Teoria dei linguaggi

Indice

1. Introduzione

1.1. Storia

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 risale agli stessi anni con il **compilatore Fortran**, ovvero un traduttore da un linguaggio di alto livello ad uno di basso livello, ovvero il *linguaggio macchina*.

1.2. Ripasso

Un **alfabeto** è un insieme *non vuoto* e *finito* di simboli, di solito indicato con Σ o Γ.

Una **stringa** (o **parola**) è una sequenza *finita* di simboli appartenenti a Σ.

Data una parola w, possiamo definire:
- $|w|$ *numero di caratteri* di w;
- $|w|_a$ *numero di occorrenze* della lettera $a \in \Sigma$ in w.

Una parola molto importante è la **parola vuota** ε, che, come dice il nome, ha simboli, ovvero $|\varepsilon| = 0$.

L'insieme di tutte le possibili parole su Σ è detto Σ^*.

Un'importante operazione sulle parole è la **concatenazione** (o *prodotto*), ovvero se $x, y \in \Sigma^*$ allora la concatenazione w è la parola $w = xy$.

Questo operatore di concatenazione:
- *non è commutativo*, infatti $w_1 = xy \neq yz = w_2$ in generale;
- *è associativo*, infatti $(xy)z = x(yz)$.

La struttura $(\Sigma^*, \cdot, \varepsilon)$ è un **monoide** libero generato da Σ.

Vediamo ora alcune proprietà delle parole:
- **prefisso**: x si dice *prefisso* di w se esiste $y \in \Sigma^*$ tale che $xy = w$;
  - **prefisso proprio** se $y \neq \varepsilon$;
  - **prefisso non banale** se $x \neq \varepsilon$;
  - il numero di prefissi è uguale a $|w| + 1$.
- **suffisso**: y si dice *suffisso* di w se esiste $x \in \Sigma^*$ tale che $xy = w$;
  - **suffisso proprio** se $x \neq \varepsilon$;
  - **suffisso non banale** se $y \neq \varepsilon$;
  - il numero di suffissi è uguale a $|w| + 1$.
- **fattore**: y si dice *fattore* di w se esistono $x, z \in \Sigma^*$ tali che $xyz = w$;
  - il numero di fattori è al massimo $\frac{|w| \cdot |w+1|}{2} + 1$.
- **sottosequenza**: x si dice *sottosequenza* di w se x è ottenuta eliminando 0 o più caratteri da w;
  - un *fattore* è una sottosequenza ordinata.

Un **linguaggio** L definito su un alfabeto Σ è un qualunque sottoinsieme di Σ^*.

2. Gerarchia di Chomsky

2.1. Rappresentazione

Vogliamo rappresentare in maniera finita un oggetto infinito come un linguaggio.

Abbiamo a nostra disposizione due modelli molto potenti:
- **generativo**: date delle regole, si parte da *un certo punto* e si generano tutte le parole di quel linguaggio con le regole date; parleremo di questi modelli tramite le *grammatiche*;
- **riconoscitivo**: si usano dei *modelli di calcolo* che prendono in input una parola e dicono se appartiene o meno al linguaggio.

Considerando il linguaggio sull'alfabeto $\{(,)\}$ delle parole ben bilanciate, proviamo a dare due modelli:
- *generativo*: a partire da una sorgente S devo applicare delle regole per derivate tutte le parole appartenenti a questo linguaggio;
  - la parola vuota ε è ben bilanciata;
  - se x è ben bilanciata, allora anche (x) è ben bilanciata;
  - se x, y sono ben bilanciate, allora anche xy sono ben bilanciate.
- *riconoscitivo*: abbiamo una *black-box* che prende una parola e ci dice se appartiene o meno al linguaggio (in realtà potrebbe non terminare mai la sua esecuzione);
  - $\#(\ = \ \#)$;
  - per ogni prefisso, $\#(\ \geq \ \#)$.

2.2. Grammatiche

Una **grammatica** è una tupla (V, Σ, P, S), con:
- V *insieme finito e non vuoto* delle **variabili**; queste ultime sono anche dette *simboli non terminali* e sono usate durante il processo di generazione delle parole del linguaggio;
- Σ *insieme finito e non vuoto* dei **simboli terminali**; questi ultimi appaiono nelle parole generate, a differenza delle variabili che invece non possono essere presenti;
- P *insieme finito* delle **regole di produzione**;
- $S \in V$ **simbolo iniziale** o **assioma**, è il punto di partenza della generazione.

2.2.1. Regole di produzione

Soffermiamoci sulle regole di produzione: la forma di queste ultime è $\alpha \longrightarrow \beta$, con $\alpha \in (V \cup \Sigma)^+$ e $\beta \in (V \cup \Sigma)^*$.

Una regola di produzione viene letta come "$se\ ho\ \alpha\ allora\ posso\ sostituirlo\ con\ \beta$".

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.

Diciamo che x deriva y in un passo, con $x, y \in (V \cup \Sigma)^*$, se e solo se $\exists(\alpha \longrightarrow \beta) \in P$ e $\exists \eta, \delta \in (V \cup \Sigma)^*$ tali che $x = \eta\alpha\delta$ e $y = \eta\beta\delta$.

Il passo di derivazione lo indichiamo con $x \Rightarrow y$.

La versione estesa afferma che x deriva y in $k \geq 0$ passi, e lo indichiamo con $x \overset{k}{\Rightarrow} y$, se e solo se $\exists x_0, ..., x_k \in (V \cup \Sigma)^*$ tali che $x = x_0$, $x_k = y$ e $x_{i-1} \Rightarrow x_i \ \forall i \in [1, k]$.

Se non ho indicazioni sul numero di passi k posso scrivere:
- $x \overset{*}{\Rightarrow} y$ per indicare un numero generico di passi, e questo vale se e solo se $\exists k \geq 0$ tale che $x \overset{k}{\Rightarrow} y$;
- $x \overset{+}{\Rightarrow} y$ per indicare che serve almeno un passo, e questo vale se e solo se $\exists k > 0$ tale che $x \overset{k}{\Rightarrow} y$.

2.2.2. Linguaggio generato da una grammatica

Indichiamo con $L(G)$ il linguaggio generato dalla grammatica G, ed è l'insieme $\left\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\right\}$.

Due grammatiche G_1, G_2 sono **equivalenti** se e solo se $L(G_1) = L(G_2)$.

Se consideriamo l'esempio delle parentesi ben bilanciate, possiamo definire una grammatica per questo linguaggio con le seguenti regole di produzione:
- $S \longrightarrow \varepsilon$;
- $S \longrightarrow (S)$;
- $S \longrightarrow SS$.

Vediamo un esempio più complesso. Siano:
- $\Sigma = \{a, b, c\}$;
- $V = \{S, B\}$;
- $P = \{S \longrightarrow aBSc \mid abc, Ba \longrightarrow aB, Bb \longrightarrow bb\}$.

Questa grammatica genera il linguaggio $L(G) = \{a^n b^n c^n \mid n \geq 1\}$: infatti, il "caso base" genera la stringa *abc*, mentre le iterazioni "maggiori" generano il numero di *a* e *c* corretti, con i primi che vengono ordinati prima di inserire anche il numero corretto di *b*.

2.3. Gerarchia

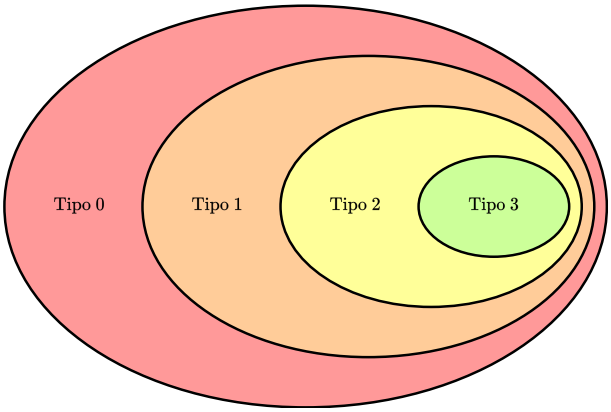
Negli anni 50 Noam Chomsky studia la generazione dei linguaggi formali e crea una **gerarchia di grammatiche formali**. La classificazione delle grammatiche viene fatta in base alle regole di produzione che definiscono la grammatica.

Grammatica	Regole	Modello riconoscitivo
Tipo 0.	Nessuna restrizione, sono il tipo più generale.	*Macchine di Turing.*
Tipo 1, dette **context-sensitive** o *dipendenti dal contesto.*	Se $(\alpha \longrightarrow \beta) \in P$ allora $\lvert\beta\rvert \geq \lvert\alpha\rvert$, ovvero devo generare parole che non siano più corte di quella di partenza. Sono dette *dipendenti dal contesto* perché ogni regola $(\alpha \longrightarrow \beta) \in P$ può essere riscritta come $\alpha_1 A \alpha_2 \longrightarrow \alpha_1 B \alpha_2$, con $\alpha_1, \alpha_2 \in (V \cup \Sigma)^*$ che rappresentano il *contesto*, $A \in V$ e $B \in (V \cup \Sigma)^+$.	*Automi limitati linearmente.*
Tipo 2, dette **context-free** o *libere dal contesto.*	Le regole in P sono del tipo $\alpha \longrightarrow \beta$, 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 $A \longrightarrow aB$ oppure $A \longrightarrow a$, con $A, B \in V$ e $a \in \Sigma$. Vale anche il simmetrico.	*Automi a stati finiti.*

Nella figura successiva vediamo una rappresentazione grafica della gerarchia di Chomsky: notiamo come sia una gerarchia propria, ovvero

$$L_3 \subset L_2 \subset L_1 \subset L_0,$$

ma questa gerarchia non esaurisce comunque tutti i linguaggi possibili. Esistono infatti linguaggi che non sono descrivibili in maniera finita con le grammatiche.



Sia $L \subseteq \Sigma^*$, allora L è di tipo i, con $i \in [0,3]$, se e solo se esiste una grammatica G di tipo i tale che $L = L(G)$, ovvero posso generare L a partire dalla grammatica di tipo i.

2.4. Potenza computazionale

Se una grammatica é 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.

Teorema 2.4.1 Una grammatica di tipo 1 è **decidibile**.

Dimostrazione
Siano G una grammatica di tipo 1 e $w \in \Sigma^*$, ci chiediamo se $w \in L(G)$.
Sia $h = |w|$, ma allora essendo G di tipo 1 ogni forma sentenziale che compare in P non deve superare la lunghezza h, altrimenti potremmo ridurre il numero di caratteri presenti nella forma sentenziale e andare contro la definizione di grammatica di tipo 1.

Sia $T_i = \left\{\gamma \in (V \cup \Sigma)^{\leq n} \mid S \overset{\leq i}{\Rightarrow} \gamma\right\}$ l'insieme di tutte le parole generate dalla grammatica G che hanno al massimo n caratteri e sono generate in massimo i passi di derivazione.
Data questa definizione di T_i possiamo affermare che:
- $T_0 = \{S\}$;
- $T_i = T_{i-1} \cup \left\{\gamma \in (V \cup \Sigma)^{\leq n} \mid \exists \beta \in T_{i-1} \text{ tale che } \beta \Rightarrow \gamma\right\}$.

Per come sono costruiti gli insiemi T_i possiamo affermare che

$$T_0 \subseteq T_1 \subseteq \ldots \subseteq (V \cup \Sigma)^{\leq n},$$

ma quest'ultimo insieme è un insieme *finito*.
Prima o poi non si potranno più generare delle stringhe, ovvero $\exists k$ tale che $T_{k-1} = T_k$.
Una volta individuato questo valore k basta controllare se $w \in T_k$. □

Questo non vale invece per le grammatiche di tipo 0: infatti, queste sono dette **semidecidibili**, in quanto un sistema riconoscitivo potrebbe non terminare mai l'algoritmo di riconoscimento e finire quindi in un loop infinito.

Teorema 2.4.2 Una grammatica di tipo 0 è **semidecidibile**.

Dimostrazione
Siano G una grammatica di tipo 0 e $w \in \Sigma^*$, ci chiediamo se $w \in L(G)$.
Non essendo G di tipo 1 non abbiamo il vincolo $|\beta| \geq |\alpha|$ nelle regole di produzione.

Sia $U_i = \left\{ \gamma \in (V \cup \Sigma)^* \mid S \overset{\leq i}{\Rightarrow} \gamma \right\}$ l'insieme di tutte le parole generate dalla grammatica G in massimo i passi di derivazione.
Data questa definizione di U_i possiamo affermare che:
- $U_0 = \{S\}$;
- $U_i = U_{i-1} \cup \{\gamma \in (V \cup \Sigma)^* \mid \exists \beta \in U_{i-1} \text{ tale che } \beta \Rightarrow \gamma\}$.

Per come sono costruiti gli insiemi U_i possiamo affermare che

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

ma quest'ultimo insieme è un insieme *infinito*.
Vista questa caratteristica, nessuno garantisce l'esistenza di un k tale che $U_{k-1} = U_k$ e quindi non si ha la certezza di terminare l'algoritmo di riconoscimento. □

Le grammatiche di tipo 0 generano i **linguaggi ricorsivamente enumerabili**: per stabilire se $w \in L(G)$ devo *elencare* con un programma tutte le stringhe del linguaggio e controllare se w compare in esse.

Questa operazione di elencazione in poche parole è la generazione degli insiemi U_i, che poi vengono ispezionati per vedere se la parole w è presente o meno.

2.5. Estensione con la parola vuota

La parola vuota é molto "noiosa" perché la sua presenza in una regola di derivazione del tipo $\alpha \longrightarrow \varepsilon$ esclude la grammatica dai tipi superiori allo 0.

Inserire la parola vuota nel linguaggio generato da una grammatica non é un'operazione critica: infatti, essa non modifica la cardinalità del linguaggio. Voglio quindi costruire l'insieme $L' = L(G) \cup \{\varepsilon\}$. Per far ciò dobbiamo aggiungere la regola di produzione $S \longrightarrow \varepsilon$, ma dobbiamo garantire di più: infatti, se in P é presente anche una regola del tipo $A \longrightarrow \alpha_1 S \alpha_2$ riesco a generare più parole di quelle che riesco effettivamente a generare senza la parola vuota.

In poche parole, *la variabile che genera la parola vuota non deve essere presente nella parte destra delle regole di produzione.*

La nuova grammatica $G' = (V', \Sigma, P', S')$ é formata da:
- V' insieme delle variabili, definito come $S \cup \{S'\}$;
- Σ insieme dei simboli terminali;
- P' insieme delle regole di produzione;
- $S' \in V'$ assioma.

Vengono aggiunte due regole di produzione:
- $S' \longrightarrow \varepsilon$ per generare la parola vuota;
- $S' \longrightarrow S$ per collegare il comportamento di G' a G.

Con queste due nuove regole riesco a generare il linguaggio $L(G)$, grazie alla regola $S' \longrightarrow S$, unito alla parola vuota, grazie alla regola $S' \longrightarrow \varepsilon$.

Questo tipo di costruzione vale per tutte le grammatiche di tipo 1. Per le grammatiche di tipo 2 é più facile: basta rilassare il vincolo $\beta \in (V \cup \Sigma)^+$ in $\beta \in (V \cup \Sigma)^*$. Infine, per le grammatiche di tipo 3 basta aggiungere la regola di produzione $A \longrightarrow \varepsilon$.

Le regole di produzione nella forma $\alpha \longrightarrow \varepsilon$ sono dette **ε-produzioni**.

3. Linguaggi di tipo 3

3.1. Automi a stati finiti (deterministici)

Gli **automi a stati finiti** sono un modello riconoscitivo usato per caratterizzare i *linguaggi regolari*.

3.1.1. Definizione informale

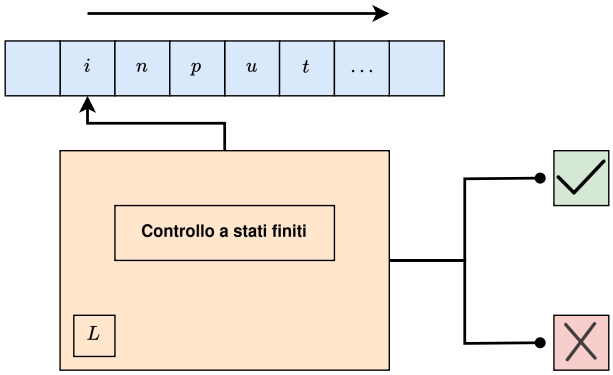
Gli automi a stati finiti sono delle macchine molto semplici: hanno un **controllo a stati finiti** che legge l'input da un **nastro** *read-only*, formato da una serie di celle, ognuna delle quali contiene un carattere dell'input.

Per leggere l'input si utilizza una **testina di lettura**, posizionata inizialmente sulla cella più a sinistra (ovvero sul primo carattere di input), e poi spostata iterazione dopo iterazione da sinistra verso destra. Gli automi che studieremo per ora sono **one-way**, ovvero la lettura avviene *solo* da sinistra verso destra.

Il controllo a stati finiti, prima della lettura dell'input, è allo **stato iniziale**.

Ad ogni iterazione, a partire dallo stato corrente e dal carattere letto dal nastro, ci si muove con la testina a destra sul simbolo successivo e si cambia stato.

Quando si arriva alla fine del nastro, in base allo stato corrente dell'automa, quest'ultimo risponde *"si"*, ovvero la parola in input appartiene al linguaggio, oppure *"no"*, ovvero la parola in input non appartiene al linguaggio.



3.1.2. Definizione formale

Un automa è una tupla $\{Q, \Sigma, \delta, q_0, F\}$, con:
- Q insieme *finito e non vuoto* degli stati;
- Σ insieme *finito e non vuoto* dell'alfabeto di input;
- $\delta : Q \times \Sigma \longrightarrow Q$ funzione di transizione, il programma della macchina;
- $q_0 \in Q$ stato iniziale;
- $F \subseteq Q$ insieme *finito e non vuoto* degli stati finali.

La parte dinamica dell'automa è la **funzione di transizione** che, dati lo stato iniziale e un simbolo del linguaggio, calcola lo stato successivo.

Possiamo estendere la funzione di transizione affinché utilizzi una parola del linguaggio. Per induzione sulla lunghezza delle parole definiamo $\delta^* : Q \times \Sigma^* \longrightarrow Q$ la funzione tale che:
- $\delta^*(q, \varepsilon) = q \quad \forall q \in Q$;
- $\delta^*(q, xa) = \delta(\delta^*(q, x), a) \quad \forall q \in Q, x \in \Sigma^*, a \in \Sigma$.

Per semplicità useremo δ al posto di δ^* perché sui singoli caratteri δ e δ^* hanno lo stesso comportamento.

3.1.3. Linguaggio

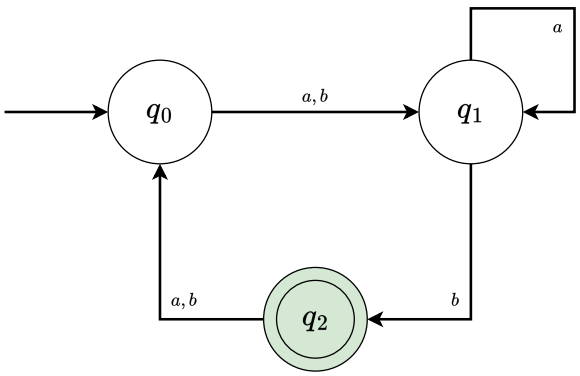
Chiamiamo $L(A) = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}$ il **linguaggio riconosciuto dall'automa**, ovvero l'insieme delle parole che applicate alla funzione di transizione, a partire dallo stato iniziale, mi mandano in uno stato finale.

3.1.4. Rappresentazione grafica

Possiamo vedere un automa come un grafo, dove:

- i vertici sono gli **stati**;
- gli archi sono le **transizioni**; gli archi sono orientati ed etichettati con la lettera dell'alfabeto che causa quella transizione.

Lo **stato iniziale** è indicato con una freccia entrante nello stato, mentre gli **stati finali** sono nodi doppiamente cerchiati.



3.2. Automi a stati finiti non deterministici

3.2.1. Definizione informale

Gli **automi a stati finiti non deterministici** (*NFA*) sono automi che hanno *almeno* uno stato dal quale escono 2 o più archi con la stessa lettera. Negli automi **deterministici** (*DFA*), invece, da *ogni* stato esce al più un arco con la stessa lettera.

La differenza principale sta nella complessità computazionale: se negli automi deterministici devo controllare se la parola ci porta in uno stato finale, negli automi non deterministici devo controllare se tra *tutti* i possibili cammini dell'**albero di computazione** ne esiste uno che ci porta in uno stato finale.

Possiamo vedere il non determinismo come *una scommessa che va sempre a buon fine.*

3.2.2. Definizione formale

Un automa non deterministico differisce da un automa deterministico solo per la funzione di transizione: infatti, quest'ultima diventa $\delta : Q \times \Sigma \longrightarrow 2^Q$. Il valore ritornato é un elemento dell'**insieme delle parti** di Q, cioè un sottoinsieme di stati nei quali possiamo finire applicando un carattere di Σ allo stato corrente.

Come prima, definiamo l'estensione della funzione di transizione per induzione sulla lunghezza delle parole come la funzione $\overline{\delta} : Q \times \Sigma^* \longrightarrow 2^Q$ tale che:

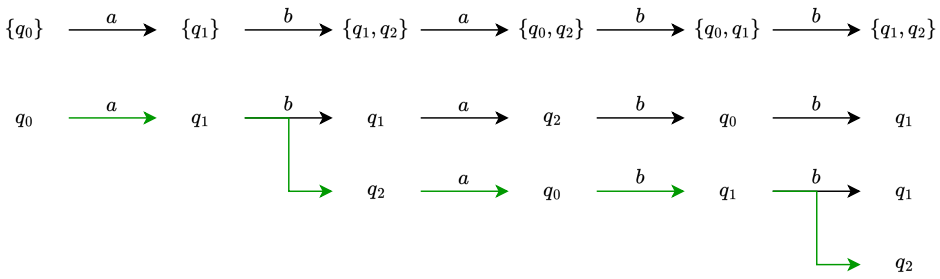
- $\delta^*(q, \varepsilon) = \{q\}$;
- $\delta^*(q, xa) = \bigcup_{p \in \delta^*(q,x)} \delta(p, a)$.

Il linguaggio riconosciuto dall'automa diventa $L(A) = \left\{ w \in \Sigma^* \mid \overline{\delta}(q_0, w) \bigcap F \neq \varnothing \right\}$.

3.2.3. Albero di computazione

L'**albero di computazione** é una rappresentazione grafica di tutti i cammini percorsi dall'automa non deterministico quando deve dire se una parola appartiene o meno al linguaggio. Il singolo cammino é detto **computazione**.

Prendiamo l'automa nella pagina precedente e aggiungiamo un cappio in q_1 causato dalla lettura del carattere b. Ci chiediamo se la parola $w = ababb$ viene riconosciuta o meno dall'automa.



Nella parte superiore vediamo i passi intermedi della *funzione di transizione*: all'inizio é un insieme che contiene solo lo stato iniziale, poi mano a mano l'insieme viene modificato con gli insiemi nei quali si é "allo stesso momento".

Nella parte inferiore vediamo invece l'*albero di computazione*.

3.3. Numero di stati

La domanda sorge spontanea: *quale é il minimo numero di stati necessari affinché un DFA per un dato linguaggio L riconosca una parola?*

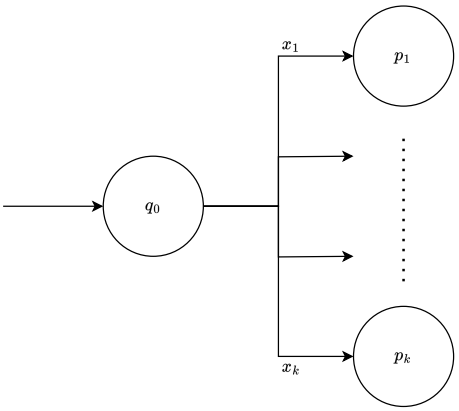
3.3.1. Distinguibilità

Dato un linguaggio $L \subseteq \Sigma^*$, due parole $x, y \in \Sigma^*$ sono **distinguibili** rispetto ad L se $\exists z \in \Sigma^*$ tale che $(xz \in L \land yz \notin L) \lor (xz \notin L \land yz \in L)$.

Teorema 3.3.1.1 Siano $L \subseteq \Sigma^*$ un linguaggio e $X \subseteq \Sigma^*$ tale che ogni coppia di parole in X é distinguibile rispetto ad L, allora ogni DFA per L deve avere almeno $|X|$ stati.

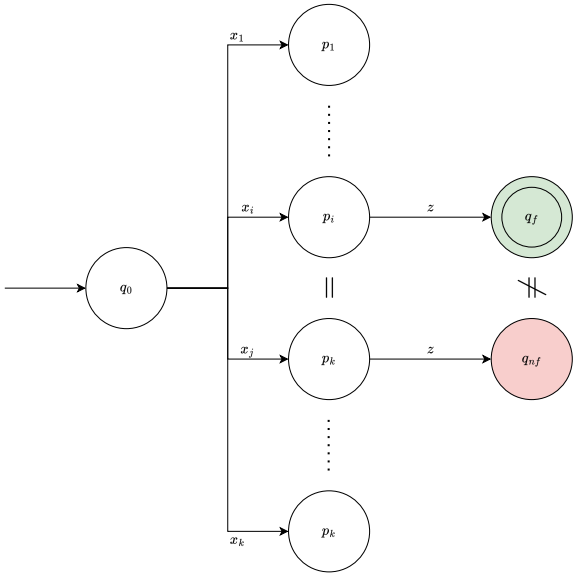
Dimostrazione

Supponiamo che l'insieme X sia $X = \{x_1, ..., x_k\}$ di cardinalità k. Definiamo l'automa $A = (Q, \Sigma, \delta, q_0, F)$ tale che $\delta(q_0, x_i) = p_i \quad \forall i \in [1, k]$.



Per assurdo sia $|Q| < k$, ma allora $\exists i, j \in [1, k]$ tali che $i \neq j$ e $\delta(q_0, x_i) = p_i = p_j = \delta(q_0, x_j)$. Questo vale perché avendo meno stati del numero di elementi da "mappare" almeno due elementi finiscono nello stesso stato.

Ma x_i e x_j sono due parole distinguibili: allora $\exists z \in \Sigma^*$ tale che $x_i z \in L \wedge x_j z \notin L$ (o viceversa).



Ma questo é assurdo: infatti, x_i e x_j sono due parole distinguibili che però finiscono in uno stato che deve essere sia finale che non finale.

Allora $|Q| \geq k$. □

La costruzione dell'insieme X é molto utile per dimostrare che alcuni linguaggi non possono essere di tipo 3: infatti, se si riesce a costruire un insieme X tale che $|X| = +\infty$ allora si dimostra che un dato linguaggio non é riconoscibile da un DFA.

Vediamo un esempio: sia $L_n = \{x \in \{a, b\}^* \mid \text{il simbolo di } x \text{ in posizione } n \text{ da destra é una } a\}$. Il più semplice DFA utilizza 2^n stati, ovvero tutte le possibili finestre di lunghezza n, mentre il più semplice NFA utilizza $n + 1$ stati.

Come costruiamo un insieme X la cui cardinalità ci faccia da lower-bound?

Sia $X = \{a, b\}^n = \left\{w \in \{a, b\}^+ \mid |w| = n\right\}$ e siano $x, y \in X$ tali che $x \neq y$, ma allora $\exists i \in [1, n]$ tale che $x_i \neq y_i$, ovvero

$$x = x_1 ... x_i ... x_n$$
$$\neq$$
$$y = y_1 ... y_i ... y_n.$$

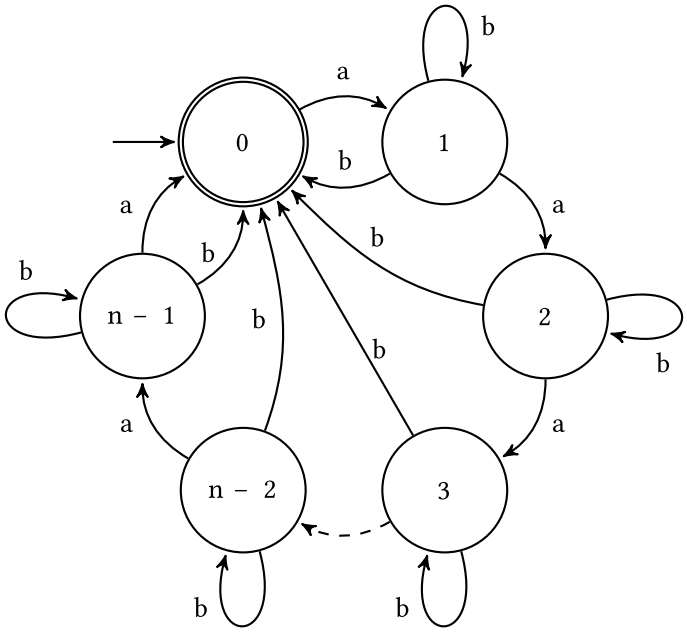
Per semplicità sia $x_i = a$ e $y_i = b$. Aggiungiamo ora $i - 1$ caratteri in coda alle parole, che per semplicità saranno delle a.

$$x = x_1 ... a ... x_n a^{i-1}$$
$$y = y_1 ... b ... y_n a^{i-1}.$$

I caratteri dopo la posizione i diventano $(n - i) + (i - 1) = n - 1$, spostando i caratteri x_i e y_i nella posizione n-esima da destra, ma per come li avevamo definiti essi sono diversi, quindi le due parole sono distinguibili.

3.3.2. Da DFA a NFA

L'**automa di Meyer e Fischer** é un NFA molto importante ideato nel 1971 per dare un lower-bound al numero di stati necessari per rappresentare un DFA a partire da un NFA di n stati.



L'automa viene chiamato M_n ed é formato da:
- $Q = \{0, ..., n - 1\}$;
- $\Sigma = \{a, b\}$;
- $q_0 = 0$;
- $F = \{0\}$.

La funzione di transizione é definita come

$$\delta(i, a) = \{i + 1\}$$
$$\delta(i, b) = \begin{cases} \varnothing & \text{se } i \neq 0 \\ \{i, 0\} & \text{se } 1 \leq i < n \end{cases}.$$

Prima della dimostrazione effettiva vediamo un paio di proprietà molto importanti di questo automa.

Sia $S \subseteq \{0, ..., n-1\}$, definiamo la parola

$$w_s = \begin{cases} b & \text{se } S = \varnothing \\ a^i & \text{se } S = \{i\} \\ a^{e_k - e_{k-1}} b a^{e_{k-1} - e_{k-2}} b ... b a^{e_2 - e_1} b a^{e_1} & \text{se } S = \{e_k, ..., e_1\} \text{ insieme ordinato e } k \geq 2 \end{cases}.$$

Proposizione 3.3.2.2 $\forall S \subseteq \{0, ..., n-1\} \quad \delta(0, w_s) = S.$

Proposizione 3.3.2.3 Dati $S, T \subseteq \{0, ..., n-1\}$, se $S \neq T$ allora w_s e w_t sono distinguibili.

Dimostrazione
Sia $x \in S \setminus T$ numero intero che appartiene a S ma non a T.
Sappiamo che $\delta(0, w_s) = S$ per la proprietà precedente e che $x \in S$, ma allora

$$0 \overset{w_s}{\rightsquigarrow} x \overset{a^{n-x}}{\rightsquigarrow} 0.$$

Sappiamo inoltre che $\delta(0, w_t) = T$ per la proprietà precedente e che $x \notin T$. Sia $y \in T$, ma allora

$$0 \overset{w_t}{\rightsquigarrow} y \overset{a^{n-x}}{\rightsquigarrow} U.$$

L'insieme U non contiene 0 perché l'unico modo per finire in 0 é aggiungere un numero di a uguale a $n - y$, ma sappiamo che $n - x \neq n - y$ per come sono stati definiti x e y.
Ma allora la stringa a^{n-x} distingue w_s e w_t. □

Ora possiamo dimostrare il seguente teorema.

Teorema 3.3.2.4 Sia A un NFA per L di n stati, allora A' DFA per L ha almeno 2^n stati.

Dimostrazione
Sia $S \subseteq \{0, ..., n-1\}$, l'insieme degli stati di A. Definisco l'insieme

$$X = \{w_s \mid S \subseteq \{0, ..., n-1\}\}.$$

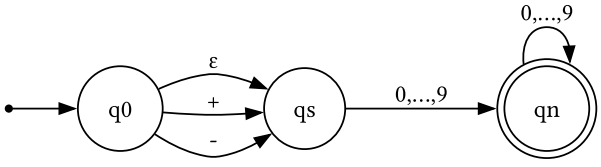
Questo insieme ha cardinalità 2^n: infatti, é formato da tutte le stringhe w_s generate da S sottoinsieme di $\{0, ..., n-1\}$ di n elementi, ma tutti i possibili sottoinsiemi di un insieme di n elementi sono 2^n.
Per la seconda proprietà le stringhe $w_s \in X$ sono tutte distinguibili, ma allora per il teorema precedente A' deve avere almeno 2^n stati. □

3.4. Estensioni

3.4.1. ε-mosse

Le **ε-mosse** sono una possibile estensione degli automi a stati finiti che permettono transizioni sulla parola vuota, ovvero permettono di spostarsi da uno stato all'altro senza leggere un carattere dal nastro.

Sono utili nei compilatori quando é possibile definire gli interi positivi senza il segno *più*.



Questa estensione non aumenta però la potenza espressiva dell'automa: infatti, ogni sequenza del tipo

$$p \overset{\varepsilon}{\rightsquigarrow} p' \overset{a}{\longrightarrow} q' \overset{\varepsilon}{\rightsquigarrow} q$$

può essere tradotta nella transizione $p \overset{a}{\longrightarrow} q$.

3.4.2. Stati iniziali multipli

L'ultima estensione che vediamo é quella degli **stati iniziali multipli**: al posto di avere un singolo stato iniziale abbiamo un insieme di stati dai quali poter iniziare.

Anche questa estensione non aumenta la potenza espressiva dell'automa: infatti, la funzione di transizione partirà direttamente con un insieme di stati e non da un insieme con un solo stato.

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

Dimostriamo che dato l'automa A per L possiamo costruire una costruire una grammatica G di tipo 3 tale che $L(A) = L(G)$.

Teorema 3.5.1 Le grammatiche di tipo 3 sono equivalenti agli automi a stati finiti.

Dimostrazione
$[A \Longrightarrow G]$ Dato $A = (Q, \Sigma, \delta, q_0, F)$ DFA per L costruisco la grammatica $G = (V, \Sigma, P, S)$ tale che:
- $V = Q$;
- Σ rimane uguale;
- $S = q_0$.

Le regole di produzione in P sono nella forma:
- $A \longrightarrow aB$ se $\delta(A, a) = B$, con $A, B \in Q$ e $a \in \Sigma$;
- $A \longrightarrow a$ se $A \in F$, con $A \in Q$ e $a \in \Sigma$.

Si può dimostrare che

$$q_0 \Rightarrow xA \iff \delta(q_0, x) = A.$$

$[G \Longrightarrow A]$ Data $G = (V, \Sigma, P, S)$ grammatica di tipo 3 costruisco un DFA $A = (Q, \Sigma, \delta, q_0, F)$ tale che:
- $Q = V \cup \{q_f\}$;
- Σ rimane uguale;
- $q_0 = S$;
- $F = \{q_f\}$.

La funzione di transizione δ é definita nel seguente modo:
- se $A \longrightarrow aB$ allora $B \in \delta(A, a)$;
- se $A \longrightarrow a$ allora $q_f \in \delta(A, a)$.

□

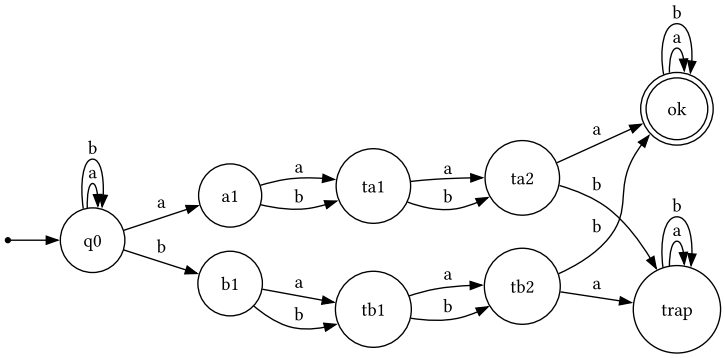
3.6. Esempi

Vediamo qualche esempio notevole.

3.6.1. L_n

Sia $L_n = \{x \in \{a, b\}^+ \mid x \text{ contiene due simboli uguali a distanza } n\}$.

Con $n = 3$, la stringa *ababba* appartiene a L_n. Costruiamo un NFA per questo linguaggio con $n = 3$.



Questo NFA ha $2n + 2$ stati, un DFA quanti stati avrebbe? Cerchiamo di costruire un insieme X di stringhe distinguibili per dare un lower bound al numero di stati.

Sia $X = \{a, b\}^n$ e siano $x, y \in X$ tali che $x \neq y$, ma allora $\exists i \in [1, n]$ tale che $x_i \neq y_i$ rappresenta la prima cifra diversa delle due parole:

$$x = x_1 ... x_i ... x_n$$
$$\neq$$
$$y = y_1 ... y_i ... y_n.$$

Le prime $i - 1$ cifre sono tutte uguali, allora se prendo la stringa $z = \overline{x_1} ... \overline{x_{i-1}} a$ ogni stringa in X diventa distinguibile: infatti, prendendo il complemento di ogni cifra evitiamo che le prime $i - 1$ cifre *"matchino"* quelle inserite. Inserendo una a in fondo a z andiamo ad accettare solo x o solo y, in base a quale parole ha la cifra i-esima uguale ad a.

La cardinalità di questo insieme é 2^n, quindi ogni DFA per L ha almeno 2^n stati.

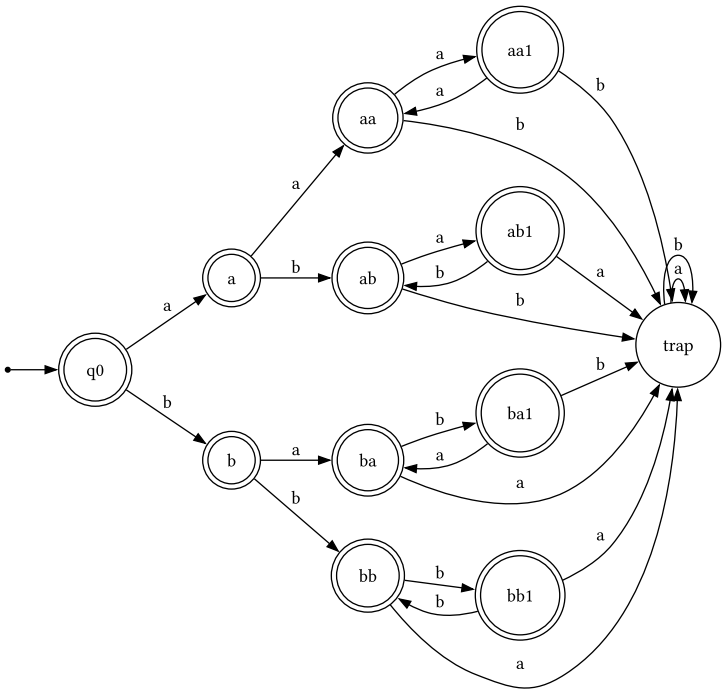
3.6.2. $L_n{}'$

Sia $L_n{}' = \left\{ x \in \{a, b\}^+ \mid \text{ogni coppia di simboli a distanza } n \text{ contiene due simboli uguali} \right\}$.

Notiamo subito come valga la relazione $L_n{}' \subset L_n$.

Con $n = 3$, la stringa *abbabba* appartiene a $L_n{}'$. In poche parole, $x \in L$ se e solo se $\exists w \in \{a, b\}^n$, $\exists y$ prefisso di w e $\exists k \geq 0$ tali che $x = w^k y$.

Costruiamo un DFA per questo linguaggio con $n = 2$, visto che con $n = 3$ il numero di stati esplode, ma di questo ne parleremo dopo.



Il numero di stati di questo DFA é $2^{n+1} - 1 + 2^{n+1}$.

Riusciamo a fare meglio con un NFA? La risposta é no: gli NFA *"lavorano male"* quando si parla di *"ogni"*, mentre *"lavorano bene"* quando si parla di *"esiste"*.

Questo perché se si parla di *"esiste"* dobbiamo fare una singola scommessa, mentre se si parla di *"ogni"* dobbiamo fare molte più scommesse.

3.7. Fooling set

Cerchiamo di fare, come per i DFA con l'insieme X di stringhe distinguibili, un lower bound al numero di stati di un NFA.

3.7.1. Definizione

Diamo la definizione di **fooling set**, o *insieme di ingannatori/imbroglioni*.

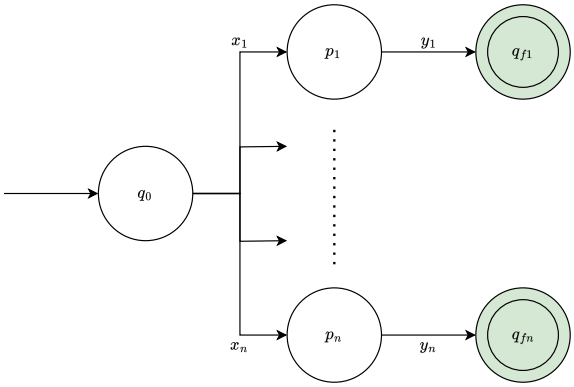
Sia $L \subseteq \Sigma^*$, l'insieme di coppie $P \subseteq \Sigma^* \times \Sigma^* = \{(x_i, y_i) \mid x_i, y_i \in \Sigma^* \wedge i \in [1, n]\}$ é un fooling set per L se e solo se:
1. $x_i y_i \in L \quad \forall i \in [1, n]$;
2. $x_i y_j \notin L \quad \forall i.j \in [1, n] \wedge i \neq j$.

Esiste una versione rilassata del fooling set, detta **extended fooling set**, che mantiene la proprietà 1 ma sostituisce la proprietà 2 con:
2. $x_i y_j \notin L \vee x_j y_i \notin L \quad \forall i.j \in [1, n] \wedge i \neq j$.

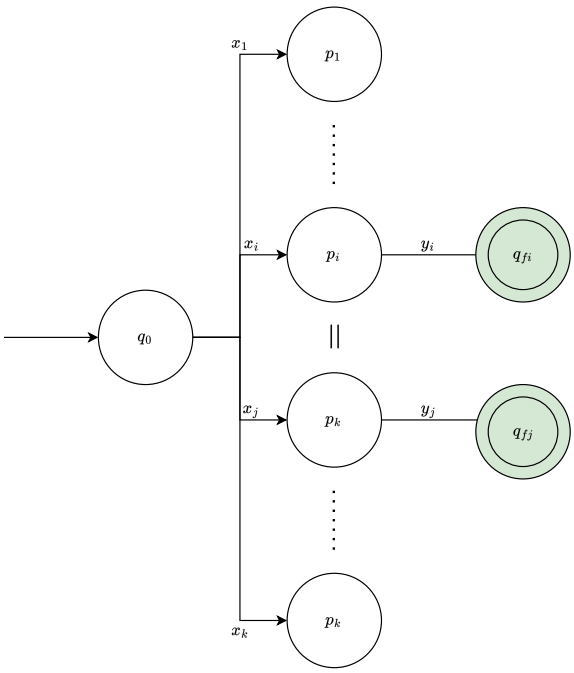
In questo caso, prese due coppie, basta che *almeno un incrocio* non appartenga a L, mentre nel primo caso *entrambi gli incroci* non appartengono a L.

Teorema 3.7.1.1 Siano $L \subseteq \Sigma^*$ e P extended fooling set per L, allora ogni NFA per L ha almeno $|P|$ stati.

Dimostrazione
Sia $A = (Q, \Sigma, \delta, q_0, F)$ NFA per L, consideriamo le parole $x_i y_i$ formate dalla concatenazione dei valori contenuti nelle coppie di P e consideriamo i cammini accettanti di queste parole.



Per assurdo sia $|Q| < n$, ma allora $\exists i, j \in [1, n]$ tali che $p_i = p_j$.



Ma questo é assurdo: infatti, a partire dallo stesso stato $p_i = p_j$ finiamo in due stati entrambi finali, ma essendo P un extended fooling set per la proprietà 2 almeno una tra le parole $x_i y_j$ e $x_j y_i$ non deve essere accettata.

Allora abbiamo dimostrato che $|Q| \geq n$. □

3.7.2. Applicazione a $L_n{}'$

Cerchiamo di definire un extended fooling set per l'insieme $L_n{}'$.

Definiamo l'insieme $P = \{(x, x) \mid x \in \{a, b\}^n\}$. Questo é un extended fooling set per $L_n{}'$ perché:
1. $xx \in L_n{}'$;

2. $xy \notin {L_n}' \vee yx \notin {L_n}'$.

La seconda proprietà vale perché essendo $x \neq y$ esiste almeno una coppia di caratteri a distanza n che non é uguale.

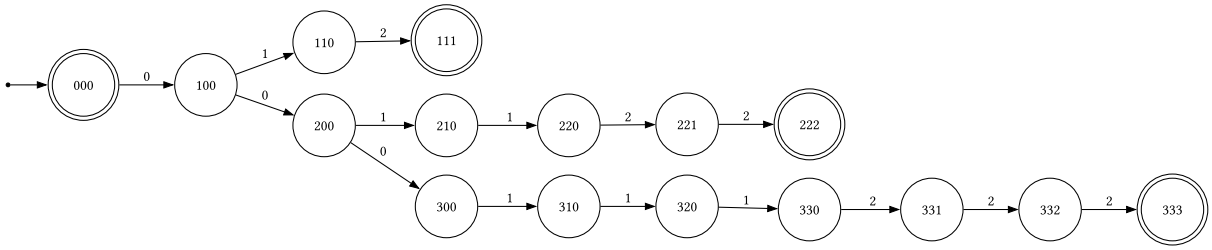
Ma allora ogni NFA per ${L_n}'$ ha almeno $|P| = 2^n$ stati.

3.7.3. Applicazione a L_k

L'ultimo esempio al quale applichiamo il concetto di extended fooling set é il linguaggio

$$L_k = \{0^i 1^i 2^i \mid 0 \leq i \leq k\}.$$

Vediamo il caso $k = 3$ e creiamo un NFA per questo linguaggio.



Diamo un lower bound al numero di stati di un NFA generico. Definiamo l'insieme

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

Questo é un extended fooling set per L_k, quindi ogni NFA per L_k ha almeno $|P| = (k+1)^2$ stati.

3.8. Espressioni regolari

3.8.1. Operazioni insiemistiche

Sia $L \subseteq \Sigma^*$ un linguaggio. Essendo un insieme, possiamo utilizzare le classiche operazioni di:

- intersezione $\bigcap$;
- unione $\bigcup$;
- complemento $()^C$.

Altre operazioni importanti sono:

- **prodotto/concatenazione** di linguaggi: dati $L_1, L_2 \subseteq \Sigma^*$ costruisco l'insieme

  $$L_1 \cdot L_2 = \{z \mid \exists x \in L_1 \wedge \exists y \in L_2 \mid z = xy\}.$$

  Questa operazione *non* é commutativa ma é associativa.
- **potenza** di un linguaggio: dato $L \subseteq \Sigma^*$ costruisco l'insieme

  $$L_k = \underbrace{L \cdot \ldots \cdot L}_{k}.$$

  Notiamo, usando l'induzione, come

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

3.8.2. Chiusura di Kleene

La **chiusura di Kleene**, o *star*, é definita come

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

In poche parole, la chiusura di Kleene rappresenta l'insieme di tutte le stringhe ottenute concatenando le parole di L, compresa la parola vuota ε.

L'insieme L^* é sempre infinito, tranne quando:
- $L = \{\varepsilon\}$, ottenendo $L^* = \{\varepsilon\}$;
- $L = \varnothing$, ottenendo $L^* = \{\varepsilon\}$ per definizione.

La **chiusura positiva** L^+ é definita come

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

Se $\varepsilon \in L$ la differenza tra chiusura e chiusura positiva non esiste, mentre se $\varepsilon \notin L$ si ottene $L^+ = L^* - \{\varepsilon\}$.

Si può dimostrare che

$$L^+ = L \cdot L^* = L \cdot \bigcup_{k \geq 0} L^k = \bigcup_{k \geq 0} L^{k+1} = \bigcup_{k \geq 1} L^k.$$

3.8.3. Definizione di espressione regolare

Le **espressioni regolari** sono un *modello dichiarativo* per le grammatiche di tipo 3, ovvero permettono di dichiarare la forma delle stringhe del linguaggio tramite una serie di operazioni.

Introduciamo una serie di coppie, dove il primo elemento rappresenta un'espressione nel mondo delle espressioni regolari e il secondo elemento rappresenta il linguaggio generato da quell'espressione:
- $\varnothing \longrightarrow$ linguaggio vuoto;
- $\varepsilon \longrightarrow \{\varepsilon\}$;
- $a \in \Sigma \longrightarrow \{a\}$;
- $E_1 + E_2 \longrightarrow L(E_1) \cup L(E_2)$;
- $E_1 \cdot E_2 \longrightarrow L(E_1) \cdot L(E_2)$;
- $E_1^* \longrightarrow [L(E_1)]^*$.

3.8.4. Teorema di Kleene

Vediamo il **teorema fondamentale degli automi a stati finiti** (o *teorema di Kleene*), che afferma la coincidenza tra la classe degli automi a stati finiti con la classe delle espressioni regolari.

Teorema 3.8.4.1 La classe di linguaggi accettati da automi a stati finiti é il più piccolo sottoinsieme di Σ^* che contiene i linguaggi finiti ed é chiusa rispetto alle operazioni di $\cup$, $\cdot$ e $*$.

Dimostrazione

Dimostriamo che a partire da un automa a stati finiti per L possiamo trovare un'espressione regolare per lo stesso linguaggio L.

Sia $A = (Q, \Sigma, \delta, q_1, F)$, con $Q = \{q_1, ..., q_n\}$. Accetto una parola se e solo se esiste un cammino nel grafo di computazione che finisce in uno stato finale.

Chiamo $R_{ij}^{(k)}$ le espressioni che iniziano nello stato q_i, finiscono nello stato q_j e visitano stati con indice $\leq k$:

$$q_i \rightsquigarrow q_{h \leq k} \rightsquigarrow q_j.$$

Definiamo $R_{ij}^{(0)}$ l'insieme di tutte le lettere che mi portano da q_i a q_j, quindi

$$R_{ij}^{(0)} = \{a \in \Sigma \mid \delta(q_i, a) = q_j\}.$$

Se $i = j$ allora

$$R_{ij}^{(0)} = \{\varepsilon\} \cup \{a \in \Sigma \mid \delta(q_i, a) = q_i\}.$$

Definiamo ora $R_{ij}^{(k)}$ sapendo tutte le espressioni fino a $R_{ij}^{(k-1)}$

Evidenzio tutti i punti del cammino che passano per lo stato q_k:

$$q_i \overset{\leq k-1}{\rightsquigarrow} q_k \overset{\leq k-1}{\rightsquigarrow} \dots \overset{\leq k-1}{\rightsquigarrow} q_k \overset{\leq k-1}{\rightsquigarrow} q_j.$$

Ma allora

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} \cup \left(R_{ik}^{(k-1)} \cdot R_{kk}^{(k-1)} \cdot R_{kj}^{(k-1)}\right).$$

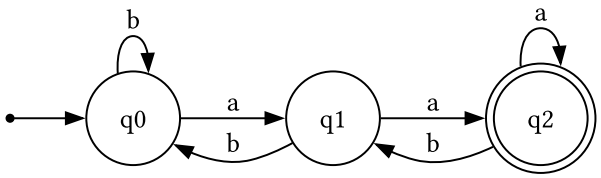
Il linguaggio L viene definito come

$$L = \bigcup_{q_i \in F} \text{cammini dallo stato 1 allo stato } q_i = \bigcup_{q_i \in F} R_{1i}^{(n)}.$$

□

3.8.5. Come costruire un'espressione regolare

Vediamo *come costruire un'espressione regolare a partire da un automa a stati finiti* con un esempio.



Per scrivere un'espressione regolare dobbiamo scrivere un sistema di n equazioni, dove $n = |Q|$. Ogni equazione E_i descrive il linguaggio che l'automa riconoscerebbe se si partisse dallo stato X_i. Ogni equazione é una somma di fattori, ognuno formato da un simbolo di Σ e uno stato in Q. La presenza di un fattore del tipo aX_j indica che dallo stato X_i si finisce nello stato X_j leggendo una a.

Scriviamo il sistema di equazioni per il DFA disegnato sopra, ricordandoci di aggiungere ε in caso l'equazione E_i descriva uno stato finale.

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

Risolviamo il sistema, introducendo una regola fondamentale:

$$X = aX + B \longrightarrow X = a^*B.$$

Questa ci permette di risolvere ogni sistema di equazioni che definisce un automa a stati finiti.

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

Raccogliamo X_2 nella seconda equazione e applichiamo la regola fondamentale precedente.

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

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

Sostituiamo X_2 dentro X_0 e applichiamo ancora la regola fondamentale dopo aver raccolto ogni fattore che contiene X_0 nell'equazione.

$$\begin{aligned} X_0 &= aa(a + ba)^*(bbX_0 + \varepsilon) + (ab + b)X_0 \\ X_0 &= aa(a + ba)^*bbX_0 + aa(a + ba)^* + (ab + b)X_0 \\ X_0 &= (aa(a + ba)^*bb + ab + b)X_0 + aa(a + ba)^* \\ X_0 &= (aa(a + ba)^*bb + ab + b)^*(aa(a + ba)^*). \end{aligned}$$

3.9. Studio della complessità

Sia $L \subseteq \Sigma^*$, la **complessità di stati** di L é il minimo numero di stati di un DFA che accetta L. La complessità di stati si indica con $\mathrm{sc}(L)$.

In modo analogo, si definisce $\mathrm{nsc}(L)$ la **complessità di stati non deterministica** come il minimo numero di stati di un NFA che accetta L.

Come sappiamo già, vale la relazione

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

In generale, ogni NFA ha almeno $n + 1$ stati, dove n é il numero di caratteri della stringa più corta del linguaggio.