N = \text{sub}(M^R)$ DFA per L
-

Teorema 8.1.4.1: N è l'automa minimo per L .

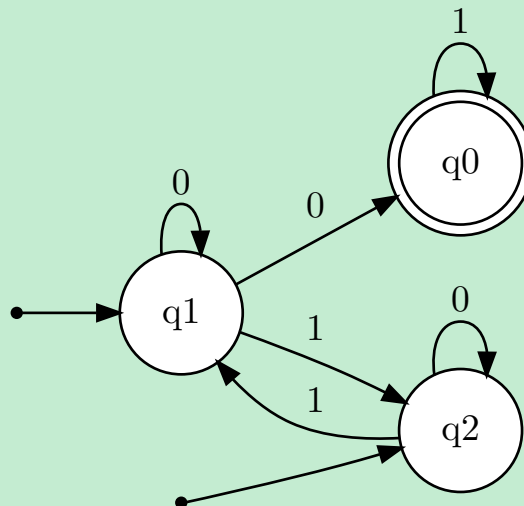
Dimostrazione 8.1.4.1.2: La dimostrazione è una conseguenza del lemma precedente: la prima coppia reversal+subset costruisce l'automa minimo per L^R , mentre la seconda coppia reversal+subset costruisce l'automa minimo per $(L^R)^R = L$. ■

Grazie all'**algoritmo di Brzozowski** noi abbiamo a disposizione un **algoritmo di minimizzazione** un po' strano dal punto di vista pratico che però ottiene l'automa minimo, anche se non è il più efficiente, ce ne sono di migliori.

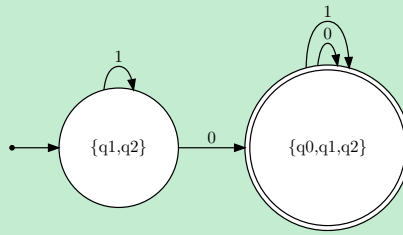
Esempio 8.1.4.1: Ci viene dato il DFA K della seguente figura.



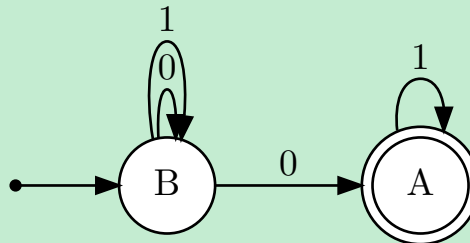
Notiamo subito che lo stato q_2 è ridondante. Andiamo a costruire K^R .



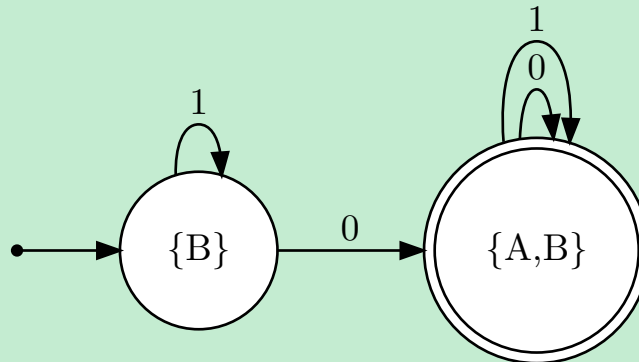
Rendiamo DFA questo automa, e chiamiamolo M .



Facciamo il renaming, chiamando A l'insieme iniziale e B l'insieme finale. Rifacciamo ora la costruzione del reversal, ottenendo M^R .



Infine, facciamo la costruzione per sottoinsiemi su M^R ottenendo N .



8.2. Shuffle

L'operazione di **shuffle**, applicata a due stringhe, le prende e le mescola, mantenendo l'ordine dei caratteri mentre le mischiamo. In poche parole, possiamo pensare di avere due **mazzieri**, ognuno dei quali tiene una stringa come lista ordinata (non lessicograficamente, ma proprio come è stata scritta) dei suoi caratteri. Ad ogni iterazione scegliamo a quale mazziere chiedere un carattere, e lo aggiungiamo alla stringa finale.

Come vediamo, l'inserimento che facciamo **non è atomico**: posso chiedere al mazziere che voglio ad ogni iterazione, l'unica cosa che mi viene chiesta è di mantenere l'ordine.

Esempio 8.2.1: Date le stringhe *aabb* e *b*, possiamo ottenere le stringhe

baabb

ababb

aabbb

aabbb

aabbb

Ovviamente, le ultime tre stringhe sono tutte uguali e vanno considerate come stringa unica.

Esempio 8.2.2: Date le stringhe *aabb* e *ab*, possiamo ottenere molte stringhe con lo shuffle. Ne vediamo un paio per vedere la non atomicità dell'operazione.

ababbb

aaabbb

L'operazione di shuffle, se la applichiamo ai **linguaggi** L' e L'' , è il linguaggio di tutte le stringhe ottenute tramite shuffle di una stringa di L' con una stringa di L'' . Ovviamente prendiamo tutte le possibili coppie di stringhe per ottenere il linguaggio completo.

8.2.1. Alfabeti disgiunti

Consideriamo due DFA A' e A'' per i due linguaggi appena definiti. Il caso più semplice di automa per lo shuffle parte con gli alfabeti per i due linguaggi **disgiunti**, ovvero L' definito sull'alfabeto $\Sigma' = \{a, b\}$ e L'' definito sull'alfabeto $\Sigma'' = \{c, d\}$.

Prendiamo spunto dall'**automa prodotto**: uniamo i due automi e ne mandiamo avanti uno alla volta in base al carattere che leggiamo. Definiamo quindi

$$A = (Q, \Sigma, \delta, q_0, F)$$

tale che:

- gli **stati** sono tutte le possibili coppie di stati dei due automi, ovvero

$$Q = Q' \times Q'';$$

- lo **stato iniziale** è formato dai due stati iniziali base, ovvero

$$q_0 = (q'_0, q''_0);$$

- l'**alfabeto** è l'unione dei due alfabeti di base, ovvero

$$\Sigma = \Sigma' \cup \Sigma'';$$

- la **funzione di transizione**, in base al carattere che legge, deve mandare avanti uno dei due automi e mantenere l'altro nello stesso stato, ovvero

$$\delta((q, p), x) = \begin{cases} (\delta'(q, x), p) & \text{se } x \in \Sigma' \\ (q, \delta''(p, x)) & \text{se } x \in \Sigma'' \end{cases};$$

- gli **stati finali** sono tutte le coppie formate da stati finali, perché devo riconoscere sia la prima che la seconda stringa, ovvero

$$F = \{(q, p) \mid q \in F' \wedge p \in F''\}.$$

Il numero di stati di questo automa è il prodotto del numero di stati dei due automi, ovvero

$$\text{sc}(\text{shuffle}(L', L'')) = \text{sc}(L') \cdot \text{sc}(L'').$$

Abbiamo considerato solo il caso in cui A' e A'' sono DFA. Per il caso non deterministico, basta modificare leggermente la funzione di transizione ma il numero di stati rimane invariato.

8.2.2. Stesso alfabeto

Se invece i due linguaggi sono definiti sullo stesso alfabeto Σ come ci comportiamo? Dobbiamo fare affidamento sul **non determinismo**: dobbiamo scommettere se il carattere che abbiamo letto deve mandare avanti il primo automa o il secondo automa. Tra tutte le possibili computazioni ce ne deve essere una che termina in una coppia di stati entrambi finali. Se nessuna computazione termina in uno stato accettabile allora rifiutiamo la stringa data.

Se partiamo da due DFA otteniamo un NFA con un numero di stati uguale al prodotto degli stati dei due automi iniziali. Questa situazione non ci piace, quindi torniamo in un DFA facendo un salto esponenziale con la costruzione per sottoinsiemi, quindi

$$\text{sc}(\text{shuffle}(L', L'')) \leq 2^{\text{nsc}(\text{shuffle}(L', L''))}.$$

Invece, se partiamo da due NFA otteniamo ancora un NFA, quindi la situazione ci piace. Il numero di stati l'abbiamo già definito ed è uguale a

$$\text{nsc}(\text{shuffle}(L', L'')) = \text{nsc}(L') \cdot \text{nsc}(L'').$$

8.2.3. Alfabeto unario

Infine, se i due linguaggi sono definiti sull'**alfabeto unario**, ovvero

$$L', L'' \subseteq \{a\}^*$$

l'operazione di shuffle collassa banalmente l'operazione di **prodotto**, perché alla fine stiamo facendo shuffle su stringhe che sono formate sempre da una e una sola lettera, quindi non conta come le mischiamo ma conta la lunghezza finale della stringa che ci esce.

8.3. Raddoppio

Sia L regolare, definiamo l'operazione α tale che

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

Esempio 8.3.1: Dato il linguaggio $L = \{a^n b^n \mid n \geq 0\}$ allora

$$\alpha(L) = \{\varepsilon\}.$$

Dato invece il linguaggio $L = a^*$ allora

$$\alpha(L) = a^*.$$

Infine, dato il linguaggio $L = \{a^n \mid n \text{ pari}\}$ allora

$$\alpha(L) = a^*.$$

Se L è regolare, riesco a dimostrare che anche $\alpha(L)$ è regolare? Abbiamo a disposizione un automa A per L , vogliamo sapere se il mio input, raddoppiato, viene accettato da L .

Come possiamo ragionare? Vogliamo cercare un cammino che fa da q_0 a $q_f \in F$ leggendo la stringa xx . Nell'automata A , leggendo x , finiamo in uno stato p : dobbiamo cercare di indovinare questo p per far partire la computazione una seconda volta e arrivare in $q_f \in F$.

L'idea è quindi di scommettere lo stato che raggiungiamo con A leggendo x , e poi mandare in parallelo due copie di A , uno dall'inizio e uno dallo stato indovinato.

Formalizziamo questo automa. Dato $A = (Q, \Sigma, \delta, q_0, F)$ DFA per L , costruiamo l'automa

$$A' = (Q', \Sigma, \delta', I', F')$$

tale che:

- l'**insieme degli stati** $Q = Q^3$ è formato da triple

$$[p, q', q'']$$

dove:

- p è lo stato che abbiamo scommesso di raggiungere con x in A ;
- q' è lo stato che portiamo avanti in A a partire da q_0 ;
- q'' è lo stato che portiamo avanti in A a partire da p ;
- l'**insieme degli stati iniziali** (multipli)

$$I' = \{[p, q_0, p] \mid p \in Q\}$$

dove scommettiamo un qualunque stato p come stato intermedio;

- l'**insieme degli stati finali**

$$F' = \{[p, p, q] \mid q \in F\}$$

formato da tutti gli stati dove l'automa A finisce in p nella computazione iniziale e finisce in uno stato finale nella computazione da p

- la **funzione di transizione** δ' è tale che

$$\delta'([p, q', q''], a) = [p, \delta(q', a), \delta(q'', a)]$$

che manda avanti i due automi in parallelo.

Purtroppo, quello che otteniamo è un **NFA** per via di tutti gli stati iniziali multipli.

8.4. Metà

Un'altra operazione strana che vediamo prende un linguaggio regolare L e calcola

$$\frac{1}{2}L = \{x \in \Sigma^* \mid \exists y \mid |y| = |x| \wedge xy \in L\}.$$

In poche parole, prendo le stringhe di L di lunghezza pari e prendo la prima metà di queste.

Esempio 8.4.1: Dato il linguaggio $L = \{a^n b^n \mid n \geq 0\}$ allora

$$\frac{1}{2}L = a^*.$$

Si può dimostrare che questa operazione **mantiene la regolarità**, ma come facciamo? Possiamo ricondurre questo problema a quello precedente, variando un po' la seconda computazione.

In questo caso facciamo molte più scommesse: al posto di mandare avanti in parallelo i due automi, con la scommessa sullo stato intermedio p , qua mandiamo avanti il primo automa normalmente e il secondo lo mandiamo avanti non deterministicamente prendendo ogni simbolo possibile di Σ . Infatti, la stringa y è randomica, la dobbiamo inventare noi.

Cambia quindi la **funzione di transizione** δ' , che prende ancora la tripla dello stato ma ora:

- mantiene invariato lo stato scommessa;
- manda avanti deterministicamente il primo automa;
- per ogni carattere di Σ fa partire una computazione con quel carattere.

Quello che otteniamo è un **turbo NFA**, se non volessimo utilizzarlo? Abbiamo rappresentazioni alternative che ci bypassano il non determinismo?

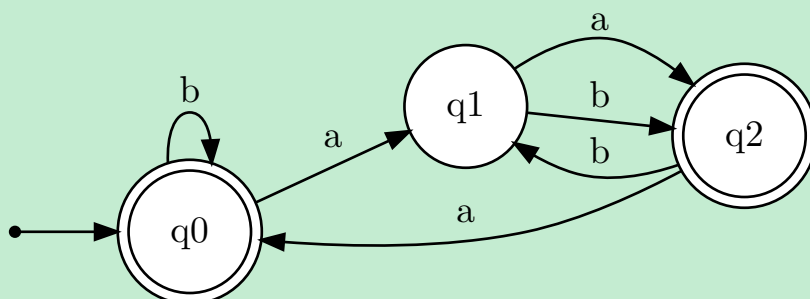
8.5. Automi come matrici

Possiamo rappresentare gli automi come **matrici di adiacenza**: esse sono matrici indicizzate, su righe e colonne, dagli stati dell'automato, e ogni cella è un valore booleano che viene posto a 1 se e solo se abbiamo una transizione dallo stato riga allo stato colonna.

Queste matrici sono dette **matrici di transizione**, e rappresentano le transizioni che possiamo fare all'interno dell'automato. Queste matrici possono essere anche **associate ad una lettera** di Σ , e queste rappresentano le transizioni che possono essere fatte nell'automato con quella lettera. Se invece le matrici sono **associate ad una stringa** rappresentano le transizioni che possono essere fatte nell'automato leggendo quella stringa, ma queste le vedremo meglio dopo.

Il **numero di possibili matrici** che possiamo costruire è finito: esso è $2^{n \times n}$, con $n = |Q|$. Questa informazione ci servirà dopo per costruire degli automi.

Esempio 8.5.1: Costruiamo le matrici di transizione del seguente automa.



Andiamo a calcolare le due matrici di transizione delle lettere a e b .

$$M_a = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \qquad M_b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Ovviamente, se l'automato è un **DFA** abbiamo un solo 1 per ogni riga.

Se calcoliamo $M_a M_b$ otteniamo la matrice che ci dice in che stato finiamo leggendo ab in base allo stato di partenza che scegliamo. Infatti:

$$M_{ab} = M_a M_b = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

che indica esattamente gli stati che raggiungiamo leggendo la stringa ab .

Possiamo banalmente estendere questa moltiplicazione ad una stringa generica $w = a_1 \dots a_n$, calcolando la matrice M_w come

$$M_w = M_{a_1} \cdots M_{a_n}.$$

Queste matrici ci danno informazioni molto interessanti: ogni riga ci dice in che stato possiamo arrivare partendo dallo stato della riga leggendo una certa sequenza di caratteri. Come possiamo utilizzare questa matrice a nostro vantaggio?

8.5.1. Prima applicazione: raddoppio

Riprendiamo l'operazione α : possiamo utilizzare le matrici per risolvere questo problema evitando il non determinismo. Se noi avessimo M_x sarebbe molto facile rispondere a $xx \in L$:

- prendiamo la riga dello stato iniziale q_0 e vediamo la colonna p che contiene l'1 della riga;
- prendiamo la riga p e vediamo la colonna q_f che contiene l'1 della riga;
- verifichiamo se $q_f \in F$.

Con queste tabelle è molto facile risolvere α : ce le teniamo nello stato e mano a mano costruiamo l'automa con le tabelle nuove, e poi alla fine verifichiamo quello scritto sopra.

Definiamo quindi l'automa

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

tale che:

- l'**insieme degli stati** tiene tutte le possibili matrici booleane con indici in Q , ovvero

$$Q' = \{0, 1\}^{|Q| \times |Q|};$$

- lo **stato iniziale** è la matrice identità $I_{|Q|}$ perché all'inizio non viene letto niente (viene letta ϵ) e quindi non ci spostiamo dallo stato nel quale siamo;
- la **funzione di transizione** δ' è tale che

$$\delta'(M, a) = MM_a;$$

- l'**insieme degli stati finali** contiene tutti gli stati che hanno delle matrici con le proprietà descritte all'inizio della sezione, ovvero

$$F' = \{M \mid \exists p \in Q \mid M[q_0, p] = 1 \wedge \exists q \in F \mid M[p, q] = 1\}.$$

Sicuramente questo è un automa a stati finiti: il numero di matrici, anche se esponenziale, è comunque un numero finito. Inoltre, questo automa che otteniamo è DFA, a differenza del precedente.

8.5.2. Seconda applicazione: metà

Torniamo ora sull'operazione $\frac{1}{2}L$: come possiamo fare questo con le matrici definite poco fa? Se prima le matrici per entrambi gli automi erano le stesse, qua la seconda parte, visto che viene inventata, si riduce ad un problema di **raggiungibilità del grafo**.

Quello che faremo è mandare avanti il primo automa deterministicamente e il secondo invece verrà rappresentato dalle potenze della **matrice dell'automa** per vedere la raggiungibilità. Qua con matrice dell'automa intendiamo la prima versione che abbiamo definito della matrice.

La matrice dell'automa la otteniamo come somma booleana di tutte le matrici M_x associate al carattere $x \in \Sigma$. Facendo poi la potenza k -esima di questa matrice riusciamo a vedere la raggiungibilità dopo aver letto k simboli.

Definiamo quindi l'automa

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

tale che:

- l'**insieme degli stati** è formato da tutte le coppie

$$Q' = Q \times \{0, 1\}^{|Q| \times |Q|}$$

dove la prima componente rappresenta lo stato dell'automa che viene mandato avanti deterministicamente e il secondo rappresenta tutte le potenze della matrice dell'automa;

- lo **stato iniziale** parte dallo stato iniziale e dalla matrice identità, ovvero

$$q'_0 = (q_0, I_{|Q|});$$

- la **funzione di transizione** δ' è tale che

$$\delta'([q, K], a) = (\delta(q, a), KM);$$

- l'**insieme degli stati finali** contiene tutte le coppie dove la matrice, osservata nella riga definita dalla prima componente, contiene un 1 in una colonna di uno stato finale, ovvero

$$F' = \{[p, K] \mid \exists q \in F \mid K[p, q] = 1\}.$$

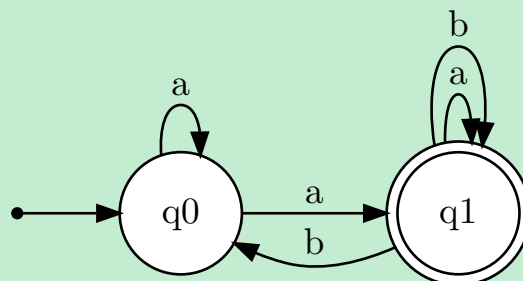
Il numero di stati è considerevole, considerando anche il prodotto cartesiano, ma abbiamo ottenuto un DFA, che invece prima non avevamo.

8.5.3. Matrici su NFA

Per ora abbiamo calcolato la matrice delle transizioni dei DFA, che succede se abbiamo un **NFA**? Ovviamente, in un NFA, avendo la possibilità di fare delle computazioni parallele, nella riga di una matrice associata ad una lettera possiamo avere più valori a 1.

Calcolando però le potenze della matrice dell'automa cosa otteniamo?

Esempio 8.5.3.1: Dato il seguente automa divertiamoci con qualche matrice.



Le matrici di transizione associate alle lettere a e b sono:

$$M_a = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

$$M_b = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$

Proviamo a calcolare la matrice M_{aa} calcolandola con la somma intera:

$$M_{aa} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}.$$

Questa matrice non rappresenta più la raggiungibilità con una matrice booleana, ma **conta il numero di cammini** che abbiamo nell'automata per raggiungere quello stato. Usando la somma booleana invece avremmo ancora la matrice che definisce la raggiungibilità.

I numeri che vediamo scritti nella tabella sono i **gradi di ambiguità** delle varie stringhe (*se ci limitiamo a quelle accettate*): questo rappresenta appunto il numero di modi in cui possiamo arrivare a quella stringa partendo dallo stato che indicizza la riga. Il **grado di ambiguità del grafo** è il massimo grado di ambiguità delle stringhe accettate.

CAPITOLO

9. Pumping Lemma

Come facciamo a dimostrare che un linguaggio non è regolare? Che tecniche abbiamo?

Prima di tutto abbiamo il **criterio di distinguibilità**: se troviamo un insieme X di parole distinguibili tra loro per un linguaggio L , allora ogni DFA per L ha almeno $|X|$ stati. Come lo utilizziamo? Se $|X| = \infty$ allora servono un numero infinito di stati, cosa che negli automi a **stati finiti** non è possibile.

Esempio 9.1: Definiamo il linguaggio

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

Gli automi a stati finiti **non sanno contare**, quindi non posso contare quante a ci sono nella stringa e poi verificare lo stesso numero di b .

Definiamo l'insieme

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

Esso è formato da stringhe distinguibili tra loro: infatti, date due stringhe $x = a^i$ e $y = a^j$, per distinguerle utilizziamo la stringa $z = b^i$.

Un altro modo per dimostrare la non regolarità è far vedere che il linguaggio dato fa saltare qualche **proprietà di chiusura**.

Esempio 9.2: Definiamo il linguaggio

$$L = \{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}$$

che palesemente non è regolare perché non posso contare, ma come lo dimostriamo?

Con la **distinguibilità** possiamo usare lo stesso insieme X di prima, ma facciamo finta di non saperlo fare.

Sappiamo che i linguaggi regolari sono chiusi rispetto all'operazione di intersezione. Prendiamo quindi un linguaggio regolare e facciamo l'intersezione con L , ad esempio facciamo

$$L \cap a^*b^*.$$

Visto che a^*b^* è un linguaggio regolare e l'intersezione chiude i linguaggi regolari, anche il linguaggio risultante deve essere regolare, ma questa intersezione genera il linguaggio dell'esempio precedente, perché date tutte le stringhe con a e b uguali filtriamo tenendo solo quelle che hanno tutte le a all'inizio e poi tutte le b .

Visto che il linguaggio risultante non è regolare, non lo è nemmeno L .

9.1. Definizione

L'ultimo metodo che abbiamo a disposizione è il **pumping lemma per i linguaggi regolari**.

Lemma 9.1.1 (*Pumping lemma per i linguaggi regolari*): Sia L un linguaggio regolare. Allora esiste una costante N tale che $\forall z \in L$, con $|z| \geq N$, possiamo scrivere z come

$$z = uvw$$

con:

1. $|uv| \leq N$;
2. $v \neq \varepsilon$;
3. $\forall k \geq 0 \quad uv^k w \in L$.

Un po' strano: il succo di questo lemma è che se prendiamo delle stringhe lunghe prima o poi qualcosa si deve ripetere. Infatti, i tre punti ci dicono questo:

1. il primo ci dice che la parte che contiene la ripetizione è all'inizio e non è troppo lontana;
2. il secondo ci dice che effettivamente viene ripetuto qualcosa;
3. il terzo ci dice che possiamo ripetere all'infinito la parte centrale senza uscire dal linguaggio, ovvero possiamo fare **pumping**, pompare.

In poche parole, la scomposizione di z avviene nei punti di ripetizione: u è la parte prima della ripetizione, v è la parte che viene ripetuta e w è la parte dopo la ripetizione.

Questa è una **condizione necessaria**: se faccio vedere se un linguaggio viola questo lemma allora non è regolare, ma potrebbe non bastare questo per far vedere che un linguaggio non è regolare.

Dimostrazione 9.1.1.1: Sia $A = (Q, \Sigma, \delta, q_0, F)$ un DFA per L . Sia $N = |Q|$.

Prendiamo una stringa $z = a_1 \dots a_m \in L$ con $|z| \geq N$. Un qualsiasi cammino accettante per z è nella forma

$$q_0 = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \dots \xrightarrow{a_m} p_m \in F$$

che attraversa $N + 1$ stati, ma noi avendone N vuol dire che almeno uno stato lo stiamo visitando 2+ volte.

Ma allora esistono $i, j \mid i < j$ tali che $p_i = p_j$. Andiamo a scomporre la nostra stringa z come $z = uvw$ tali che

$$\begin{aligned} u &= a_1 \dots a_i \\ v &= a_{i+1} \dots a_j \\ w &= a_{j+1} \dots a_m. \end{aligned}$$

Visto che $i < j$ allora la parte centrale ha almeno un elemento, quindi $v \neq \varepsilon$.

Inoltre, visto che $p_i = p_j$ vuol dire che partendo da p_i , leggendo la parte di stringa v , finiamo in p_j . Ma allora è possibile ripetere un numero questo cammino un numero arbitrario di volte.

Infine, per assunzione la lunghezza della stringa è $|z| = m \geq N$. Quando arriviamo all' N -esimo carattere abbiamo visto $N + 1$ stati, ovvero sono già passato in uno stato ripetuto,

quindi $|uv| \leq N$ perché la ripetizione deve avvenire prima dell'inizio dell'ultima parte della stringa. ■

9.2. Applicazioni

Vediamo come utilizzare il pumping lemma per dimostrare la non regolarità. Generalmente, faremo delle dimostrazioni per assurdo: assumendo la regolarità faremo vedere che esiste una stringa tale che ogni sua scomposizione possibile fa cadere almeno uno dei punti del pumping lemma.

Esempio 9.2.1: Definiamo il linguaggio $L = \{a^n b^n \mid n \geq 0\}$.

Per assurdo sia L un linguaggio regolare. Sia N la costante del pumping lemma. Definiamo la stringa $z = a^N b^N$ che rispetta la minima lunghezza delle stringhe. Infatti, $|z| = 2N \geq N$.

Scriviamo ora z come $z = uvw$. Deve valere il punto 1, ovvero $|uv| \leq N$, ma questo implica che u e v sono formate da solo a , quindi

$$u = a^i \wedge v = a^j \wedge w = a^{N-i-j} b^N \quad | \quad j \neq 0$$

perché deve valere $v \neq \varepsilon$.

Sappiamo inoltre che la ripetizione arbitraria di v mi mantiene nel linguaggio, ovvero che $\forall k \geq 0$ allora $uv^k w \in L$ ma questo non è vero: se scegliamo $k = 0$ la stringa uw non è più accettata perché essa è nella forma

$$uw = a^i a^{N-i-j} b^N = a^{N-j} b^N$$

e ovviamente $N - j \neq N$, quindi L non è regolare.

Facciamo un ultimo esempio dell'applicazione del pumping lemma.

Esempio 9.2.2: Definiamo il linguaggio

$$L = \{a^{2^n} \mid n \geq 0\} = \{a^{2^0}, a^{2^1}, \dots\} = \{a, aa, aaaa, \dots\}$$

insieme delle potenze di due scritte in unario.

Questo ovviamente non è regolare. Come lo dimostriamo con il pumping lemma?

Sia N la costante del PL per L . Prendiamo una stringa di L lunga almeno N , ovvero la stringa $z = a^{2^N}$ la cui lunghezza è $|z| \geq N$. Scomponiamo ora z come $z = uvw$, cosa possiamo dire?

Sappiamo che $v \neq \varepsilon$, quindi $|v| \geq 1$.

Inoltre, sappiamo della ripetizione arbitraria di v , quindi le stringhe $uv^k w$ devono stare in L . Cosa possiamo dire della lunghezza di questa stringa? Sappiamo che

$$|uv^k w| = 2^N + |v|(k-1) = 2^N + j(k-1) \stackrel{k=2}{=} 2^N + j < 2^{N+1}.$$

Che valore assume j ? La potenza successiva è 2^{N+1} ma j essendo un pezzo di z è al massimo 2^N quindi $j < 2^N$ e quindi $2^N + j < 2^{N+1}$.

Ma allora L non è regolare.

10. 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.

10.1. 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 10.1.1: 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 10.1.1.1: 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 $N \leq |z| < 2N$, 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.

10.2. 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.

10.3. 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.

10.4. 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.

11. Automi two-way

In questo capitolo discuteremo alcune **varianti di automi a stati finiti**, passando prima per variazioni molto leggere, poi per alcune che modificano profondamente il modello di calcolo e infine daremo un'occhiata agli **automi two-way**.

11.1. Varianti di automi a stati finiti

Vediamo prima di tutto delle varianti molto light dei nostri bellissimi automi.

11.1.1. Automi pesati e probabilistici

La prima, e unica, 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.

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.

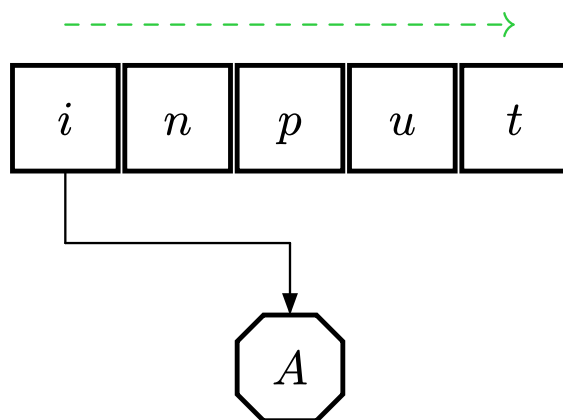
Questi automi li possiamo usare come **riconoscitori a soglia**: tutte le parole oltre una certa soglia le accettiamo, altrimenti le rifiutiamo.

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.

11.2. Varianti pesanti di automi a stati finiti

Passiamo ora ad alcune varianti un po' più di hardcore, con alcune che cambiano completamente il modello di calcolo che siamo abituati a vedere.

Parlando di **modelli di calcolo**, 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.

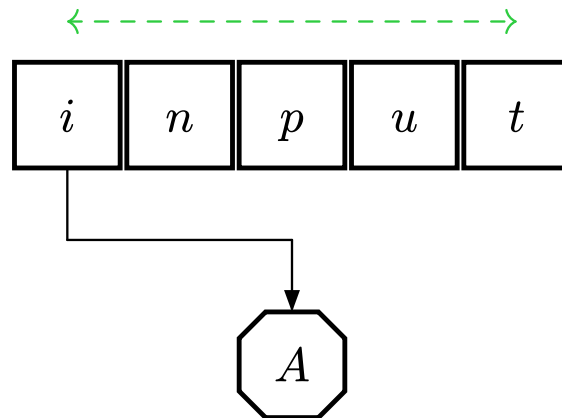


La classe di linguaggi che riconosce un automa a stati finiti è la classe dei **linguaggi regolari**.

Modifichiamo ora questo modello, toccando un po' tutti gli aspetti possibili.

11.2.1. 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.



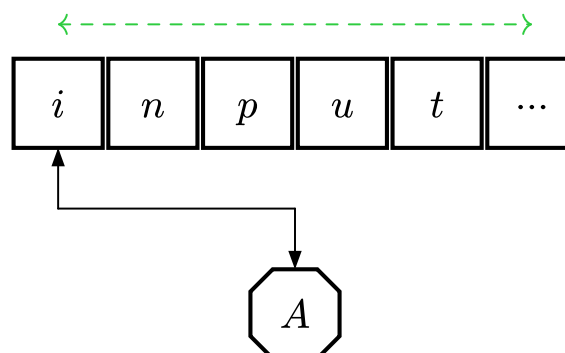
11.2.2. Read-only VS read-write

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.

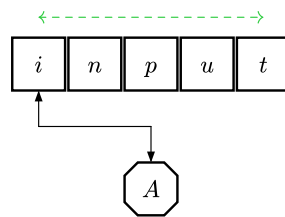
Consideriamo quindi un automa two-way che però mantiene la read-only del nastro: la classe che otteniamo è ancora una volta quella dei **linguaggi regolari**, e questo lo vedremo dopo.

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**:



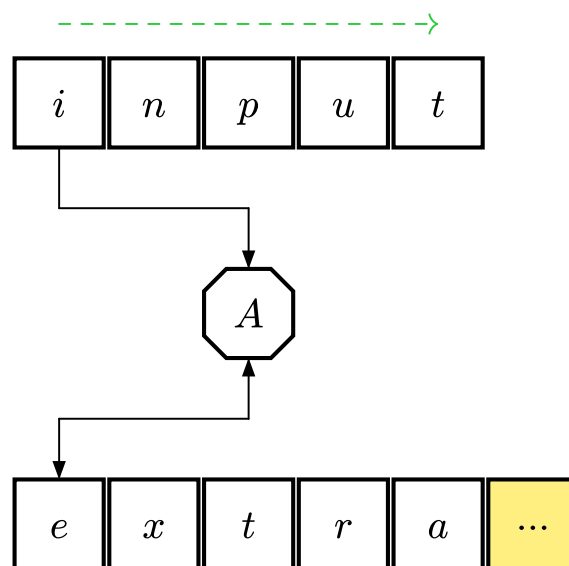
- se invece lasciamo il nastro grande quando l'input andiamo a riconoscere i linguaggi di tipo 1, ovvero otteniamo un **automa limitato linearmente**. Quest'ultima cosa vale perché nelle grammatiche di tipo 1 le regole di produzione non decrescono mai, e un automa limitato linearmente per capire se deve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



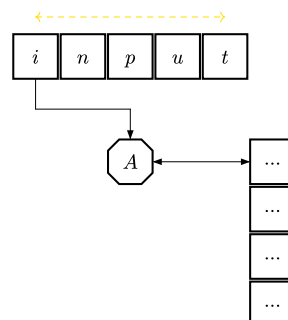
11.2.3. Memoria esterna

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.



11.3. 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.

11.3.1. Esempi vari

Esempio 11.3.1.1: Riprendiamo l'operazione α . Dato L regolare, α era tale che

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

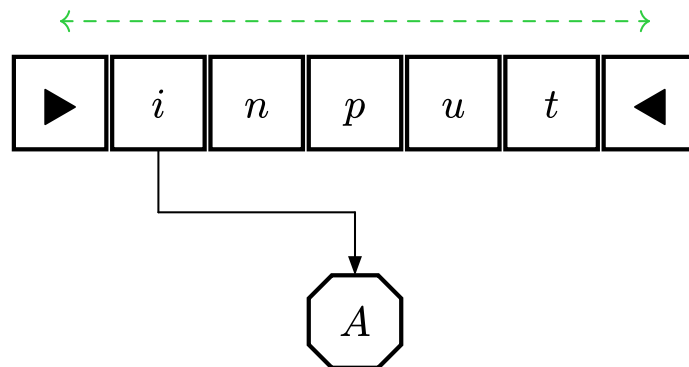
Abbiamo A un DFA che accetta L , come posso costruire un two-way per $\alpha(L)$? Potremmo leggere x la prima volta, ricordarci in che stato siamo arrivati, tornare indietro e poi ripartire a leggere x dallo stato nel quale eravamo arrivati e vedere se finiamo in uno stato finale.

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 11.3.1.2: 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 11.3.1.3: 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 11.3.1.4: 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.

11.3.2. 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.

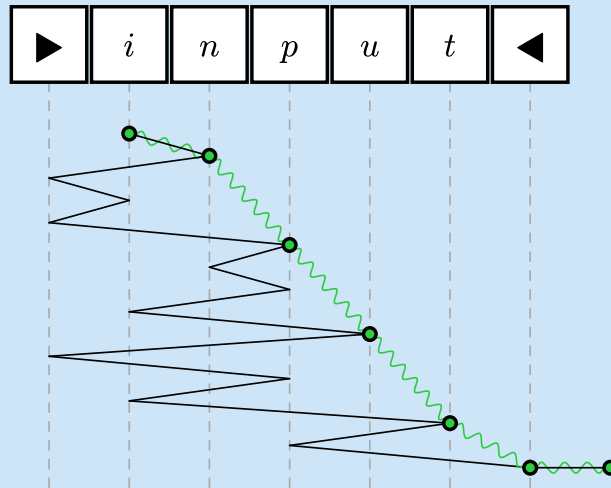
11.3.3. Potenza computazionale

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 11.3.3.1: Vale

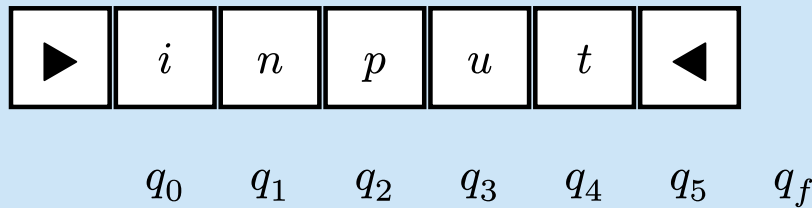
$$L(2DFA) = L(1DFA).$$

Dimostrazione 11.3.3.1.1: 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 noi 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

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

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

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

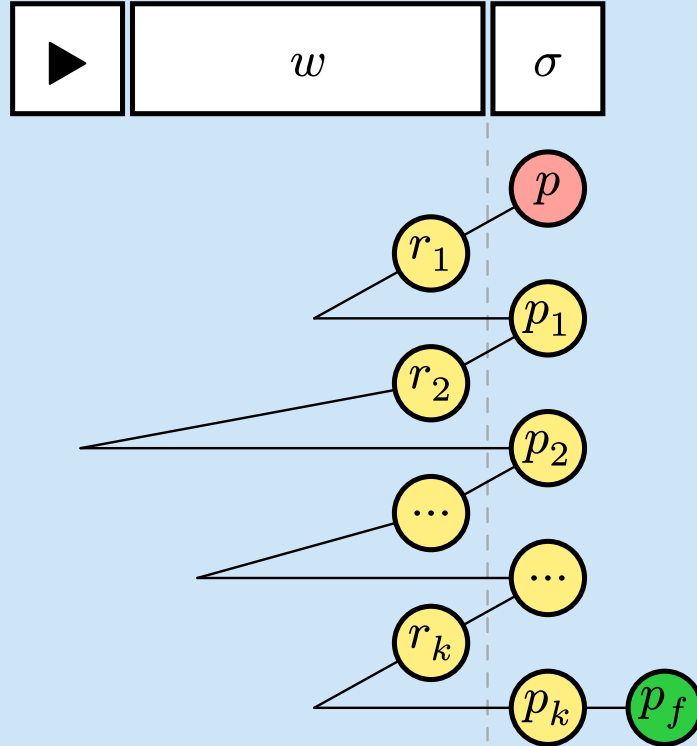
Supponiamo di aver calcolato la tabella di w , vediamo come costruire induttivamente la tabella di $w\sigma$, con $w \in \Sigma^*$ e $\sigma \in \Sigma$. Se vale

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

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

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

- 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**. Abbiamo visto la trasformazione da 2DFA a 1DFA, ma la stessa trasformazione può essere fatta per il passaggio da 2NFA a 1NFA.

11.3.4. Problema di Sakoda & Sipser

Abbiamo visto queste trasformazioni, ma quanto costano?

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.

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 11.3.4.1: 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 11.3.4.2: 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 consideriamo un **alfabeto unario** $\Sigma = \{a\}$:
 - 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

$$L = NL (????);$$

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

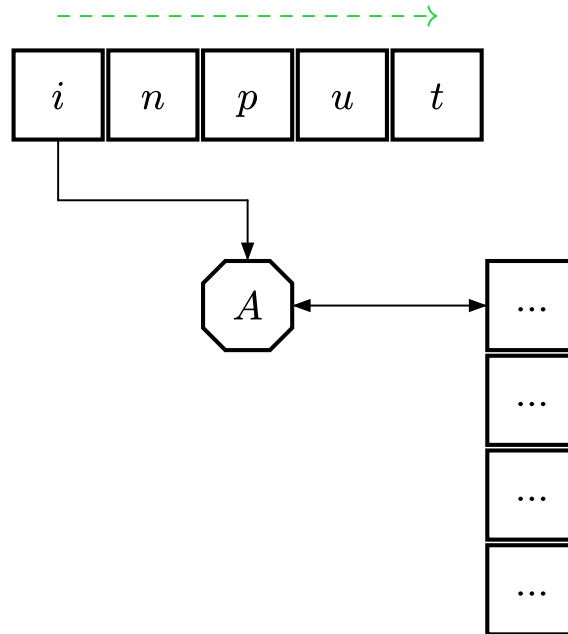
Dei ricercatori hanno trovato degli **automi completi** per questi problemi, ovvero degli automi che permettono lo studio dei problemi solo su questi pochi automi scelti per poi far «*arrivare*» tutte le conseguenze a tutti gli altri automi. Scritto malissimo, sono tipo gli NP-completi.

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

Parte III — Linguaggi context-free

1. Automi a pila

Chiusa la (grande) parte dei linguaggi regolari e degli automi a stati finiti è ora di passare ad una nuova classe di riconoscitori: gli **automi a pila**. 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 74](#) 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 si inserisce qualcosa si spinge giù quello che già c'era dentro.

1.1. Definizione

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 **distuggere**.

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:

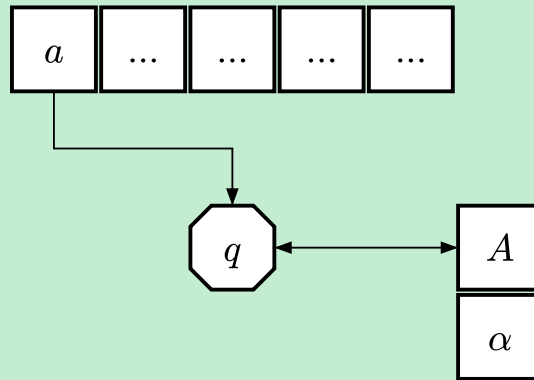
$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow \text{PF}(Q \times \Gamma^*).$$

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

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

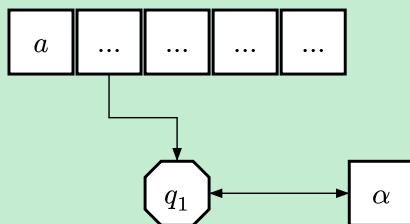


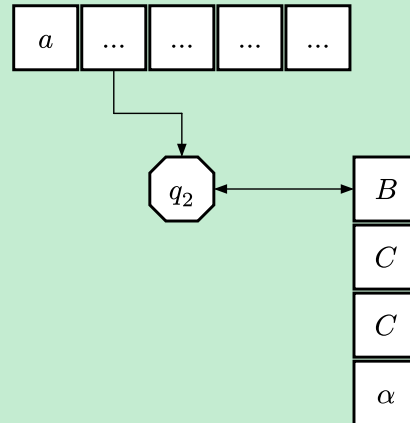
Con α nell'[Immagine 75](#) si intende una stringa generica in Γ^* che potremmo eventualmente avere sotto al carattere A in cima.

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.



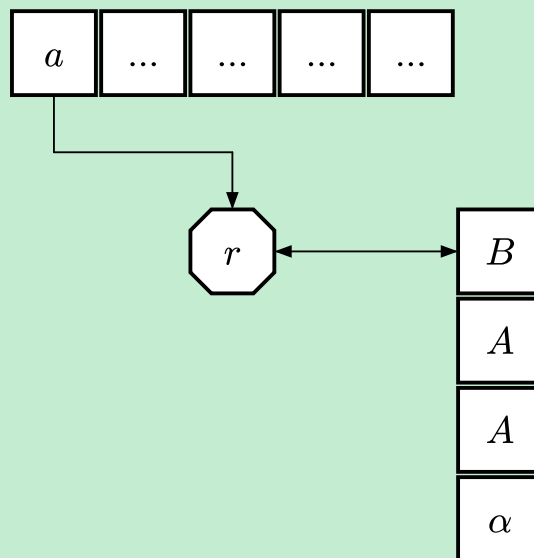


Per convenzione, quando inseriamo una stringa sulla pila, l'inserimento avviene da destra verso sinistra. In poche parole, se inseriamo la stringa $X \in \Gamma^*$ nella pila, se la togliessimo noi leggeremmo, in ordine, esattamente X . In altre parole ancora, quando leggiamo una stringa da inserire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

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

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

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

1.2. 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 1.2.1: 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 riuscivamo a ricordare il numero di a e poi confrontare questo numero con le b , mentre ora riusciamo a farlo, le pile sanno contare.

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

Andiamo a definire la funzione di transizione.

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

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

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

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

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

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

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

Andiamo a terminare la lettura delle b , quindi

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

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

Esempio 1.2.2: 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 1.2.3: 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

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

1.3. 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 1.3.1 (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.

1.4. Trasformazioni

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

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

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

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

un PDA tale che:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- simuliamo l'automa M senza aggiungere niente, quindi

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

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

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

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

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

1.5. Esempi

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

Esempio 1.5.1: 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 1.5.2: 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**.

2. Equivalenza tra CFL e grammatiche di tipo 2

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.

2.1. Ripasso e introduzione

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

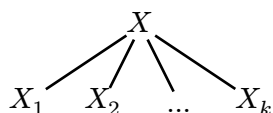
$$X \rightarrow X_1 \dots X_k \quad | \quad X \in V \wedge X_1, \dots, X_k \in (V \cup \Sigma) \wedge k \geq 0.$$

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.

In un albero generico, la **radice** e i **nodi interni** sono **variabili**, mentre le foglie sono combinazioni di **variabili e terminali**. In un albero che rappresenta una stringa del linguaggio nella radice ho l'**assioma** S e nelle foglie ho **solo terminali**.

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**:

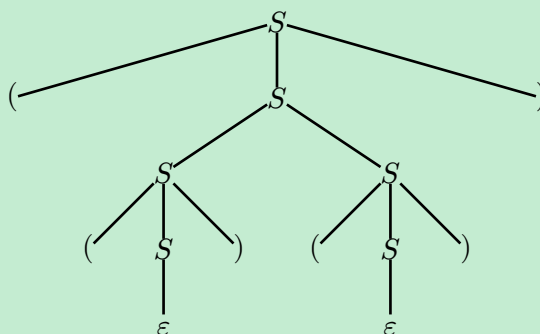


Vediamo questi alberi applicati a qualche esempio.

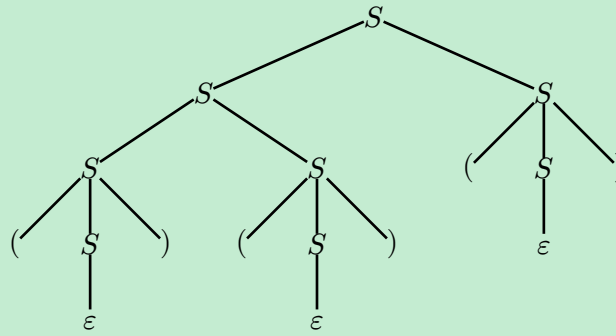
Esempio 2.1.1: Riprendiamo la grammatica per le parentesi tonde bilanciate, che avevamo fatto a inizio corso, con le seguenti regole di produzione:

$$S \rightarrow \varepsilon \mid (S) \mid SS.$$

Vediamo due alberi di derivazione che abbiamo in questo linguaggio.



Questo primo albero rappresenta la derivazione della stringa $w = (())$.



Questo secondo albero genera invece la stringa $w = ()()$.

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

Gli alberi sono un **modo compatto** di descrivere un processo di derivazione.

Esempio 2.1.2: Rimaniamo con il linguaggio dell'Esempio 2.1.1, 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 2.1.3: Nelle tre derivazioni dell'Esempio 2.1.2 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 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 2.1.4: La grammatica delle parentesi tonde bilanciate è **ambigua**, mentre la grammatica della 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).

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 2.2.1: Vale

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

Dimostriamo questo con un esempio.

Esempio 2.2.1: Definiamo la grammatica $G = (V, \Sigma, P, S)$ tale che:

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

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

$$S \longrightarrow TU$$

$$T \longrightarrow aTb \mid \varepsilon$$

$$U \longrightarrow bUa \mid \varepsilon$$

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:

$$\delta(q, \varepsilon, S) = \{(q, TU)\}$$

$$\delta(q, \varepsilon, T) = \{(q, aTb), (q, \varepsilon)\}$$

$$\delta(q, \varepsilon, U) = \{(q, bUa), (q, \varepsilon)\}.$$

E terminiamo con le regole del secondo tipo:

$$\delta(q, a, a) = \{(q, \varepsilon)\}$$

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

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.

2.3. Da automa a pila a grammatica di tipo 2

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

Esempio 2.3.1: Definiamo l'alfabeto $\Sigma = \{ (,) \}$ e consideriamo l'alfabeto delle stringhe che rappresentano sequenze di parentesi tonde bilanciate. Come costruiamo un automa a pila per questo linguaggio?

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

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

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

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

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

2.3.1. Forma normale per gli automi a pila

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

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

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

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

Esempio 2.3.1.1: Avendo delle regole nella forma

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

allora:

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

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

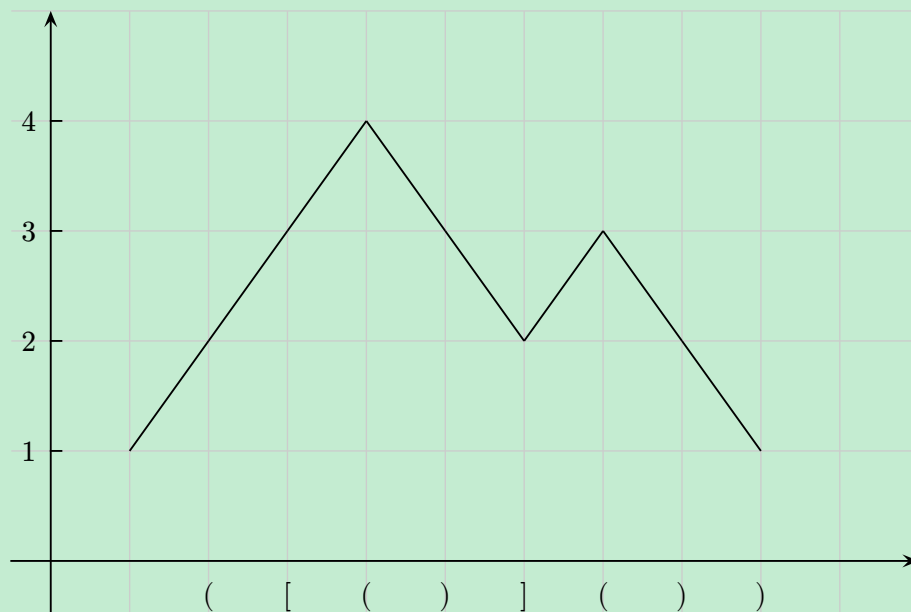
Esempio 2.3.1.2: Definiamo l'alfabeto $\Sigma = \{ (,), [,] \}$. Vogliamo descrivere la computazione accettante della stringa $w = ([()])()$ data in input ad un automa a pila.

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

$$\begin{array}{cccccccc}
 & & & & A & & & \\
 & & & B & B & B & & A \\
 & A & A & A & A & A & A & \\
 Z_0 & Z_0 & Z_0 & Z_0 & Z_0 & Z_0 & Z_0 & Z_0
 \end{array}
 \cdot$$

$$\begin{array}{cccccccc}
 (& [& (&) &] & (&) &)
 \end{array}$$

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



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

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

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

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

Aggiungiamo ancora una regola:

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

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

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

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

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

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

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

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

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

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

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

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

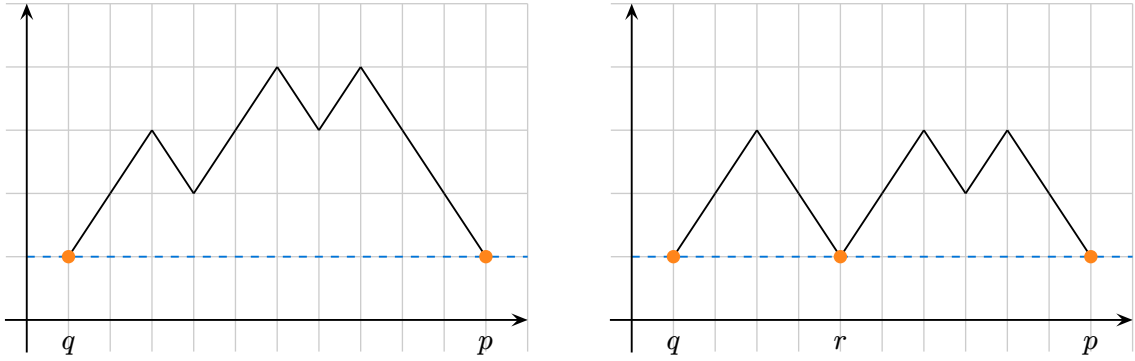
$$(p, -) \in \delta(q, \varepsilon, A).$$

2.3.2. Dimostrazione

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

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

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



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

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

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

Vediamo le **mosse** che possiamo fare.

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

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

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

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

Aggiungiamo anche la produzione

$$[qAq] \rightarrow \varepsilon$$

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

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

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

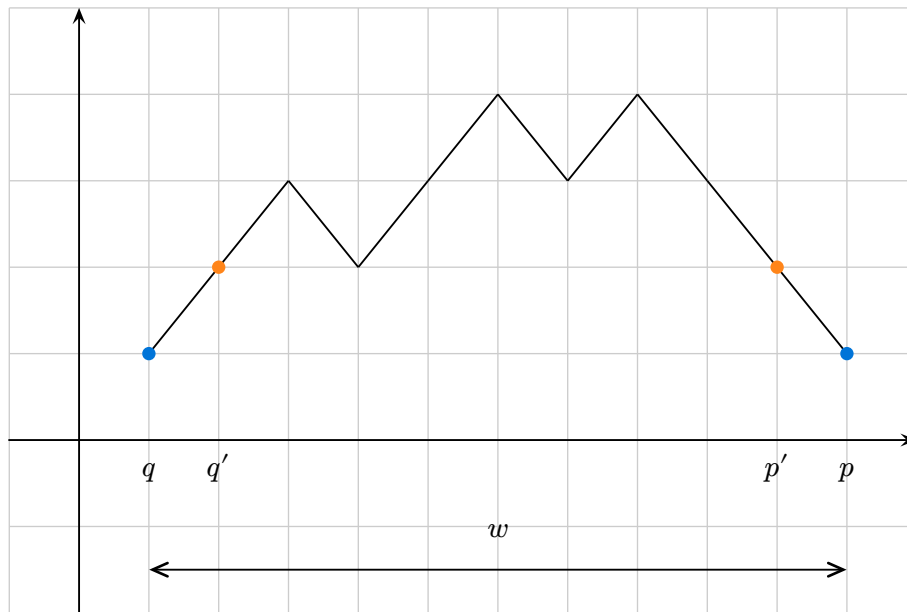
$$\forall (p, -) \in \delta(q, \varepsilon, A) \quad [qAp] \rightarrow \varepsilon.$$

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

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

Ora facciamo le costruzioni **induttive**.

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



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

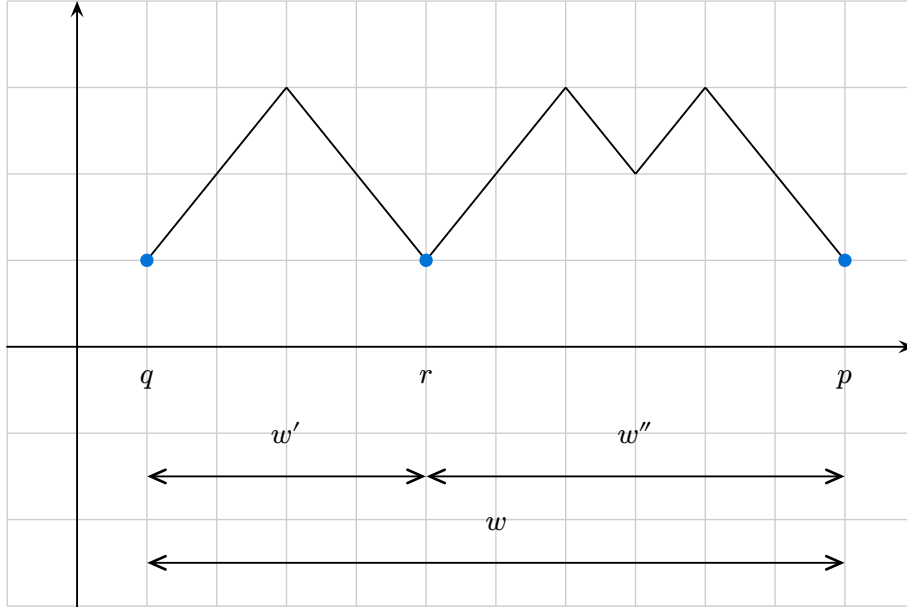
Se chiamiamo w la parte di stringa che stiamo riconoscendo tra la prima A e la seconda A , allora visto che non leggiamo altro durante la push di B stiamo andando a consumare ancora w ma partendo da B . Supponiamo che la mossa che carica la B mi manda nello stato q' e che la mossa che scarica la B parte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

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

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

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

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



Come regole aggiungiamo

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

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

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

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

Per far sì che ciò accada, dobbiamo imporre le regole

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

Lemma 2.3.2.1: Vale

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

Lemma 2.3.2.2: Vale

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

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

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

3. Forme normali per le grammatiche di tipo 2

Nel Capitolo 2 abbiamo visto una **forma normale** per i PDA per facilitare una dimostrazione già di suo molto pesante. Ora vediamo delle **forme normali** per le grammatiche di tipo 2.

Abbiamo detto che le produzioni sono nella forma

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

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**.

3.1. FN di Greibach

La prima forma normale che vediamo è quella di Greibach.

3.1.1. Definizione

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

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

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. Inoltre, con la FN di Greibach possiamo costruire un PDA leggermente più semplice.

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 avanzeranno 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.

3.1.2. Esempio

Esempio 3.1.2.1: Modifichiamo leggermente la grammatica che abbiamo visto nell'Esempio 2.2.1 del Capitolo 2 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	$abbaa$	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	$abbaa$	$abbU$	Come prima
U A	$abbbaa$	$abbUA$	Come prima
A A	$abbbaa$	$abbbaAA$	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.

3.2. Forma normale di Chomsky

La seconda e ultima forma normale che vediamo è la **FN di Chomsky**, forma utile e maneggevole per alcune dimostrazioni che faremo.

3.2.1. Definizione

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

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

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

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

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

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

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

3.2.2. Costruzione

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

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

FN di Chomsky

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

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

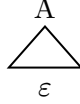
3.2.2.1. Eliminazione delle ε -produzioni

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

Definizione 3.2.2.1.1 (*Variabile cancellabile*): Una variabile A è **cancellabile** se

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

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



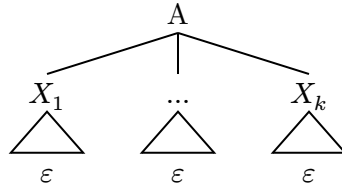
Banalmente, una variabile è cancellabile se nella grammatica ho una regola nella forma

$$A \rightarrow \varepsilon.$$

Se però non abbiamo questa produzione, ma abbiamo delle regole

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

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



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

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

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

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

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

Vale ovviamente la catena

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

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

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

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

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

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

Esempio 3.2.2.1.1: Data la regola di produzione

$$A \rightarrow BCaD$$

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

In questo caso possiamo:

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

Le produzioni che otteniamo sono

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

Vediamo un esempio un po' tedioso.

Esempio 3.2.2.1.2: Data la regola di produzione

$$A \rightarrow CDE$$

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

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

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

3.2.2.2. Eliminazione delle produzioni unitarie

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

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

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

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

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

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

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

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

3.2.2.3. Eliminazione dei simboli inutili

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

Definizione 3.2.2.3.1 (*Simbolo utile*): Un simbolo $X \in V \cup \Sigma$ è **utile** se:

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

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

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

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

Riassunto in una sola sentenza, possiamo dire che

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

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

$$A \rightarrow \alpha$$

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

3.2.2.4. Eliminazione dei terminali

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

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

Esempio 3.2.2.4.1: Date le regole di produzione

$$A \rightarrow AaabcC \mid bC \mid bb$$

cerchiamo di applicare l'eliminazione dei terminali.

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

$$A \rightarrow AX_aX_aX_bC \mid X_bC \mid X_bX_b$$

$$X_a \rightarrow a$$

$$X_b \rightarrow b.$$

Se avessi avuto anche la regola

$$C \rightarrow b$$

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

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

3.2.2.5. Smontaggio delle produzioni

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

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

Infatti, le produzioni:

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

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

Esempio 3.2.2.5.1: Data la produzione

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

la andiamo a smontare creando le produzioni

$$A \rightarrow B_1 Z_1$$

$$Z_1 \rightarrow B_2 Z_2$$

$$Z_2 \rightarrow B_3 Z_3$$

$$Z_3 \rightarrow B_4 B_5.$$

3.2.3. Esempio

Applichiamo questa costruzione ad un esempio.

Esempio 3.2.3.1: Costruire la FN di Chomsky a partire da queste produzioni:

$$S \rightarrow aB \mid bA$$

$$A \rightarrow a \mid aS \mid bAA$$

$$B \rightarrow b \mid bS \mid aBB.$$

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

Partiamo quindi con il cancellare tutti i terminali:

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

Infine, smontiamo le catene di variabili che abbiamo ottenuto:

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

AGGIUNGI CAPITOLO + ESE + CAPITOLO

4. Pumping lemma

Prima di iniziare con il topic di questo capitolo facciamo qualche **esempio**.

Esempio 4.1: Definiamo il linguaggio

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

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

Esempio 4.2: Definiamo ora il linguaggio

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

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

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

4.1. Prerequisiti per il pumping lemma

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

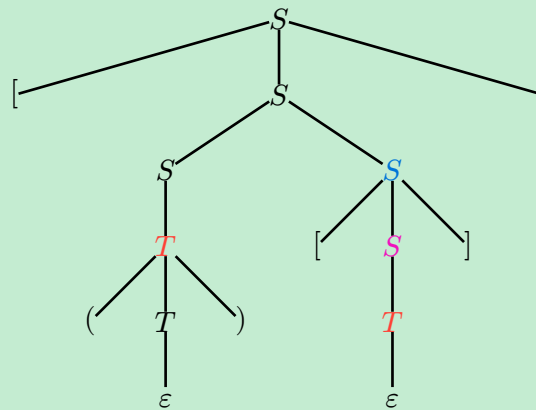
Esempio 4.1.1: Le regole di produzione

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

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

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

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

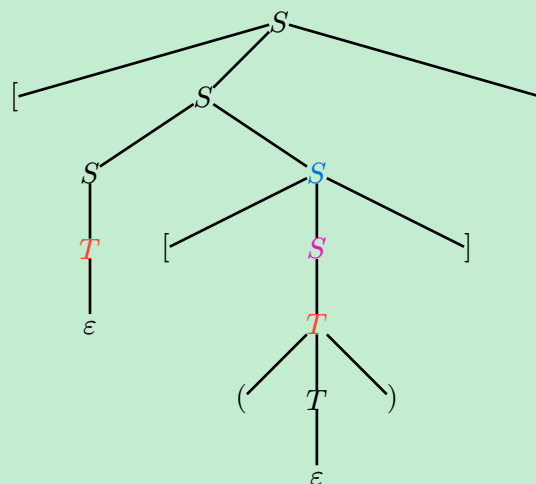


Questo è l'albero di derivazione della stringa

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

Facciamo un paio di **osservazioni**.

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



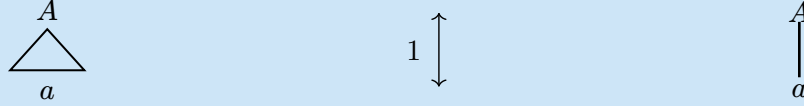
Con questa operazione otteniamo la stringa

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

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

[Passo base: $k = 1$]

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

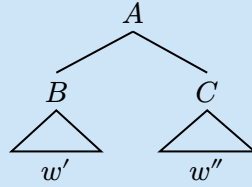


Avendo questa profondità, e generando quindi solo un terminale, la lunghezza delle stringhe è

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

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

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



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

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

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

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

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

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

■

4.2. Pumping lemma per i CFL

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

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

Vediamo la definizione formale e la dimostrazione.

Lemma 4.2.1 (*Pumping lemma per i CFL*): Sia L un linguaggio CF. Allora $\exists N > 0$ tale che $\forall z \in L$ tale che $|z| \geq N$ essa può essere scritta come $z = uvwxy$ tale che:

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

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

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

Dimostrazione 4.2.1.1: Abbiamo a disposizione un linguaggio CF, dal quale possiamo ricavare facilmente una grammatica CF e, per costruzione, una grammatica in FN di Chomsky. Sia quindi $G = (V, \Sigma, P, S)$ una grammatica in FN di Chomsky per $L/\{\varepsilon\}$.

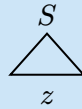
Fissato $k = |V|$, definiamo

$$N = 2^k.$$

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

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

Questo albero, molto piccolino, è nella forma:



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

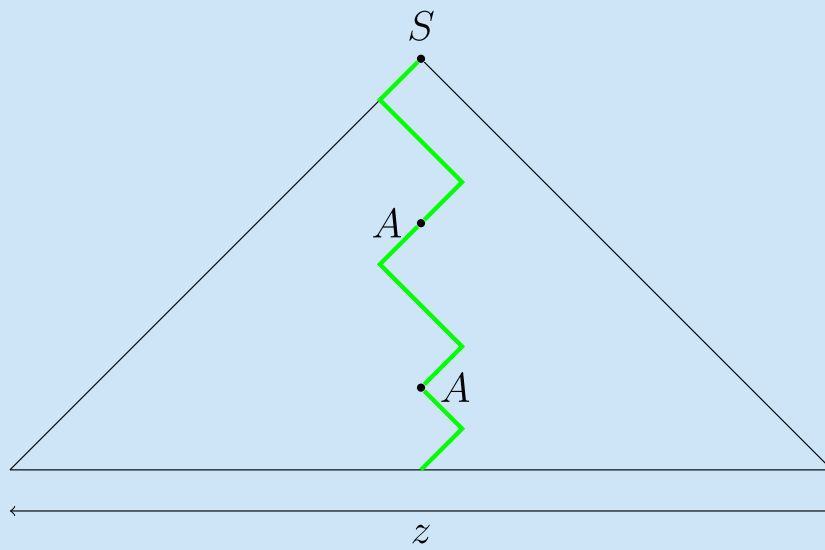
$$|z| \leq 2^{h-1}.$$

Unendo le due disuguaglianze otteniamo che

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

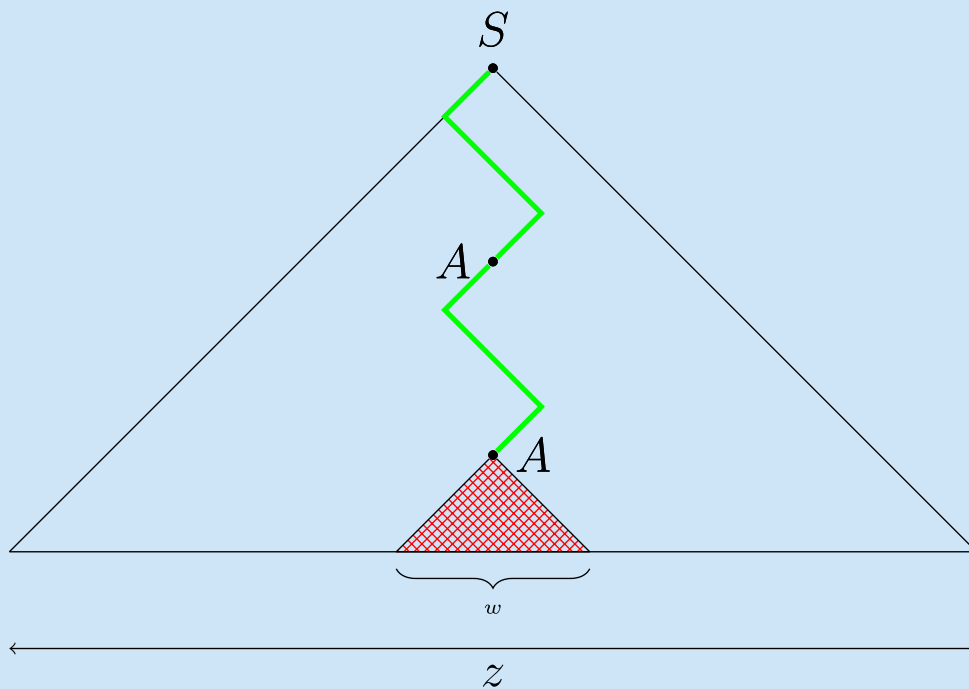
La profondità di un albero è il cammino massimo dalla radice alla foglia più lontana. Visto che questa profondità è almeno $k + 1$, stiamo attraversando $k + 1$ archi e quindi $k + 2$ nodi.

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

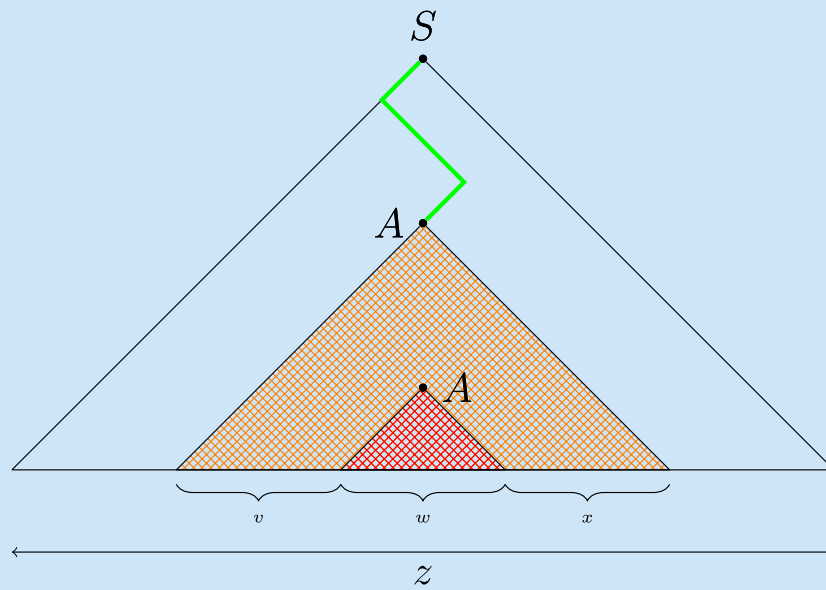


Nella figura precedente abbiamo indicato con due pallini la variabile A che viene ripetuta durante il cammino dal fondo verso la radice. Ora iniziamo la divisione in fattori.

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

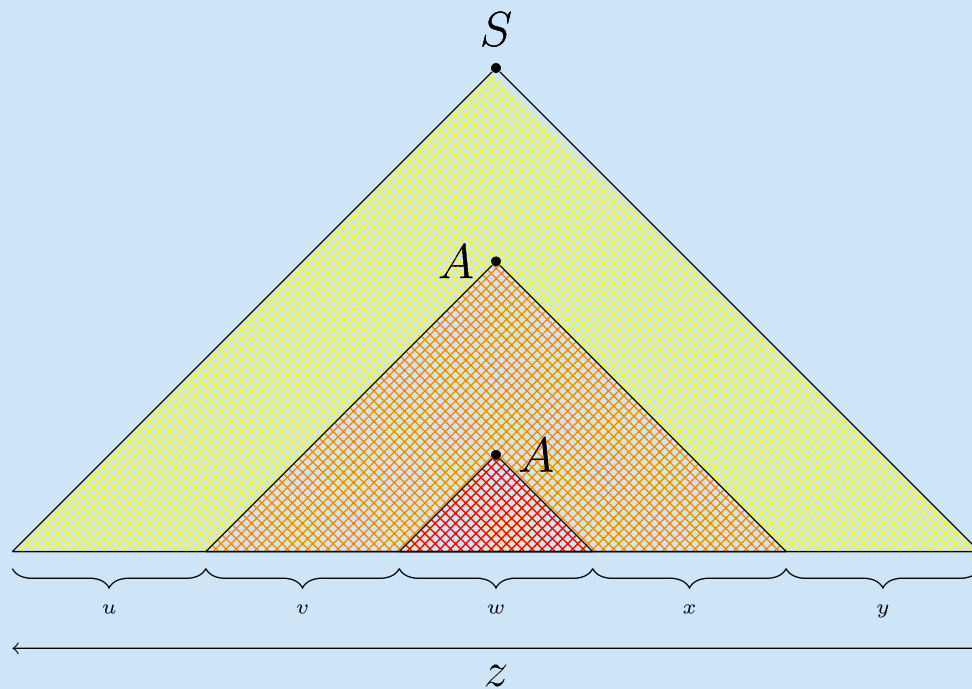


Consideriamo ora l'albero che parte dalla A più sopra: esso genera un altro fattore di z , che contiene quello precedente più due fattori esterni, che chiamiamo v e x , ovvero $A \stackrel{*}{\Rightarrow} vwx$.



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

$$S \Rightarrow^* uvwxy.$$



Abbiamo quindi mostrato che esiste la decomposizione.

[TERZO PUNTO]

Cosa osserviamo dal disegno?

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

$$S \xRightarrow{*} uAy.$$

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

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

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

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

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

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

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

[SECONDO PUNTO]

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

Qua abbiamo due casi:

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

[PRIMO PUNTO]

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

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

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

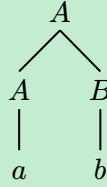
Vediamo un esempio per capire meglio la dimostrazione del secondo punto del pumping lemma.

Esempio 4.2.1: Abbiamo una grammatica in FN di Chomsky con le regole di produzione

$$A \rightarrow a \mid AB$$

$$B \rightarrow b.$$

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



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

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

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

4.3. Applicazioni del pumping lemma

Andiamo ad applicare il pumping lemma al linguaggio di prima.

Esempio 4.3.1: Sia quindi

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

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

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

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

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

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

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

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

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

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

Contiamo il numero di terminali che abbiamo:

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

- $\#_c(z') = \#_c(z) - \#_c(vx) = N - \#_c(vx)$ per la parte cancellata.

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

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

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

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

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

Vediamo un altro **linguaggio a blocchi**.

Esempio 4.3.2: Definiamo ora

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

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

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

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

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

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

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

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

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

Andiamo per **casi**:

- se $vwx \in a^+$ allora pompriamo la stringa con $i = 2$ e otteniamo una stringa con un numero di a più grande del numero di b e di c , ma b deve essere uguale al massimo, quindi questo è un assurdo;
- se $vwx \in c^+$ allora pompriamo nello stesso modo con $i = 2$;
- se $vwx \in b^+$ allora togliamo un po' di b con $i = 0$ per renderle in numero minore del numero massimo, che è N , quindi questo è un assurdo;
- se $vwx \in a^+ b^+$ dobbiamo andare per **casi** per capire dove avviene la divisione tra a e b :
 - se $v \in a^+ b^+$ allora la divisione avviene nel primo fattore; se pompriamo con $i = 2$ otteniamo la stringa $uvvwxy$ formata da una serie di a , da una serie di b ,

poi ancora una serie di a , ma questa operazione ci fa perdere la struttura del linguaggio, quindi questo è un assurdo;

- ▶ se $x \in a^+b^+$ allora la divisione avviene nel terzo fattore; se pompiamo con $i = 2$ otteniamo la stessa perdita di struttura di prima;
- ▶ se $v = a^l \wedge x = b^r$ allora la divisione è nel fattore centrale; visto che questi due fattori assieme non sono vuoti, vuol dire che $l + r > 0$; se andiamo a prendere la stringa uv^iwx^iy noi stiamo aggiungendo $i - 1$ volte un numero l di a e un numero r di b , ovvero

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

Andiamo ancora per **casi**:

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

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

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

Esempio 4.3.3: Definiamo ora

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

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

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

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

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

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

Esempio 4.3.4: Definiamo infine

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

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

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

$$z = a^m.$$

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

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

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

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

Sappiamo inoltre che

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

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

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

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

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

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

5. Lemma di Ogden

5.1. Fail del pumping lemma

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

Esempio 5.1.1: Definiamo il linguaggio

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

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

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

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

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

Abbiamo diversi casi da controllare:

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

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

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

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

$[vwx \in c^+]$

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

Infatti, questo dipende dalla z che abbiamo a disposizione:

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

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

ma allora scegliendo

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

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

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

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

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

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

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

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

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

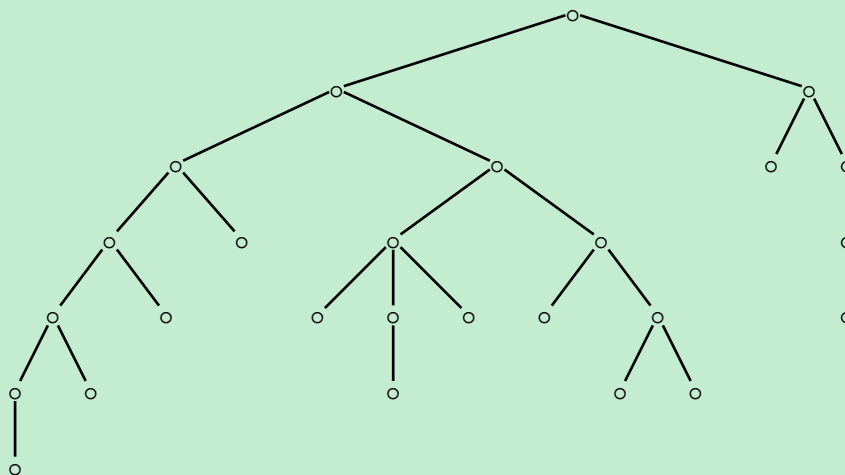
Quindi anche in questo caso ci sono casi favorevoli ma non tutti lo sono.

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

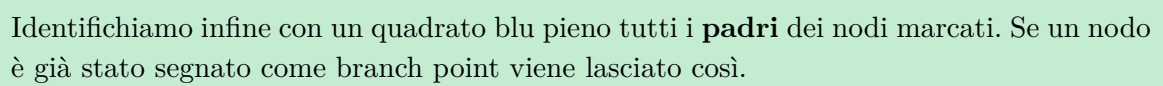
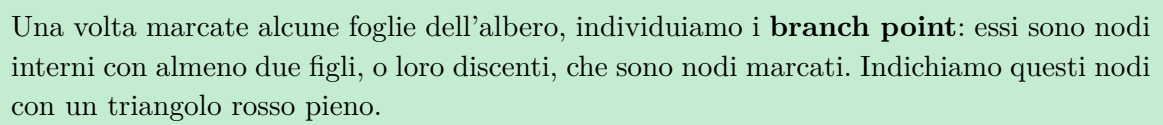
5.2. Lemma di Ogden

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

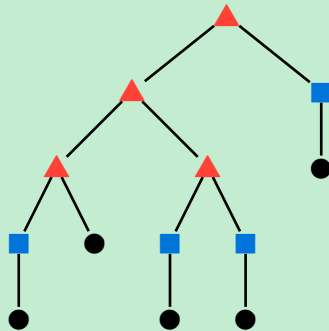
Esempio 5.2.1: Ci viene dato un albero di derivazione di una grammatica generica.



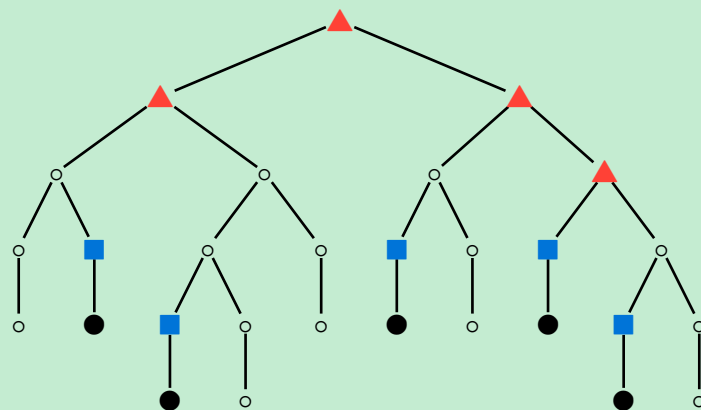
Andiamo a **marcare**, con un pallino nero grosso, alcune foglie dell'albero.



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

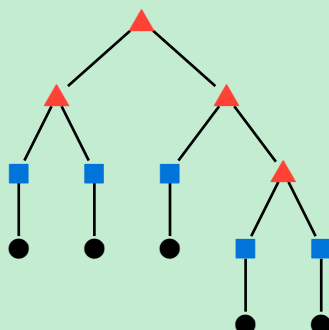


Esempio 5.2.2: Vediamo ora un albero in FN di Chomsky, già con nodi marcati e speciali.



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

Ora vediamo l'albero semplificato.



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

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

Lemma 5.2.1: Sia $G = (V, \Sigma, P, S)$ una grammatica in FN di Chomsky. Sia

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

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

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

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

Dimostrazione 5.2.1.1: Si dimostra per induzione, come il lemma del capitolo scorso.

■

Introduciamo finalmente il **lemma di Ogden**.

Lemma 5.2.2 (*Lemma di Ogden*): Sia $L \subseteq \Sigma^*$ un linguaggio CFL. Allora $\exists N > 0$ tale che $\forall z \in L$ in cui vengono marcate almeno N posizioni può essere decomposta come $z = uvwxy$ tale che:

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

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

Dimostrazione 5.2.2.1: La dimostrazione di questo teorema è analoga a quella del pumping lemma, ma ragiona sull'albero semplificato associato a quello di derivazione di z . ■

5.3. Applicazioni

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

Esempio 5.3.1: Sia di nuovo

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

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

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

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

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

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

$$vwx \in a^+ \quad | \quad vwx \in a^+ b^+ \quad | \quad vwx \in a^+ b^N c^+.$$

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

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

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

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

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

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

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

Esempio 5.3.2: Definiamo ora

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

Possiamo vedere questo linguaggio come

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

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

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

6. Ambiguità

6.1. Esempi

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

Esempio 6.1.1: Definiamo il linguaggio

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

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

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

Esempio 6.1.2: Diamo una grammatica per il linguaggio precedente. Le produzioni sono

$$\begin{aligned} S &\rightarrow S_1 C \mid A S_2 \\ S_1 &\rightarrow a S_1 b \mid \varepsilon \\ S_2 &\rightarrow b S_2 c \mid \varepsilon \\ A &\rightarrow a A \mid \varepsilon \\ C &\rightarrow c C \mid \varepsilon. \end{aligned}$$

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

$$\begin{aligned} S_1 &\stackrel{*}{\Rightarrow} a^n b^n \\ S_2 &\stackrel{*}{\Rightarrow} b^n c^n \\ A &\Rightarrow a^k \\ C &\stackrel{*}{\Rightarrow} c^k. \end{aligned}$$

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



6.2. Definizione

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

Definizione 6.2.1 (*Grado di ambiguità di una stringa*): Sia $G = (V, \Sigma, P, S)$ una grammatica CF. Sia $w \in \Sigma^*$. Chiamiamo **grado di ambiguità** di w rispetto a G il numero di alberi di derivazione di w , oppure il numero di derivazioni leftmost di w .

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

Definizione 6.2.2 (*Grado di ambiguità di una grammatica*): Il **grado di ambiguità** di una grammatica G è il massimo grado di ambiguità delle stringhe $w \in \Sigma^*$.

Il concetto di **ambiguità** è legato al **non determinismo**: abbiamo visto nell'equivalenza tra grammatiche di tipo 2 e automi a pila che questi ultimi potevano simulare le derivazioni leftmost della grammatica. Se ad un certo punto la grammatica ha più derivazioni leftmost che mi portano poi nella stessa stringa allora stiamo introducendo del non determinismo. Viceversa, quando guardavamo le computazioni possibili in un automa a pila e dovevamo generare le regole di produzione, quando eravamo di fronte ad una scelta dovevamo generare delle regole ambigue.

Definizione 6.2.3 (*Grado di ambiguità di un automa a pila*): Il **grado di ambiguità** di un automa a pila è il numero di computazioni accettanti.

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

Definizione 6.2.4 (*Grammatiche inerentemente ambigue*): Sia L un CFL. Allora L è **inerentemente ambigua** se ogni grammatica CF per L è ambigua.

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

Lemma 6.2.1 (*Lemma di Ogden*): Sia $G = (V, \Sigma, P, S)$ una grammatica CF. Allora $\exists N$ tale che $\forall z \in L$ in cui sono marcate almeno N , possiamo scrivere z come $z = uvwxy$ con:

1. vwx contiene al più N posizioni marcate;
2. vx contiene almeno una posizione marcata;
3. $\exists A \in V$ tale che
 - $S \xRightarrow{*} uAy$,
 - $A \xRightarrow{*} vAx$,
 - $A \xRightarrow{*} w$,

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

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

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

Dimostrazione 6.2.1.1: La dimostrazione cambia leggermente dalla scorsa volta.

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

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

e poi la dimostrazione va avanti allo stesso modo. ■

Riprendiamo l'esempio precedente per il discorso sull'ambiguità.

Esempio 6.2.1: Definiamo il linguaggio

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

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

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

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

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

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

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

Teorema 6.2.1: Il linguaggio L dell'Esempio 6.2.1 è inerentemente ambiguo.

Dimostrazione 6.2.1.2: Dobbiamo dimostrare che ogni grammatica G che genera L è ambigua, quindi esiste almeno una stringa in ogni G che è generata in almeno due modi.

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

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

Definiamo la stringa

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

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

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

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

Di questa stringa sappiamo che

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

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

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

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

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

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

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

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

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

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

Riprendiamo la terza condizione, considerando stringhe generiche nella forma

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

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

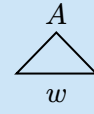
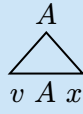
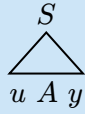
$$(i-1)j = m! \implies i = \frac{m!}{j} + 1$$

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

Con questa imposizione otteniamo la stringa

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

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



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

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

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

Quello che otteniamo è che:

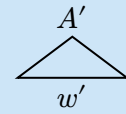
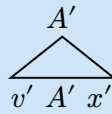
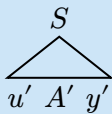
- i fattori v' e x' di z' sono formati rispettivamente dalle sole b e dalle sole c , ovvero sono le stringhe

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

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

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

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



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

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

Nel caso del linguaggio Esempio 6.2.1, il **grado di ambiguità** è 2. In alcuni casi, il grado di ambiguità cresce in base alla lunghezza della stringa che si sta riconoscendo, rendendo di fatto **infinito** il grado della grammatica e/o del linguaggio.

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

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

6.3. Ambiguità e non determinismo

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

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

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

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

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

definiti da una funzione di transizione

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

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

Definizione 6.3.1 (*PDA deterministico*): Il PDA M è **deterministico** se:

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

In poche parole, le due condizioni ci dicono che:

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

Abbiamo visto due diverse **accettazioni** per gli automi a pila, e abbiamo dimostrato che nel caso non deterministico queste sono equivalenti. Nella trasformazione da stati finali a pila vuota, ogni volta che si finiva in uno stato finale si scommetteva di aver finito l'input svuotando la pila, ma lo facendo non deterministicamente. La trasformazione da pila vuota a stati finali invece ogni volta che svuotava la pila andava in uno stato finale, ma questo non introduceva non determinismo perché facevamo una pura simulazione e aggiungevamo regole che non interferivano tra loro.

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

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

Esempio 6.3.1: Definiamo il linguaggio regolare

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

che possiamo riconoscere tranquillamente con un DFA a due stati.

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

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

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

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

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

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

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

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

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

Esempio 6.3.2: Sia L il linguaggio delle palindrome pari, ovvero

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

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

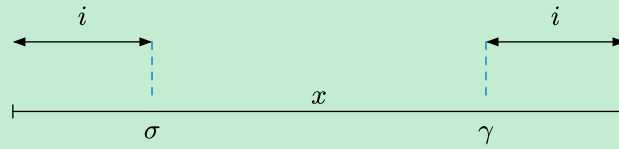
Anche se gli automi per questo linguaggio sono non deterministici, mi va bene una sola scommessa se la stringa è palindroma, quindi non abbiamo ambiguità.

Vediamo una grammatica G per L :

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

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

Esempio 6.3.3: Vediamo il complemento del linguaggio precedente, ovvero il linguaggio delle stringhe nelle quali esiste almeno una posizione alla stessa distanza dai bordi in cui i caratteri sono diversi.



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

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

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

Possiamo evitare l'ambiguità in questo automa?

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

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

Esempio 6.3.4: Per sfizio scriviamo L^C in termini di grammatica.

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

Le regole di produzione sono

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

$$T \longrightarrow aUb \mid bUa$$

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

7. Operazioni tra linguaggi

In questo capitolo discutiamo le **proprietà di chiusura** dei linguaggi CFL e DCFL.

Introduciamo subito due linguaggi che ci serviranno per fare dei controesempi.

Definizione 7.1 (*Linguaggi comodi*): Definiamo il **linguaggio**

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

e il **linguaggio**

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

entrambi DCFL, molto facile da dimostrare.

7.1. Operazioni insiemistiche

Partiamo con le **operazioni insiemistiche**.

7.1.1. Unione

7.1.1.1. CFL

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

Siano

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

due grammatiche CF. Creiamo la grammatica

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

formata da:

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

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

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

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

7.1.1.2. DCFL

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

Esempio 7.1.1.2.1: Prendiamo i due linguaggi definiti nella Definizione 7.1.

Abbiamo detto che sono entrambi DCFL, ma la sua unione

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

deve essere riconosciuta da un automa non deterministico.

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

7.1.2. Intersezione

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

7.1.2.1. CFL

Esempio 7.1.2.1.1: Prendiamo ancora i due linguaggi della Definizione 7.1.

Definiamo il linguaggio

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

che abbiamo visto non essere nemmeno CFL.

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

7.1.2.2. DCFL

Vale lo stesso discorso dei CFL.

7.1.3. Intersezione con un regolare

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

7.1.3.1. CFL

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

7.1.3.2. DCFL

Stesso discorso per i DCFL.

7.1.4. Complemento

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

7.1.4.1. CFL

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

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

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

Esempio 7.1.4.1.1: Definiamo il linguaggio

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

Questo linguaggio è CFL, ma il suo complemento

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

invece non è CFL.

Esempio 7.1.4.1.2: Al contrario, dato il linguaggio

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

che non è CFL, se ne calcoliamo il complemento questo è CFL.

7.1.4.2. DCFL

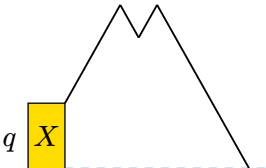
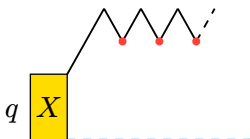
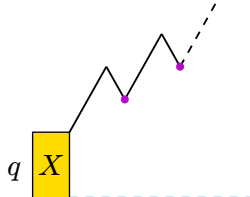
Nel caso deterministico non possiamo usare De Morgan perché non siamo chiusi l'unione, quindi nessun ragionamento di quel tipo ha senso.

Incredibilmente, i DCFL **sono chiusi** rispetto all'operazione di complemento.

Nei linguaggi regolari ci bastava **completare** l'automa e poi invertire «banalmente» gli stati finali con i non finali e viceversa, ma qui non è così facile perché non siamo sempre sicuri che l'automa arrivi in fondo alla sua computazione. Inoltre, nei DCFL abbiamo a disposizione le ε -mosse, che non possiamo togliere perché perdiamo potenza, a differenza degli automi a stati finali dove si manteneva lo stesso potere riconoscitivo.

Ok teniamo le ε -mosse, perché sono fastidiose? Perché l'automa, con una sequenza di ε -mosse, potrebbe entrare in loop infinito e quindi non accettare la stringa. Potremmo fregarcene, ma invece ci interessa molto, perché questa situazione di non accettazione deve essere presa in considerazione ed essere accettata.

Che possibili casi abbiamo durante una sequenza di ε -mosse? Supponiamo di essere nello stato q e di avere sulla pila il carattere X , senza mosse che possono leggere l'input.

Nessun loop	Loop sullo stesso piano	Loop crescente
		

In ordine abbiamo:

- una sequenza di ε -mosse che poi cancella il simbolo X sulla pila;
- una sequenza di ε -mosse che ogni tanto ritorna in una configurazione con stato p e Y sulla pila alla stessa altezza della configurazione analoga precedente;

- una sequenza di ε -mosse che ogni tanto ritorna in una configurazione con stato p e Y sulla pila ad altezza maggiore della configurazione analoga precedente.

Ci interessa sapere queste informazioni, e data la funzione di transizione è possibile conoscere queste informazioni per ogni coppia di stato q e carattere X .

Vediamo quindi come costruire questo automa per il complemento, anche se dobbiamo passare per molti passaggi intermedi. Buona fortuna.

Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un automa a pila deterministico che accetta il linguaggio L . Trasformiamo M nell'automa M' , sempre per L , che ha la proprietà di **non finire mai in loop**, ovvero riesce sempre a leggere tutto l'input. Inoltre, M' risulterà essere ancora deterministico.

Costruiamo quindi l'automa

$$M' = (Q', \Sigma, \Gamma', \delta', q'_0, X_0, F')$$

definito da:

- Q' **insieme degli stati** formato da quelli di M e da tre nuovi stati, che ci permetteranno di fare il solito truccaccio di riempire la pila e di evitare i loop, ovvero

$$Q' = Q \cup \{q'_0, d, f\};$$

- Γ' **alfabeto di lavoro** che aggiunge solo il carattere del truccaccio, ovvero

$$\Gamma' = \Gamma \cup \{X_0\};$$

- F' **insieme degli stati finali** che aggiunge il nuovo stato appena aggiunto, ovvero

$$F' = F \cup \{f\}.$$

La funzione di transizione viene arricchita di un sacco di mosse, che ora vediamo in ordine.

Prima di tutto facciamo il classico truccaccio, ovvero lasciamo qualcosa sotto la pila così che M' non si blocchi se durante la simulazione di M quest'ultimo dovesse svuotare la pila. Aggiungiamo quindi la regola

$$\delta'(q'_0, \varepsilon, Z_0) = (q_0, Z_0 X_0)$$

che appunto mette il tappo in fondo e ci permette di iniziare la simulazione di M .

Se in una certa configurazione non abbiamo ε -mosse o mosse normali a disposizione allora finiamo in uno stato trappola, il **death state**, ovvero

$$\forall q \in Q \quad \forall a \in \Sigma \cup \{\varepsilon\} \quad \forall X \in \Gamma \quad \delta(q, a, X) = \emptyset \implies \delta'(q, a, X) = (d, X).$$

Se ad un certo punto troviamo X_0 sulla pila vuol dire che quest'ultima è stata svuotata da M , quindi l'automa si è bloccato e con M' devo andare nel death state, ovvero

$$\forall a \in \Sigma \quad \delta'(q, a, X_0) = (d, X_0).$$

Nello stato trappola leggo l'input per intero senza toccare altro, quindi

$$\forall a \in \Sigma \quad \forall X \in \Gamma \quad \delta'(d, a, X) = (d, X).$$

Se in una certa configurazione ci sono ε -mosse potrei entrare in un loop, ma questa informazione l'abbiamo già calcolata, quindi con la presenza di ε -mosse e di un loop da (q, X) abbiamo

$$\delta'(q, \varepsilon, X) = \begin{cases} (d, X) & \text{se il loop non visita uno stato finale} \\ (f, X) & \text{altrimenti} \end{cases}.$$

Se sono finito nello stato finale è perché ho trovato un loop con stato finale, ma se l'ho trovato non alla fine della stringa devo andare nello stato trappola, quindi

$$\forall a \in \Sigma \quad \forall X \in \Gamma \quad \delta'(f, a, X) = (d, X).$$

In tutti gli altri casi teniamo le mosse che già c'erano, quindi

$$\forall q \in Q \quad \forall a \in \Sigma \cup \{\varepsilon\} \quad \forall X \in \Gamma \quad \delta'(q, a, X) = \delta(q, a, X).$$

Perfetto, ora che abbiamo un automa che non si blocca mai costruiamo un automa per il complemento. Facciamo un po' di renaming per semplicità.

Sia M un DPDA per L che scandisce sempre l'intero l'input. Vogliamo costruire

$$M' = (Q', \Sigma, \Gamma, \delta', q'_0, Z_0, F')$$

per il complemento di M , ovvero quando M risponde **SI** noi rispondiamo **NO** e viceversa.

Ora M arriva sempre in fondo all'input, ma può fare comunque ε -mosse alla fine. Se durante queste ultime sequenze visitiamo almeno uno stato finale dobbiamo rifiutare, altrimenti accettiamo.

Definiamo quindi M' formato da:

- **Q' insieme degli stati** che memorizza, oltre allo stato nel quale si trova, anche se nella sequenza di ε -mosse che stiamo facendo abbiamo visto o meno uno stato finale. In realtà, ci dice di più questa flag, che il prof chiama un **bit e mezzo**, perché infatti Q' è definito come

$$Q' = Q \times \{y, n, A\}$$

con le flag che indicano rispettivamente se nella sequenza abbiamo visitato uno stato in F , se non l'abbiamo fatto o se non l'abbiamo fatto e non siamo più in grado di fare ε -mosse;

- **F' insieme degli stati finali** formato dalle coppie che hanno come seconda componente la A , perché non sono passato da stati finali e mi sono fermato, quindi

$$F' = Q \times \{A\};$$

- **q'_0 stato iniziale** che dipende dallo stato iniziale vecchio, ovvero

$$q'_0 = \begin{cases} [q_0, n] & \text{se } q_0 \notin F \\ [q_0, y] & \text{se } q_0 \in F \end{cases};$$

- **δ' funzione di transizione** che dati $q \in Q$ e $X \in \Gamma$:
 - se $\delta(q, \varepsilon, X) = (p, \gamma)$ allora posso eseguire delle ε -mosse, quindi in base alla seconda componente degli stati definisco

$$\delta'([q, y], \varepsilon, X) = ([p, y], \gamma)$$

$$\delta'([q, n], \varepsilon, X) = \begin{cases} ([p, y], \gamma) & \text{se } p \in F \\ ([p, n], \gamma) & \text{altrimenti} \end{cases}$$

$$\delta'([q, A], \varepsilon, X) = \emptyset;$$

- se invece nella configurazione corrente ho finito le ε -mosse allora posso avere delle mosse che leggono simboli in input. Potendo fare questa operazione di lettura, dobbiamo dimenticarci dell'ultimo loop eseguito se contiene degli stati finali, quindi se $\delta(q, a, X) = (p, \gamma) \mid a \in \Sigma$ allora

$$\delta'([q, y], a, X) = \begin{cases} ([p, y], \gamma) & \text{se } p \in F \\ ([p, n], \gamma) & \text{altrimenti} \end{cases}$$

Se invece nell'ultimo loop abbiamo terminato il giro senza passare da stati finali passiamo per A , ovvero

$$\begin{aligned} \delta'([q, n], \varepsilon, X) &= ([q, A], X) \\ \delta'([q, A], a, X) &= \begin{cases} ([p, y], \gamma) & \text{se } \varepsilon \in F \\ ([p, n], \gamma) & \text{altrimenti} \end{cases} \end{aligned}$$

Finita, finalmente, questa costruzione senza senso.

7.1.5. Riassunto

Operazione	CFL	DCFL
Unione	✓	✗
Intersezione	✗	✗
Intersezione con un regolare	✓	✓
Complemento	✗	✓

7.2. Operazioni regolari

Vediamo ora le **operazioni regolari**.

7.2.1. Prodotto

La prima operazione regolare, ovvero l'unione, l'abbiamo già analizzata. Vediamo ora il **prodotto**.

7.2.1.1. CFL

Per i CFL è facile creare una grammatica G a partire da due grammatiche G' e G'' CFG con un assioma S e una regola di produzione

$$S \rightarrow S' S''$$

che permetta di produrre le due stringhe separatamente a partire dai loro assiomi.

7.2.1.2. DCFL

Purtroppo, i DCFL **non sono chiusi** rispetto al prodotto.

Esempio 7.2.1.2.1: Prendiamo di nuovo i linguaggi della Definizione 7.1, che sono entrambi deterministici, e creiamo il linguaggio

$$L_0 = L' \cup dL''$$

che è deterministico perché in base al primo carattere capisce subito che linguaggio riconoscere. Creiamo ora il linguaggio

$$L_1 = \{\varepsilon, d\}L_0.$$

In base al carattere che scegliamo di anteporre alle stringhe di L_0 noi possiamo avere un numero di d iniziali differenti. Scriviamo quindi L_1 come l'insieme

$$L_1 = \{d^s a^i b^j c^k \mid s \in \{0, 1, 2\}\}.$$

Analizziamo i vari casi:

- se $s = 0$ allora stiamo scegliendo ε e L' , quindi $i = j$;
- se $s = 2$ allora stiamo scegliendo d e dL'' , quindi $j = k$;
- se $s = 1$ allora:
 - stiamo scegliendo d e L' , quindi $i = j$;
 - stiamo scegliendo ε e dL'' , quindi $j = k$.

Filtriamo alcune stringhe di L_1 calcolando

$$L_1 \cap da^*b^*c^* \stackrel{s=1}{=} \{da^i b^j c^k \mid i = j \vee j = k\}$$

che ovviamente non è DCFL. Ricordandoci che i DCFL sono chiusi rispetto all'intersezione con un regolare, se il linguaggio di destra non è DCFL allora non lo è nemmeno L_1 , che era ottenuto però come concatenazione di due linguaggi DCFL.

Questa cosa è **tristissima**: non siamo chiusi nemmeno con un linguaggio finito, è drammatico.

Fatto stranissimo, se invece concateniamo con un linguaggio regolare a destra si ottiene un linguaggio DCFL, senza senso questa classe di linguaggi.

7.2.2. Star

Vediamo infine la **star** per finire le operazioni regolari.

7.2.2.1. CFL

Nei CFL basta creare una nuova grammatica G a partire da G' CFG con le regole di produzione

$$S \longrightarrow S'S \mid \varepsilon$$

per iniziare a concatenare tante stringhe di G' a nostro piacere.

Un automa invece ogni volta che arriva in uno stato finale fa ripartire la computazione dall'inizio.

7.2.2.2. DCFL

Sfigati come sono, i DCFL non sono chiusi nemmeno per la star di Kleene.

Esempio 7.2.2.2.1: Non ho ben capito l'esempio che ha scritto.

7.2.3. Riassunto

Operazioni	CFL	DCFL
Prodotto	✓	✗

Star	✓	✗
------	---	---

7.3. Considerazioni

Come vediamo, siamo messi molto male rispetto ai linguaggi regolari, dove praticamente tutte le operazioni permettevano la chiusura dei linguaggi regolari.

In questo caso, i CFL ancora ancora si salvano, però perdono il **complemento**, e questo sarà molto fastidioso in futuro. I DCFL non diciamo niente, visto che tolti l'intersezione con i regolari, che non è una vera e propria operazione interna, hanno solo una proprietà di chiusura.

8. 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.

8.1. Pumping lemma

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.

8.2. Linguaggio inerentemente ambiguo

Una seconda tecnica è dimostrare che L è **inerentemente ambiguo**, per far sì che ogni automa per L sia ambiguo e quindi che L è non deterministico.

Esempio 8.2.1: Avevamo visto, con questa tecnica, la dimostrazione di

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

8.3. Proprietà di chiusura

Una terza tecnica è usare le **proprietà di chiusura** rispetto al complemento. Se facciamo vedere che $L^C \notin \text{CFL}$ allora L non può essere DCFL perché questi ultimi sono chiusi rispetto al complemento, ed essendo $L^C \notin \text{CFL}$ allora vale anche $L^C \notin \text{DCFL}$.

Esempio 8.3.1: 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

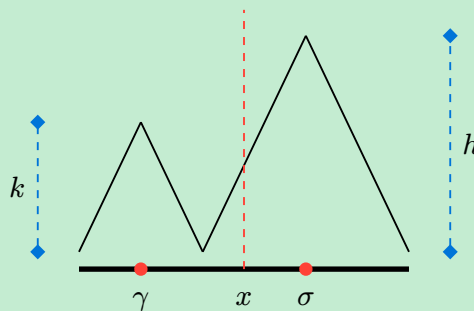
- scarichiamo i k simboli sulla pila leggendo altri k simboli in input, arrivando fino al simbolo iniziale della pila. Con questa mossa abbiamo letto i primi due blocchi di k simboli;

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 8.3.2: Vediamo una grammatica per il linguaggio precedente.

Le regole di produzione sono:

$$\begin{aligned} S &\rightarrow AB \mid BA \mid A \mid B \\ A &\rightarrow aAa \mid aAb \mid bAa \mid bAb \mid a \\ B &\rightarrow aBa \mid aBb \mid bBa \mid bBb \mid b. \end{aligned}$$

Se scegliamo solo una lettera generiamo stringhe dispari, che controlla l'automa a stati finiti. Se scegliamo invece una concatenazione di due lettere allora abbiamo che

$$\begin{aligned} A &\stackrel{*}{\Rightarrow} xAy \Leftrightarrow xay \quad | \quad |x| = |y| \\ B &\stackrel{*}{\Rightarrow} zBv \Leftrightarrow zbv \quad | \quad |z| = |v|. \end{aligned}$$

Ma allora stiamo generando della stringhe

$$S \Rightarrow AB \stackrel{*}{\Rightarrow} \underbrace{xayz}_{\text{stringa}} bv \quad | \quad |x| + |v| = |y| + |z|.$$

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

Esempio 8.3.3: 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 8.3.4: 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.

8.4. Relazione di Myhill-Nerode

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 8.4.1: 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 8.4.1: Definiamo di nuovo il linguaggio

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

Facciamo vedere che

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

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

Prendiamo quindi due stringhe generiche

$$x = x_1 \dots x_n$$

$$y = y_1 \dots y_m$$

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

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

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

$$xz = xx^R \in \text{PAL}$$

$$yz = yx^R = y_1 \dots y_m x_n \dots x_1 \notin \text{PAL}$$

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

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

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

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

Come stringa z scegliamo $z = cx^R$ dove

$$c = \begin{cases} a & \text{se } y_{n+1} = b \\ b & \text{altrimenti} \end{cases}.$$

Se applichiamo questa stringa alle due che abbiamo a disposizione otteniamo

$$xz = xcx^R \in \text{PAL}$$

$$yz = xy_{n+1} \dots y_m cx^R \notin \text{PAL}$$

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.

9. Risultati particolari

9.1. Ricorsione

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

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

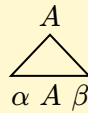
con la classe $\text{LR}(k)$ che indica degli oggetti molto tecnici, poco naturali, che sono usati nei **parser**. Se $k = 1$ allora stiamo considerando direttamente i DCFL.

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

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

In poche parole, esiste una variabile che ha un albero di derivazione in cui sulle foglie ho due stringhe diverse dalla parola vuota:



È importante che entrambe siano diverse dal vuoto:

- se è vuota α abbiamo ricorsione all'inizio, che si può eliminare;
- se è vuota β abbiamo ricorsione in coda, che si può eliminare.

Se anche solo una è vuota non abbiamo più una **vera ricorsione**.

Teorema 9.1.1: 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 9.1.1.1: Se L è un linguaggio CFL e non regolare allora ogni G per L è self-embedding.

9.2. Linguaggio di Dyck

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

Definizione 9.2.1 (*Linguaggio di Dyck*): Definiamo l'alfabeto

$$\Omega_k = \{((1, (2, \dots (k,)_1,)_2, \dots,)_k)\}$$

formato da k tipi di **parentesi**. Questo insieme contiene k parentesi aperte e le k parentesi chiuse corrispondenti, quindi $|\Omega_k| = 2k$.

Il **linguaggio di Dyck**

$$D_k \subseteq \Omega_k^*$$

è l'insieme delle parentesi bilanciate costruite sull'insieme Ω_k .

Ora vediamo un teorema ideato dal nostro amico Chomsky e dal franco-tedesco Schutzenberger.

Teorema 9.2.1 (*Teorema di Chomsky-Schutzenberger*): Dato $L \subseteq \Sigma^*$ un CFL, allora:

- $\exists k > 0$ numero intero,
- \exists morfismo $h : \Omega_k \rightarrow \Sigma^*$,
- $\exists R \subseteq \Omega_k^*$ linguaggio regolare

tali che

$$L = h(D_k \cap R).$$

Questo è un **teorema di rappresentazione** ed è fuori di testa: scegliamo un insieme di parentesi, prendiamo il linguaggio di Dyck corrispondente, lo filtriamo con un linguaggio regolare definito sullo stesso linguaggio, applichiamo un morfismo che trasformi le parentesi in altri caratteri e otteniamo un CFL che abbiamo sotto mano.

Esempio 9.2.1: 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 $\Omega_k = \{(,)_1\}$ e definiamo il morfismo h tale che

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

Tra tutte le stringhe di parentesi tonde bilanciate filtriamo le sequenze in cui le parentesi aperte si trovano prima delle parentesi chiuse, quindi scegliamo

$$R = (^*)^*.$$

Esempio 9.2.2: Se prendiamo L il linguaggio delle parentesi bilanciate, allora scegliamo l'identità come morfismo e come linguaggio regolare quello che fa passare tutto.

Esempio 9.2.3: 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

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

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

$$R = [(1 + (2)^*[]_1 +)_2]^*.$$

Esempio 9.2.4: Definiamo infine il linguaggio PAL delle stringhe palindrome di lunghezza anche dispari. Qua dobbiamo modificare leggermente la soluzione precedente

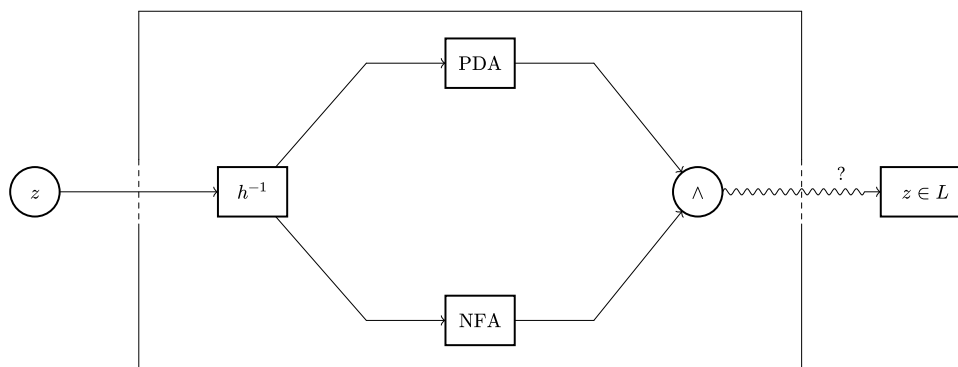
Scegliamo $k = 4$ usando le parentesi definite prima, alle quali aggiungiamo due parentesi, che usiamo per codificare l'eventuale simbolo centrale, che può essere una a o una b , quindi:

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

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 9.2.1](#), 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.

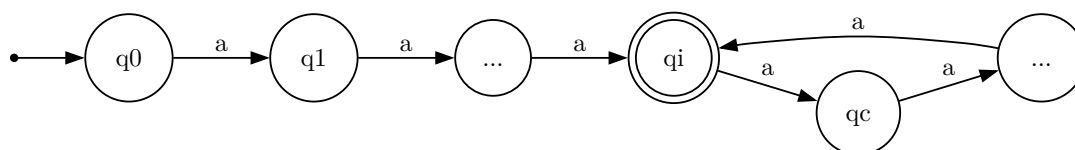
10. Alfabeti unari

In questo capitolo vediamo alcuni risultati con gli **alfabeti unari**: questi sono alfabeti molto particolari formati da un solo carattere, ovvero sono nella forma

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

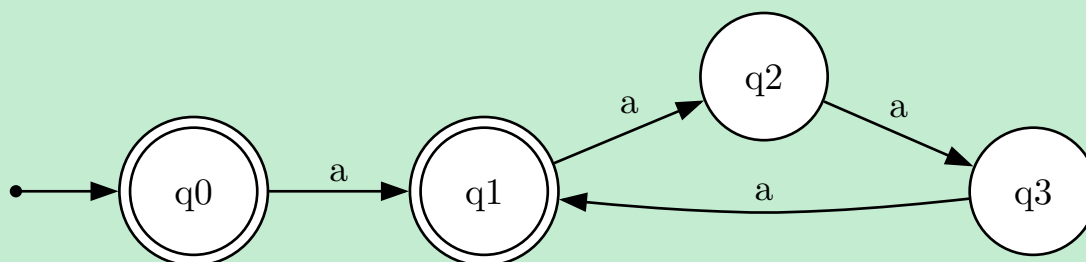
10.1. Linguaggi regolari

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 10.1.1: 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 10.1.2: Definiamo tre linguaggi

$$L_1 = a^{28}(a^3)^*$$

$$L_2 = a^{11}(a^3)^*$$

$$L_3 = a^{37}(a^3)^*$$

Cosa aggiungono questi linguaggi al linguaggio L dell'Esempio 10.1.1?

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**.

10.2. 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

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

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

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

10.3. Teorema di Parikh

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

Definizione 10.3.1 (*Immagine di Parikh sulle stringhe*): Sia $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ un alfabeto. L'**immagine di Parikh** sulle stringhe è la funzione

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

tale che

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

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

Esempio 10.3.1: 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 10.3.2 (*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 10.3.2: 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 10.3.1 (*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.

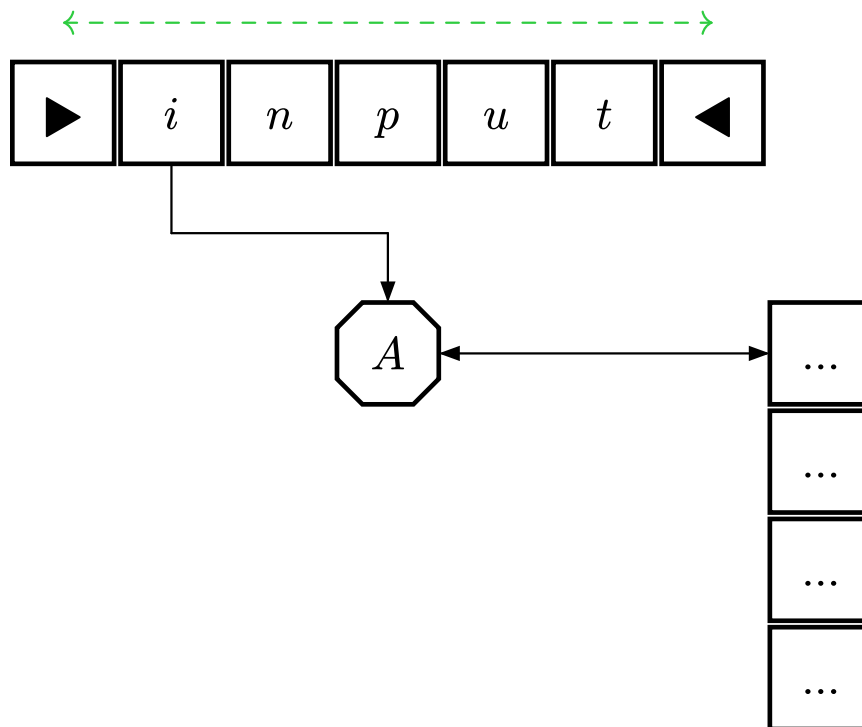
11. Automi a pila two-way

È arrivato il momento di modificare un po' la macchina che stiamo usando da forse troppo tempo, aggiungendo alcune funzionalità che faranno solo che bene.

11.1. 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.

11.2. Esempi

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

Esempio 11.2.1: 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 11.2.2: 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 11.2.3: 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 11.2.4: 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

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

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

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

12. Problemi di decisione

Vediamo in questo capitolo qualche **problema di decisione**. Per ora vedremo i problemi a cui sappiamo rispondere con quello che sappiamo, questo perché alcuni dei problemi di decisione richiedono conoscenze delle **macchine di Turing**, che vedremo nell'ultima parte delle dispense.

12.1. 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

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

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 si considerano sottoclassi per avvicinarsi a complessità lineari.

12.2. Linguaggio vuoto e infinito

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

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

Teorema 12.2.1: Sia $L \subseteq \Sigma^*$ un CFL, e sia N la costante del pumping lemma per L . Allora:

1. $L \neq \emptyset \iff \exists z \in L \mid |z| < N$;
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.

12.3. 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.

12.4. Altri problemi

Per vedere altri problemi di decisione sui CFL guardate il capitolo sui **problemi di decisione delle macchine di Turing**.

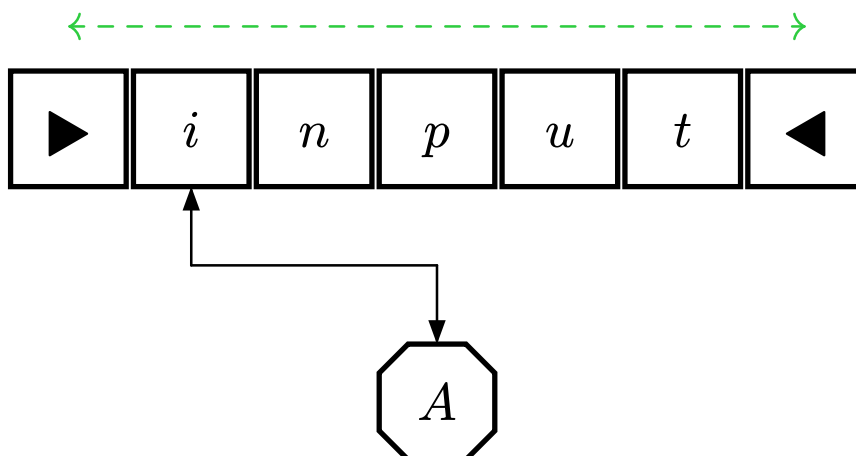
Parte IV — Linguaggi context-sensitive

1. Automi limitati linearmente

Toccata e fuga nel campo dei **linguaggi context-sensitive**, visto che li vedremo in pochissimi capitoli di questa magica dispensa.

1.1. Definizione

I linguaggi di tipo 1 hanno come modello riconoscitivo gli **LBA**, o **Linear Bounded Automata**. In poche parole, sono degli automi a stati finiti two-way con la testina che però può scrivere sul nastro.



Gli LBA vedono il nastro come se fosse una memoria, nella quale possono anche scrivere. Visto che possiamo scrivere solo sul nastro, lo spazio a nostra disposizione rimane **limitato dall'input**.

Questo modello accetta tutti e soli i **linguaggi context-sensitive** CS.

1.2. Classi di complessità associate

Visto che trattiamo dello spazio, qua possiamo definire qualche **classe di complessità**.

Gli **LBA** usano spazio pari alla lunghezza dell'input, ovvero vale

$$\text{LBA} = \text{CS} = \text{NSPACE}(n).$$

In poche parole, gli LBA corrispondono alle **macchine riconosctrici non deterministiche in spazio lineare**, o qualcosa di simile.

Consideriamo ora i **DLBA**, ovvero i **CS deterministici**, che anche loro usano spazio pari alla lunghezza dell'input, ovvero vale

$$\text{DLBA} = \text{PSPACE}(n).$$

In poche parole, i DLBA corrispondono alle **macchine riconosctrici deterministiche in spazio lineare**, o qualcosa di simile.

Quello che non sappiamo è la questione

$$\text{PSPACE}(n) \stackrel{?}{=} \text{NSPACE}(n).$$

Si sa che dal non determinismo si passa al determinismo pagando un fattore quadratico, ovvero

$$\text{NSPACE}(n) \subseteq \text{PSPACE}(n^2).$$

Si è congetturato che questi insiemi sono diversi, ma non è ancora stato dimostrato.

1.3. Esempi

Riprendiamo gli esempi che abbiamo visto nei 2DPDA e cerchiamo di rifarli.

Esempio 1.3.1: Definiamo

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

Come possiamo riconoscere L con un LBA?

Eseguiamo in ordine i seguenti passi:

1. troviamo una a , la marchiamo con una X e cerchiamo una b ;
2. troviamo una b , la marchiamo con una X e cerchiamo una c ;
3. troviamo una c , la marchiamo con una X e ripartiamo dal punto 1.

Con un controllo a stati finiti molto veloce controlliamo prima di tutto di avere tutte a , poi tutte b e infine tutte c . Dopo questo controllo partiamo con la logica appena descritta.

Ad esempio, se dovessimo riconoscere la stringa $z = aaabbbccc$ il nastro si presenta nel seguente modo durante le iterazioni:

a	a	a	b	b	b	c	c	c
X	a	a	b	b	b	c	c	c
X	a	a	X	b	b	c	c	c
X	a	a	X	b	b	X	c	c
X	X	a	X	b	b	X	c	c
X	X	a	X	X	b	X	c	c
X	X	a	X	X	b	X	X	c
X	X	X	X	X	b	X	X	c
X	X	X	X	X	X	X	X	c
X	X	X	X	X	X	X	X	X

Esempio 1.3.2: Definiamo il linguaggio

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

Cerchiamo di fare ancora delle **divisioni** per 2, ma ora lo facciamo lavorando sul nastro.

Andiamo quindi a marcare con una X un carattere a ogni due letti, e facendo così controlliamo anche se le a sono pari o dispari. Una volta arrivati alla fine torniamo all'inizio dal nastro e ripartiamo, ignorando completamente le X che abbiamo messo. Come prima, alla fine del processo deve rimanere una sola a sul nastro.

Vediamo il riconoscimento della stringa $z = a^8$.

<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>X</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>X</i>	<i>a</i>	<i>X</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>X</i>	<i>a</i>	<i>X</i>	<i>a</i>	<i>X</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>X</i>	<i>a</i>	<i>X</i>	<i>a</i>	<i>X</i>	<i>a</i>	<i>X</i>	<i>a</i>
<i>X</i>	<i>X</i>	<i>X</i>	<i>a</i>	<i>X</i>	<i>a</i>	<i>X</i>	<i>a</i>
<i>X</i>	<i>X</i>	<i>X</i>	<i>a</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>a</i>
<i>X</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>a</i>
<i>X</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>X</i>

Tra la terzultima e la penultima riga l'automa scorre fino in fondo l'input, non cancella la seconda *a* perché ne ha appena cancellata una, torna indietro e poi si sposta sull'ultima *a* per cancellarla effettivamente.

Esempio 1.3.3: Definiamo il linguaggio

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

Vediamo prima un **approccio non deterministico**. Cerchiamo di indovinare la metà della stringa, marcandola con una *X* e ricordandoci del carattere marcato. Ci spostiamo poi all'inizio del nastro, controllando l'uguaglianza con il carattere marcato. Se i due caratteri corrispondono ricomincio ma partendo dall'inizio, quindi marco un carattere iniziale, mi sposto a metà, controllo e riparto. Se azzecciamo la metà con una computazione allora abbiamo vinto e riusciamo a riconoscere la stringa in input.

Vediamo ora un **approccio deterministico**. L'idea l'abbiamo praticamente già vista, ma la dobbiamo adattare a questo preciso linguaggio per come è definito.

Prendiamo ad esempio la stringa $z = abbababbab$. Facciamo finta di avere un **secondo nastro** sotto quello che già abbiamo e facciamo quello che abbiamo fatto prima, ovvero ogni due simboli ne andiamo a marcare uno.

<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>b</i>
<i>X</i>		<i>X</i>		<i>X</i>		<i>X</i>		<i>X</i>	

Con questa passata controlliamo come sempre se la stringa ha lunghezza pari o dispari.

Aggiungiamo un **terzo nastro**, nel quale prendiamo tutte le *X* del secondo nastro e le trasciniamo tutte in fondo a sinistra.

a	b	b	a	b	a	b	b	a	b
X		X		X		X		X	
X	X	X	X	X					

Ora che conosco la metà, grazie al terzo nastro, ci spostiamo sul carattere appena dopo l'ultima X e ripeto l'algoritmo non deterministico appena visto.

Tutto bello, ma come facciamo senza nastri? Possiamo usare una sorta di **alfabeto esteso**, ovvero un alfabeto che come simboli ha i simboli che si ottengono concatenando le colonne dei vari nastri impilati. Ad esempio, il primo carattere a prima diventa aX e poi aXX .

Come vediamo, tutto quello che prima abbiamo fatto con i **2DPDA** siamo riusciti a farlo anche ora usando gli **LBA**, tra l'altro tutti **deterministici**.

2. Equivalenza tra LBA e grammatiche di tipo 1

Vediamo, come abbiamo visto anche per gli altri due modelli, l'equivalenza tra **LBA** e grammatiche di tipo 1, ovvero le **grammatiche context-sensitive**.

2.1. Dimostrazione

Prima di tutto, ripassiamo velocemente come sono fatte queste grammatiche.

Una grammatica $G = (V, \Sigma, P, S)$ di tipo 1 ha **produzioni** nella forma

$$\alpha \rightarrow \beta \quad \text{tali che} \quad |\alpha| \leq |\beta|.$$

In poche parole, le produzioni sono **non decrescenti**, e questo ci aiutava con la decidibilità delle grammatiche di tipo 1, e quindi anche delle grammatiche di tipo 2 e di tipo 3.

Teorema 2.1.1: Le grammatiche di tipo 1 sono equivalenti agli automi limitati linearmente.

Dimostrazione 2.1.1.1: Vediamo entrambe le trasformazioni.

[Grammatica \rightarrow LBA]

Data una grammatica, costruiamo un LBA che fa il contrario del processo di produzione: se ho in input una stringa w devo costruire un LBA per capire se $S \xRightarrow{*} w$.

Questo lo facciamo imponendo la macchina a scrivere le forme sentenziali sul nastro, ovvero se sul nastro troviamo un certo β allora lo sostituiamo con un certo α e qualche marcatore per fare da tappo. Ovviamente, il processo è non deterministico, e inoltre non ci fa strabordare perché sappiamo che α non sorpassa β come lunghezza.

[LBA \rightarrow grammatica]

Si può costruire una grammatica data la descrizione di un LBA, fidiamoci. ■

2.2. Determinismo e non determinismo

Sappiamo quindi che le due rappresentazioni sono equivalenti, ma non sappiamo la relazione che esiste tra **determinismo** e **non determinismo**.

3. Operazioni tra linguaggi

Per finire con i **CSL** vediamo le **proprietà di chiusura** di questa classe di linguaggi. Purtroppo vedremo solo le tre **operazioni insiemistiche** base, le altre sono abbastanza pesanti.

	DLBA	LBA
Unione	 Non deterministicamente faccio partire i due automi in parallelo e controllo che almeno uno accetti	 Faccio partire il automa mantenendo l'input sul nastro, se la macchina dice NO faccio partire il secondo e controllo
Intersezione	 Uguale a sopra	 Uguale a sopra
Complemento	 Abbiamo a disposizione il teorema di Immerman-Szelepcsényi	 Risultato sorprendente perché è rimasto aperto per molti anni, ma è stato risolto nell'86/87

Parte V — Linguaggi senza restrizioni

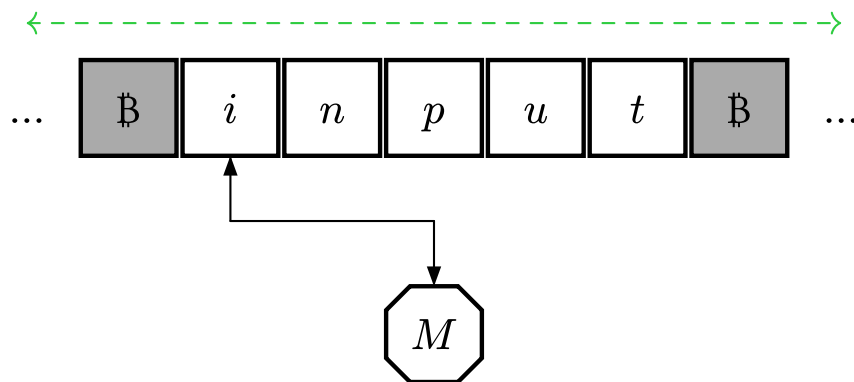
1. Macchine di Turing

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

1.1. Introduzione

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 $\blacktriangleright \blacktriangleleft$ 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 \rightarrow macchina di Turing]

Nelle grammatiche di tipo 1 la condizione sulle produzioni $\alpha \rightarrow \beta$ 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 sfiorare il nastro, ma ora che abbiamo spazio a disposizione lo possiamo sfruttare per fare tutte le derivazioni possibili.

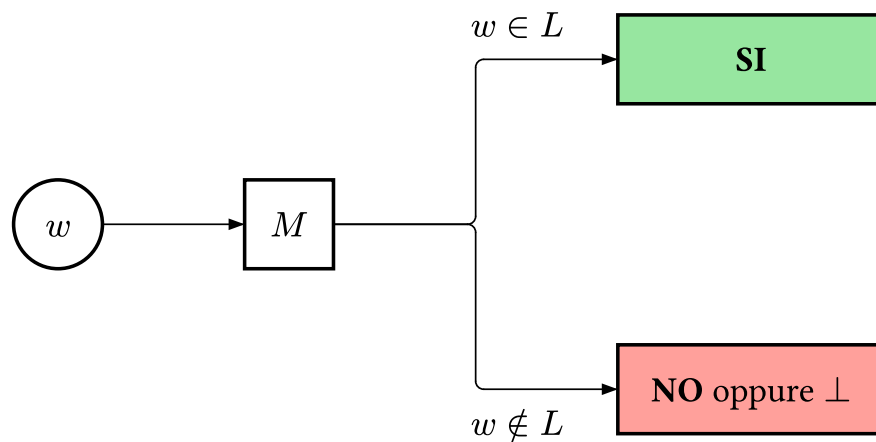
[Macchina di Turing \rightarrow grammatica]

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**.



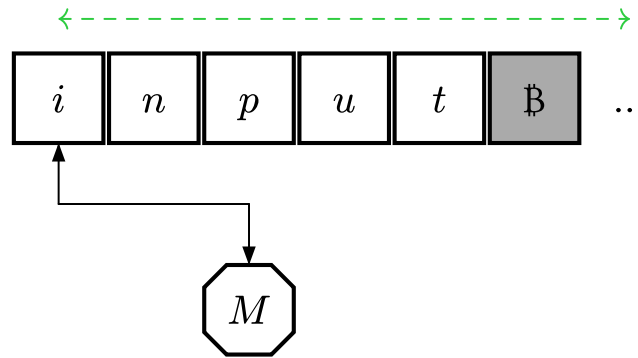
Le macchine di Turing sono un modello in grado di calcolare tutte le **funzioni calcolabili** da un computer. Queste funzioni sono dette **funzioni ricorsive**, ma con «ricorsive» non intendiamo le funzioni che chiamano sé stesse.

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.

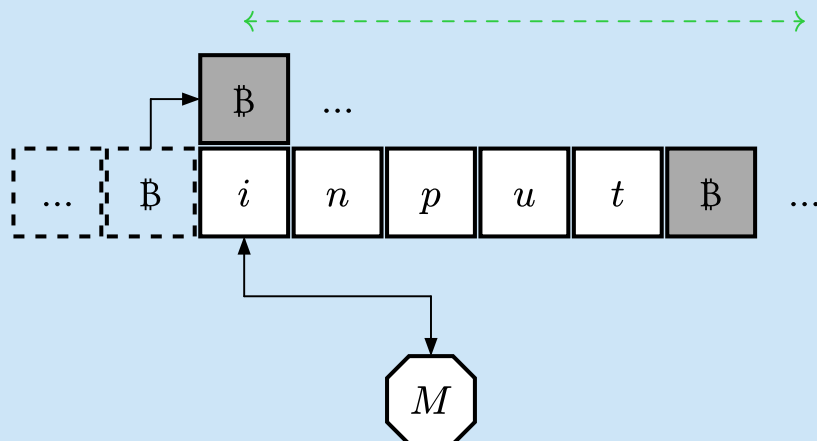
1.2.1. Nastro semi-infinito

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**.



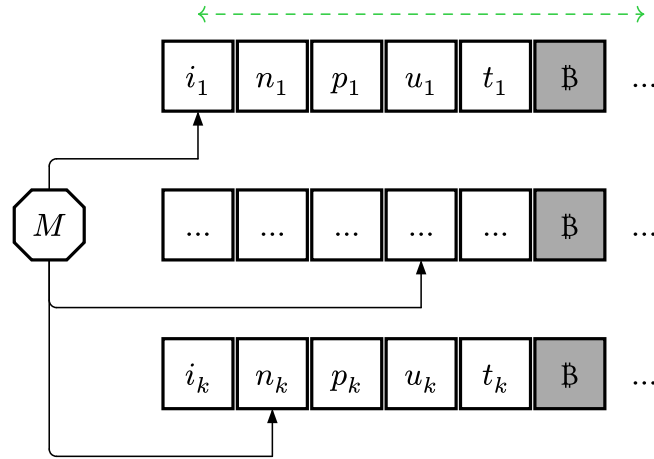
Lemma 1.2.1.1: Una MdT con nastro semi-infinito è equivalente ad una MdT a 1 nastro.

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.

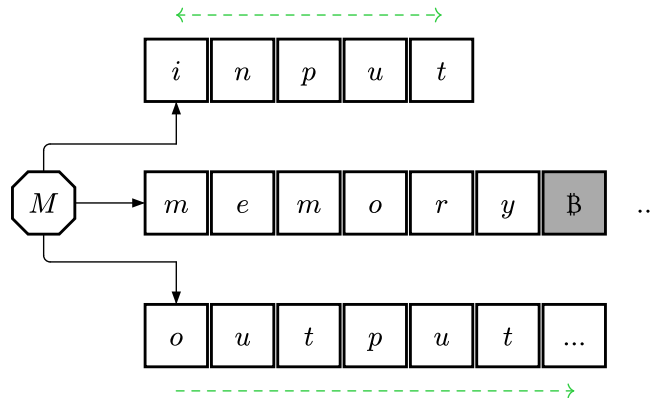
Dato invece una MdT a k nastri, la possiamo simulare con una MdT a 1 nastro rendendo i simboli delle k -tuple, come se avessimo appunto k tracce a disposizione. Per indicare dove sono le testine in ogni momento, marchiamo i caratteri corrispondenti con un puntino.

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

1.2.3. Nastri dedicati

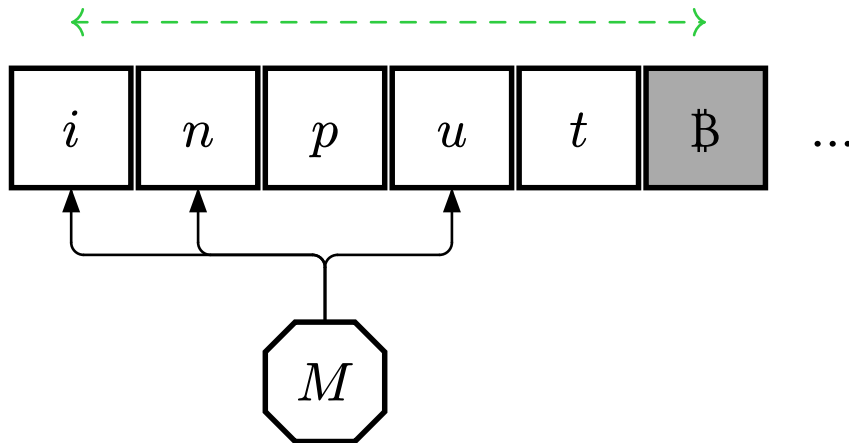
Una versione di MdT a più nastri è quella che **dedica un nastro al solo input**, che quindi ha la testina in sola lettura. Ci sono poi k nastri usati come **memoria**.

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

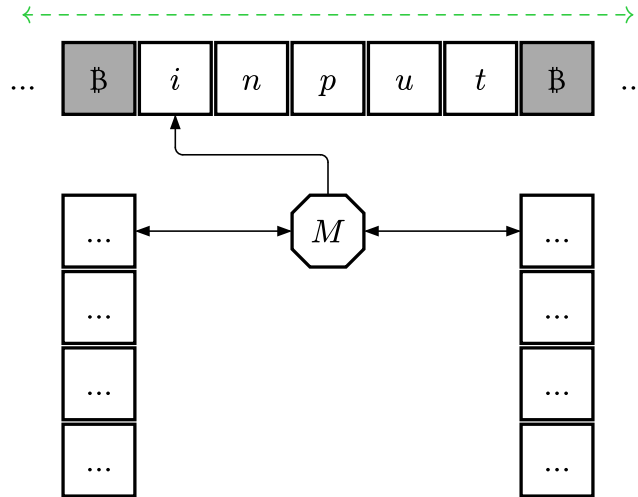
$$L = \{a^n b^n \mid n \geq 0\}.$$

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. Eseguiamo 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**.

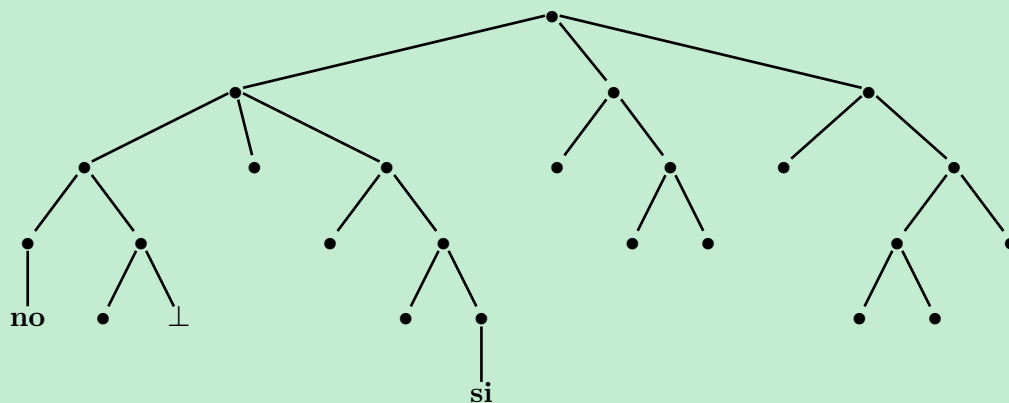
In poche parole, faccio un attraversamento dell'albero di computazione, e ogni volta che trovo un **SI** mi fermo e accetto, altrimenti torno indietro.

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 sfiorare 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 \mathbb{B} , che però non sta in Σ , ovvero

$$\Sigma \subseteq \Gamma \wedge \mathbb{B} \in \Gamma/\Sigma;$$

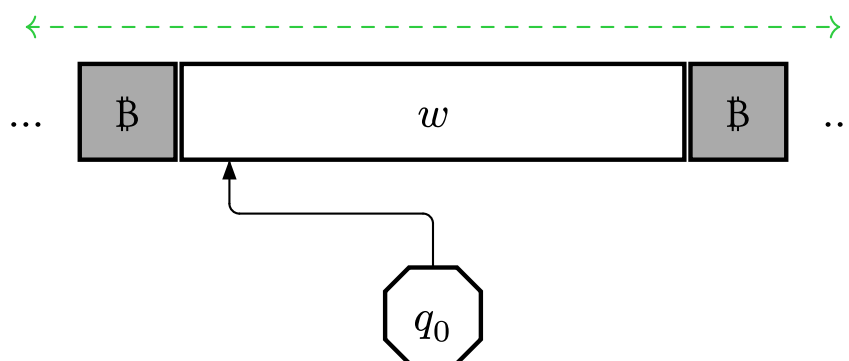
- δ funzione di transizione che permette di processare le stringhe date in input. Essa è la funzione

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;

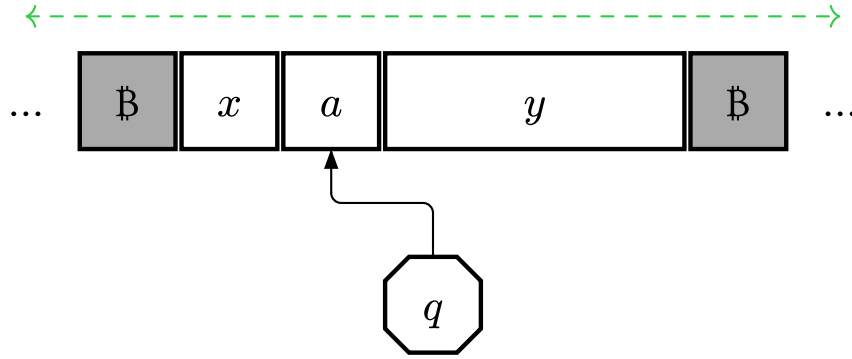
- lo **stato** basta indicarlo e siamo a posto.

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 B di destra.

La **configurazione iniziale** su input w è

$$q_0w.$$

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

$$xqay' \vdash xbp y';$$

- 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

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

2. Problemi di decisione

Abbiamo visto che la computazione di una MdT è un susseguirsi di **configurazioni**, che racchiudono l'input, lo stato corrente della macchina e anche la posizione della testina, implicitamente usando una divisione della stringa di input.

Quando però 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.

2.1. Definizioni preliminari

Quanto detto fin'ora ci servirà per riconoscere dei linguaggi molto particolari che sono **basati sulle configurazioni** di una MdT generica.

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 2.1.1: 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

$$xqay \vdash xbpq$$

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

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

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**

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

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 in maniera **algoritmica** 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**.

Prendiamo ora il **complemento** di questo linguaggio, ovvero

$$(\text{valid}(M))^C = \Delta^* / \text{valid}(M).$$

Una stringa, per appartenere a questo linguaggio, deve far cadere almeno una delle **quattro condizioni**, ovvero:

- ¬a. tra due $\#$ consecutivi o nella prima o nell'ultima configurazione abbiamo 0 o 2 + stati, così da non avere una configurazione. Questo controllo lo possiamo fare con una **FSM** che fa da contatore;
- ¬b. nella configurazione iniziale lo stato deve essere diverso da q_0 , oppure non abbiamo una stringa possibile di input oppure ancora lo stato si trova in posizione diversa, ma comunque utilizziamo un **FSM** di nuovo;
- ¬c. nella configurazione finale lo stato non deve essere finale, quindi ancora una **FSM**;
- ¬d. dobbiamo verificare se

$$\exists i \in \{2, \dots, k\} \mid \neg(\alpha_{i-1} \vdash \alpha_i).$$

Per verificare questa condizione possiamo usare il **non determinismo** e scommettere quale sia la coppia di configurazioni non valide. Qua è facile usare il non determinismo perché abbiamo un \exists , quindi è possibile fare delle scommesse. La macchina risultante legge la configurazione scelta e la carica sulla pila, e poi controlla che quella dopo non sia una configurazioni tra le possibili successive. Ci serve allora un **PDA**.

Abbiamo appena mostrato che

$$(\text{valid}(M))^C \in \text{CFL}.$$

Inoltre, se ci viene data una MdT M abbiamo un **algoritmo** che costruisce un PDA per questo linguaggio, perché basta guardare la descrizione della macchina M per fare tutti i controlli.

2.2. 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 2.2.1 (*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 2.2.2 (*Vuotezza*): Il **problema della vuotezza** ha come input una MdT M e si chiede se

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

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.

2.2.1. Intersezione vuota di DCFL

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

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

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

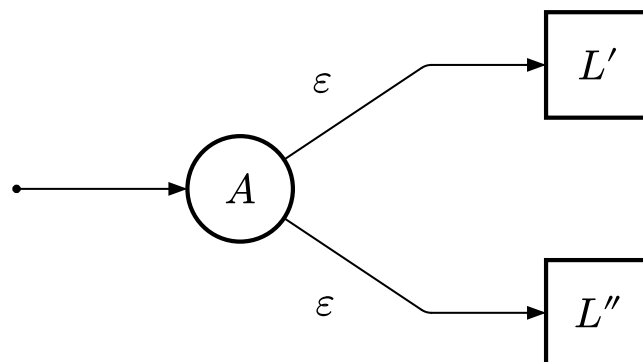
Ma abbiamo detto che questo non lo possiamo decidere in una MdT, quindi non è possibile decidere se l'intersezione di due linguaggi DCFL è vuota.

2.2.2. 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:

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

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.

2.2.3. Universalità

Dato un linguaggio $L \subseteq \Sigma^*$ CFL, ci chiediamo se $L = \Sigma^*$.

Nei linguaggi regolari avevamo risposto positivamente alla domanda, visto che disponevamo della **chiusura rispetto al complemento**. Qua come ci comportiamo?

Per assurdo sia il **problema di universalità** decidibile. Prendiamo come linguaggio L proprio il linguaggio $(\text{valid}(M))^C$. Ma questo vuol dire che

$$L = \Delta^* \iff (\text{valid}(M))^C = \Delta^* \iff \text{valid}(M) = \emptyset \iff L(M) = \emptyset$$

che abbiamo mostrato essere indecidibile, quindi anche l'universalità nei CFL lo è.

2.2.4. Equivalenza

Dati due linguaggi L_1 e L_2 CFL, ci chiediamo se $L_1 = L_2$.

Il problema di equivalenza è un **caso generale** del problema di universalità, perché scegliendo $L_2 = \Delta^*$ ci possiamo ricollegare al problema precedente.

Addirittura non siamo possiamo rispondere all'**equivalenza con un regolare**, visto che Δ^* è un linguaggio regolare.

2.2.5. Contenimento

Dati due linguaggi L_1 e L_2 CFL, ci chiediamo se $L_1 \subseteq L_2$.

Non ho ben capito perché, ma non si può decidere.

2.2.6. Regolarità

Dato L un CFL, ci chiediamo se il linguaggio L è regolare. Questa domanda è «lecita», visto che i regolari sono un sottoinsieme dei CFL. Il problema sembra molto simile al precedente, ma qua la situazione è leggermente diversa da prima perché in quel caso il linguaggio regolare ci veniva dato.

Per risolvere questo problema ci serve l'operazione di **quoziente tra linguaggi**.

Dati due linguaggi L_1 e L_2 definiamo l'operazione

$$L_1/L_2 = \{x \in \Sigma^* \mid \exists y \in L_2 \mid xy \in L_1\}.$$

In poche parole, prendiamo tutte le stringhe nella forma xy di L_1 e andiamo a togliere il suffisso y che però dobbiamo trovare in L_2 .

Esempio 2.2.6.1: Ci vengono dati i seguenti linguaggi.

$$L_1 = a^+bc^+$$

$$L_2 = bc^+$$

$$L_3 = c^+$$

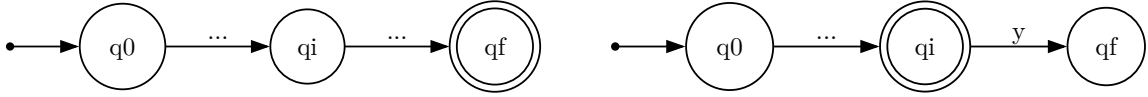
Calcoliamo i possibili quozienti di questi linguaggi.

L_1/L_2	L_1/L_3	L_2/L_3
a^+	a^+bc^*	bc^*

Dato un automa $M = (Q, \Sigma, \delta, q_0, F)$ che riconosce il linguaggio regolare L_1 , possiamo costruire un automa $M' = (Q, \Sigma, \delta, q_0, F')$ tale che

$$F' = \{q \in Q \mid \exists y \in L_2 \mid \delta(q, y) \in F\}.$$

In poche parole, rendiamo finali tutti gli stati dai quali, leggendo una stringa di L_2 , riusciamo a finire in uno stato finale del primo automa.



Questo automa M' che abbiamo appena definito calcola il quoziente, ovvero

$$L(M') = L_1/L_2.$$

Se il linguaggio L_1 è **regolare** anche L_1/L_2 è **regolare** per ogni L_2 possibile. Quindi, dato un linguaggio L_2 a caso, per testare se va bene dobbiamo provare qualche stringa di L_2 .

Torniamo al problema di **regolarità**. Abbiamo in input un CFL e vogliamo sapere se è regolare. Se sapessimo rispondere a questa domanda riusciremmo a risolvere il problema di universalità.

Infatti, sia $G = (V, \Sigma, P, S)$ una grammatica per L . Prendiamo un linguaggio CFL ma non regolare, ovvero $L_0 \in \text{CFL} / \text{Reg}$ definito da una grammatica G_0 .

Costruiamo il linguaggio

$$L_1 = L_0\#\Sigma^* \cup \Sigma^*\#L$$

per il quale possiamo costruire una grammatica G_1 che è CF.

Vediamo due casi:

- se $L = \Sigma^*$ allora il linguaggio L_1 è nella forma

$$L_1 = \Sigma^*\#\Sigma^*$$

perché la seconda parte è uguale in entrambe le parti, mentre la prima, dovendo prendere un'unione, la scegliamo come quella che ha più stringhe, quindi proprio Σ^* ;

- se invece $L \neq \Sigma^*$ allora esiste una stringa che non sta in L , ovvero scegliamo $w \in \Sigma^*/L$. In questo caso definiamo il linguaggio

$$L_2 = L_1 / \#w.$$

Questo linguaggio è esattamente L_0 perché la seconda parte dell'unione non dà contributi. Il linguaggio L_0 non è regolare, per ipotesi, quindi non lo è nemmeno L_1 .

Cosa abbiamo ottenuto: se $L = \Sigma^*$ abbiamo mostrato che L_1 è regolare, mentre se $L \neq \Sigma^*$ abbiamo ottenuto un non regolare, quindi la discriminante per dire se un linguaggio è regolare è dimostrare l'universalità, ovvero

$$L = \Sigma^* \iff L_1 \in \text{Reg}.$$

Abbiamo detto che il primo problema non è decidibile, quindi non lo è nemmeno questo.

2.2.7. Regolarità ma peggio

Se il punto precedente vi è sembrato il male, possiamo andare ancora peggio.

Ci viene dato un linguaggio CFL e ci garantiscono che riconosce un linguaggio regolare. Si può costruire un automa equivalente all'automa a pila che viene fornito con il linguaggio?

La risposta è **NO**: non esiste un algoritmo che, dato un PDA per un linguaggio regolare, è in grado di costruire un automa a stati finiti equivalente. Se esistesse un algoritmo di questo tipo riusciremmo a rendere decidibili i **linguaggi non ricorsivi**, ovvero i linguaggi di tipo 0.

Ci viene data una MdT M per un linguaggio non ricorsivo, ovvero una macchina che risponde **SI**, **NO** oppure va in loop su una stringa data in input.

Ci viene data anche $w \in \Sigma^*$ e una grammatica G_w per il linguaggio

$$(\text{valid}(M, w))^C.$$

Questo linguaggio è esattamente $\text{valid}(M)$ dove però l'input viene fissato a w , ovvero la configurazione iniziale è q_0w . Ovviamente, prendiamo il **complemento** di questo linguaggio.

Se w viene accettata da M allora abbiamo (almeno) una computazione valida, ovvero

$$(\text{valid}(M, w))^C = \Delta^* / \{\text{computazione valida su } w\}.$$

Se invece w non viene accettata da M allora $\text{valid}(M, w)$ è vuoto, ovvero

$$(\text{valid}(M, w))^C = \Delta^*$$

che è un linguaggio regolare, visto che dobbiamo accettare tutto.

Sappiamo che i linguaggi regolari sono **chiusi** rispetto al complemento, quindi il suo complemento è regolare, ovvero

$$(\text{valid}(M, w))^C \in \text{Reg} \iff \text{valid}(M, w) \in \text{Reg}.$$

Nei regolari possiamo decidere l'universalità, ma noi sappiamo che

$$(\text{valid}(M, w))^C = \Delta^* \iff w \text{ accettata da } M$$

ma questo non è possibile perché i linguaggi non ricorsivi sono semi-decidibili.

2.2.8. Regolarità ma ancora peggio

Come **corollario** del punto precedente abbiamo il fatto che, sapendo che un CFL riconosce un regolare e che abbiamo sotto mano l'automa risultante, il suo numero di stati non può essere limitato da alcuna **funzione calcolabile**.

2.3. Riassunto

Problema	Regolari	DCFL	CFL
$L = \emptyset$	✓	✓	✓
L finito/infinito	✓	✓	✓
$L = \Sigma^*$	✓	✓	✗
$L_1 = L_2$	✓ Si usa la riduzione all'automa minimo	✓ Risolto nel 1992	✗ Non funziona anche con L_2 regolare
$L \in \text{Reg}$	✓ Non diciamo che è de- cidibile, è direttamen- te vero	✓ Si ottiene un automa doppiamente espo- nenziale nella descri- zione della pila	✗
$L_1 \cap L_2 = \emptyset$	✓	✗	✗