Automi, Calcolabilità e Complessità

Alessio Marini, 2122855

Appunti presi durante il corso di **Automi, Calcolabilità e Complessità** nell'anno **2025/2026** del professore Daniele Venturi.
Gli appunti li scrivo principalmente per rendere il corso più comprensibile **a me** e anche per imparare il linguaggio Typst. Se li usate per studiare verificate sempre le informazioni 🙏🏼.

Contatti:
alem1105
marini.2122855@studenti.uniroma1.it

September 27, 2025

Contents

1. Automi 3
 1.1. Linguaggi Regolari 3
 1.2. Operazioni sui Linguaggi 7
 1.3. Non Determinismo 7
 1.4. Equivalenza tra DFA e NFA 9
 1.5. Chiusura per Linguaggi Regolari 12
 1.6. Espressioni Regolari 16
 1.7. Pumping Lemma 21
 1.8. Linguaggi Acontestuali 23
  1.8.1. Unione tra CFG 25
  1.8.2. Trasformare un DFA in un CFG 25
  1.8.3. Forma Normale di Chomsky 26
 1.9. Pushdown Automata - PDA 28
 1.10. Pumping Lemma per i CFL 33
2. Calcolabilità 35
 2.1. TM Multinastro 37
 2.2. TM non Deterministica 37
 2.3. Linguaggi Decidibili 38
 2.4. Riducibilità 43

1. Automi

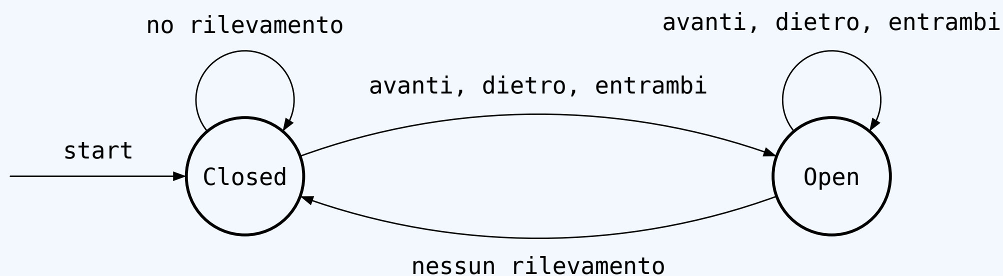
1.1. Linguaggi Regolari

Iniziamo introducendo in modo *informale* cos'è un **Automa**.
Un Automa è un modello matematico di calcolo che può elaborare informazioni e agire in modo automatico seguendo una serie di stati predefiniti.

Esistono diversi tipi di automi, il primo che vediamo si chiama **Automa a stati Finiti Deterministico (DFA)**, questo tipo di automa ha un **numero finito di stati** e processa gli input che riceve in modo **sequenziale** bit a bit.

Esempio - Sistema di controllo di una porta automatica

La porta ha dei sensori "avanti" e "dietro" per rilevare i movimenti ed aprire la porta.



Adesso però diamo una definizione più formale per i DFA.

Definizione - DFA

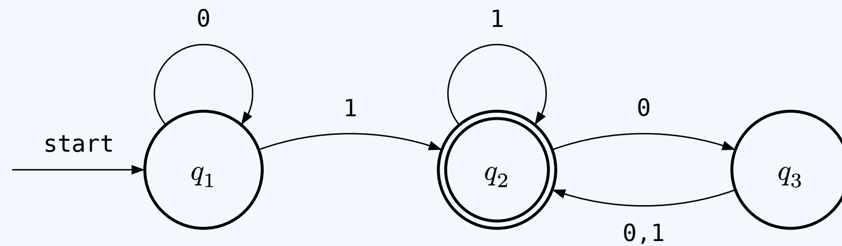
Un DFA è una quintupla $(Q, \Sigma, \delta, q_0, F)$ dove:

- Q: Insieme finito degli stati
- Σ: Insieme finito dei simboli in input (**alfabeto**)
- $\delta : Q \times \Sigma \to Q$: **Funzione di Transizione**, ci dice come avvengono i passaggi di stato
- q_0: Stato **iniziale**
- $F \subseteq Q$: Insieme degli **stati di accettazione**

Nei nostri diagrammi indicheremo gli stati di accettazione, o finali, con un *doppio bordo*. Inoltre diciamo che un automa "accetta" un input se termina in uno stato di accettazione, altrimenti diremo che lo "rifiuta".

Vediamo quindi un altro esempio di DFA.

Esempio - DFA di un linguaggio binario



Vediamo i vari elementi rispetto alla definizione che abbiamo dato prima:

- $Q = \{q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- $q_0 = q_1$
- $F = \{q_2\}$

Mentre per la δ possiamo o rappresentare una tabella o in questo caso semplicemente guardare le frecce del diagramma. Questo DFA, ad esempio, *accetta* se riceve come input la stringa `w=11101`

Introduciamo altre definizioni che ci serviranno per definire in modo formale il concetto di **linguaggio accettato**.

In modo informale un linguaggio è l'insieme di tutte le stringhe rinosciute da un DFA, se ad esempio abbiamo il DFA M allora l'insieme delle stringhe riconosciute da M lo denotiamo con $L(M)$. Ad esempio il DFA dell'esempio precedente ha come linguaggio:

$$A = \{w \mid w \text{ contiene almeno un '1' e un numero pari di '0' segue l'ultimo '1'}\}$$

E possiamo dire quindi che $L(M) = A$ o anche "M riconosce A".

Per definire il linguaggio accettato dobbiamo modificare la funzione di transizione e introdurre la **funzione di transizione estesa**.

Definizione - Funzione di Transizione Estesa

La denotiamo con δ^* e ha lo stesso funzionamento della funzione di transizione classica, la differenza è che insieme allo stato accetta un'intera stringa e non un solo carattere.

- $\delta^* : Q \times \Sigma^* \to Q$
- $\delta^*(q, \varepsilon) = \delta(q, \varepsilon)$
- $\delta^*(q, ax) = \delta^*(\delta(q, a), x)$

Con Σ^* indichiamo l'insieme delle stringhe in input e $x \in \Sigma^*, a \in \Sigma$. Questo significa quindi che la funzione di transizione estesa processa intere stringhe ma sempre in modo sequenziale carattere per carattere.

Introduciamo anche il concetto di **configurazione**, ovvero lo stato in cui si trova il DFA, in particolare è definita come una tupla $(q, x) \in Q \times \Sigma^*$ e indica appunto lo stato in cui si trova l'automa e quello che gli resta da leggere. La configurazione iniziale è quella che si trova nello stato q_0 quindi ad esempio (q_0, x).

Adesso possiamo definire anche il **passo di computazione** ovvero una relazione binaria tra configurazioni che ci porta appunto da una configurazione ad un'altra. Si ha:

$$(p, ax) \vdash_M (q, x) \Leftrightarrow \delta(p, a) = q$$

Con $p, q \in Q, a \in \Sigma, x \in \Sigma^*$. Stiamo dicendo quindi che passiamo dalla configurazione (p, ax) alla configurazione (q, x) leggendo a (ci resta poi da leggere x) che è equivalente a dire che da p passiamo a q se leggiamo a.

Anche la relazione binaria $\vdash_M$ può essere estesa considerando la **chiusura riflessiva e transitiva** e la denotiamo con $\vdash_M^*$:

- $(q, x) \vdash_M^* (q, x)$, ovvero quando non legge nessun input.
- $(q, aby) \vdash_M (p, by) \land (p, by) \vdash_M (r, y) \Rightarrow (q, aby) \vdash_M^* (r, y)$, possiamo quindi raccogliere piú passi di configurazione.

Con $p, q, r \in Q; a, b \in \Sigma; y \in \Sigma^*$

Adesso possiamo finalmente utilizzare tutti questi concetti per formalizzare la definizione di **linguaggio accettato**.

Definizione - Linguaggio Accettato

Diciamo che $x \in \Sigma^*$ è accettato da $M = (Q, \Sigma, \delta, q_0, F)$ se

$$\delta^*(q_0, x) \in F \quad \text{oppure} \quad (q_0, x) \vdash_M^* (q, \varepsilon) \text{ con } q \in F$$

Possiamo definirlo anche come insieme ovvero $L(M) = \{x \in \Sigma^* : \delta^*(q_0, x) \in F\}$

Adesso che sappiamo cos'è un DFA e il suo linguaggio accettato, possiamo definire i **linguaggi regolari**.

Definizione - Linguaggio Regolare

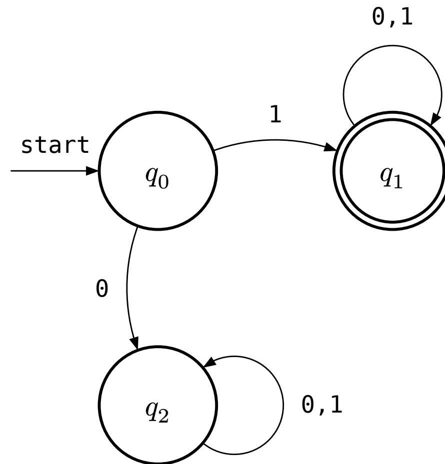
Un linguaggio è chiamato **regolare** se un DFA lo riconosce, indichamo con `REG` l'insieme dei linguaggi regolari:

$$\text{REG} = \{L \subseteq \Sigma^* \mid \exists \text{ DFA } M \text{ t.c. } L(M) = L\}$$

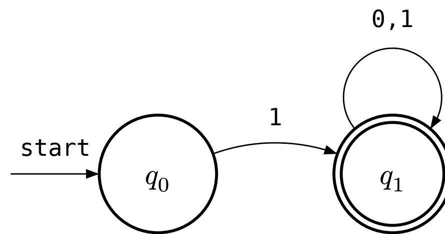
Ci interessa capire come costruire un DFA dato un certo linguaggio, ad esempio prendiamo il linguaggio:

$$L = \{x \in \{0,1\}^* \mid x = 1 \| y, y \in \{0,1\}^*\}$$

L'operatore $\|$ indica la concatenazione di stringhe, in questo caso quindi stiamo indicando tutte le stringhe che iniziano con 1. Proviamo a costruire un DFA A che riconosce il linguaggio:



Lo stato q_2 viene chiamato "stato pozzo", ovvero uno stato non di accettazione che non ha archi uscenti verso altri stati. Possiamo disegnare anche un DFA equivalente senza gli archi di q_2:



In questo caso se in q_0 riceviamo 0 semplicemente ignoramo l'input.

Adesso peró vogliamo dimostrare la correttezza del DFA appena costruito, ovvero dimostrare che il DFA accetta x **se e solo se** $x \in L$. Costruiamo una dimostrazione per induzione, per prima cosa però notiamo due cose nel nostro DFA:

- $\delta^*(q_1, u) = q_1 \ \forall u \in \{0,1\}^*$
- $\delta^*(q_2, u) = q_2 \ \forall u \in \{0,1\}^*$

Stiamo dicendo che se entriamo in q_1 o q_2 non andremo più in altri stati. Adesso iniziamo la dimostrazione per induzione:

- **Caso Base** - Abbiamo $|x| = 0$ ovvero $x = \varepsilon$ stringa vuota. Si ha che

$$\delta^*(q_0, \varepsilon) = \delta(q_0, \varepsilon) = q_0 \notin F$$

- **Ipotesi Induttiva** - Sia $n > 0$, supponiamo $|w| \leq n$. Si ha:

$$\delta^*(q_0, w) = \begin{cases} q_1 \text{ se } w \text{ inizia con } 1 \\ q_2 \text{ se } w \text{ inizia con } 0 \end{cases}$$

  Possiamo dirlo perchè abbiamo visto all'inizio che una volta entrati in q_1 o q_2 il DFA non cambia più stato.

- **Passo Induttivo** - Prendiamo x t.c. $|x| = n + 1$, possiamo scomporla in $x = au$ con $a \in \{0,1\}$ e $u \in \{0,1\}^*$. Otteniamo quindi

$$\delta^*(q_0, x) = \delta^*(q_0, au) = \delta^*\left(\underbrace{\delta(q_0, a)}_{\text{ha 2 soluzioni}}, u\right)$$

  Le 2 soluzioni sono quelle che abbiamo visto nell'ipotesi induttiva, infatti:

$$\delta(q_0, a) = \begin{cases} q_2 \text{ se } a = 0 \\ q_1 \text{ se } a = 1 \end{cases}$$

Dato che abbiamo dimostrato che l'automa non uscirá mai da nessuno dei due stati e che entra sicuramente in uno dei due possiamo quindi dire che:

$$\delta^*(q_0, x) = q_1 \Leftrightarrow a = 1$$

1.2. Operazioni sui Linguaggi

I linguaggi non sono altro che insiemi di stringhe, questo significa che possiamo applicare delle operazioni su di essi, vediamole alcune:

- **Unione**: $L_1 \cup L_2 = \{x \in \Sigma^* \mid x \in L_1 \vee x \in L_2\}$
- **Intersezione**: $L_1 \cup L_2 = \{x \in \Sigma^* \mid x \in L_1 \wedge x \in L_2\}$
- **Complemento**: $\overline{L_1} = \{x \in \Sigma^* \mid x \notin L_1\}$
- **Concatenzaione**: $L_1 \circ L_2 = \{xy\} \mid x \in L_1 \wedge y \in L_2\}$ importante notare che non é commutativi quindi $L_1 \circ L_2 \neq L_2 \circ L_1$.

  Vediamo un esempio, prendiamo:

  - $\Sigma = \{a, b\}$
  - $L_1 = \{a, ab, ba\}$
  - $L_2 = \{ab, b\}$

  Otteniamo $L_1 \circ L_2 = \{aab, ab, abab, abb, baab, bab\}$
- **Potenza**: La definiamo in modo ricorsivo

  $$\text{per stringhe} \begin{cases} x^0 & = \varepsilon \\ x^{n+1} & = x^n x \end{cases} \text{; per linguaggi} \begin{cases} L^0 & = \{\varepsilon\} \\ L^{n+1} & = L^n \circ L \end{cases}$$

  Quindi ad esempio se prendiamo $L = \{a, ab, ba\}$ possiamo dire:

  $$L^2 = L \circ L = \{aa, aab, aba, aba, abab, abba, baa, baab, baba\}$$
- **Operatore** *: $L^* = \bigcup_{n \geq 0} L^n = \{\varepsilon\} \cup L^1 \cup L^2 \cup \ldots$ ovvero tutte le combinazioni di stringhe possibili. Ad esempio se $L = \{a, b\}$ abbiamo $L^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$

1.3. Non Determinismo

Per ora abbiamo visto come un DFA che si trova in uno stato quando processa un input farà un solo movimento verso un altro stato, oppure rimane nello stesso. Nel non determinismo invece abbiamo dei comportamenti diversi:

- Quando un automa si trova in uno stato e processa un input può andare in **diversi** stati **contemporaneamente**.
- Sono ammessi gli ε-archi, ovvero degli archi che non hanno bisogno di input per essere percorsi.
- L'automa a questo punto, visto che esplora più rami di computazione contemporaneamente, accetta se esiste almeno un ramo che accetta. Vedremo successivamente cosa significa "ramo di computazione".

A questo punto non parliamo più di DFA ma di **NFA** ovvero automi **non deterministici** a stati finiti.

Definizione - NFA

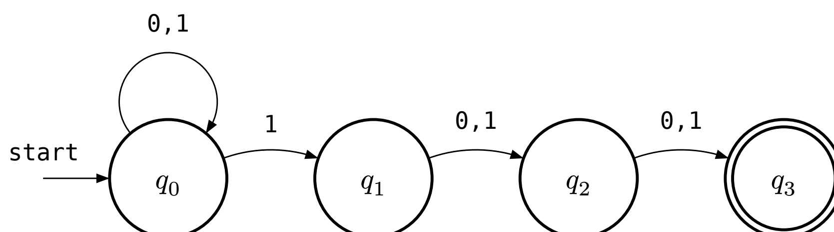
Un NFA è una tupla $(Q, \Sigma, \delta, q_0, F)$ dove tutti gli elementi sono come per i DFA ma abbiamo un cambio nella funzione di transizione. infatti questa diventa:

- $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ con Σ_ε indichiamo $\Sigma \cup \{\varepsilon\}$, mentre con $\mathcal{P}(Q)$ l'insieme delle parti di Q ovvero l'insieme di tutti i possibili sottoinsiemi di Q

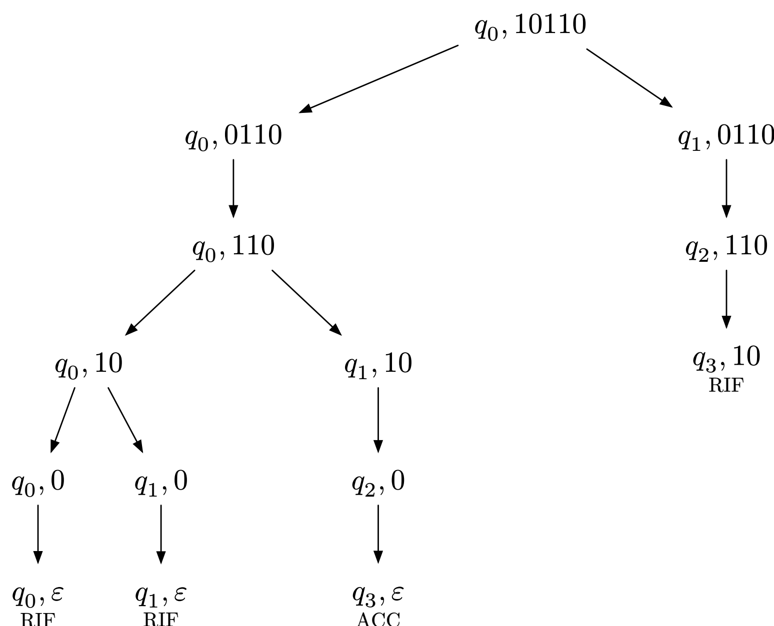
Per chiarire il concetto di "ramo di computazione" vediamo un esempio di NFA. Costruiamo un NFA che riconosce il linguaggio

$$L = \{x : x \in \{0,1\}^* \land x \text{ ha un '1' in terzultima posizione}\}$$

Costruiamo l'NFA:



Cosa succede quanto l'NFA computa la stringa `10110`? In un NFA quando uno stato permette più transizioni si creano più rami, uno per ogni strada presa, ad esempio con l'NFA appena costruito se inseriamo la stringa `10110` con il primo carattere 1 l'NFA andrà sia in q_0 che in q_1 e questo porta la creazione di due rami di computazione. L'automa accetta se esiste almeno un ramo fra questi che accetta. Per chiarezza vediamo quali rami si creano in questo esempio:

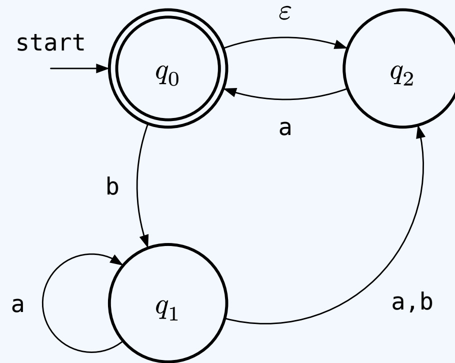


Per quanto riguarda gli ε-archi invece, se l'automa si trova in uno stato che ne possiede uno allora l'automa si duplica in due rami senza leggere nessun input:

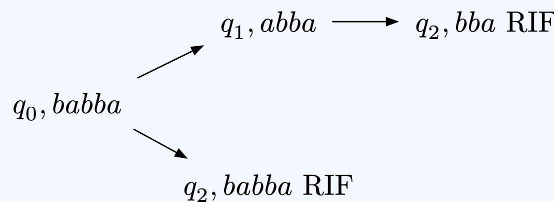
- Un ramo rimane nella stessa configurazione di prima.
- L'altro ramo segue l'ε-arco.

Vediamo un altro esempio per chiarire anche questo concetto.

Esempio - NFA con ε-archi



Vediamo cosa succede con la computazione della stringa `babba`



Dobbiamo adesso rivedere i concetti di **configurazione, computazione ed accettazione** in un NFA:

- Dato un NFA N, indichiamo come configurazione una coppia $(q, x) \in Q, \Sigma_\varepsilon^*$
- Avremo

$$(p, ax) \vdash_N (q, x) \Leftrightarrow q \in \delta(p, a) \quad \text{con } x \in \Sigma_\varepsilon^*; a \in \Sigma_\varepsilon; p, q \in \Sigma$$

  Da notare che q, p non sono più degli stati singoli ma un insieme di stati infatti $q, p \in \mathcal{P}(Q)$, ovviamente possono anche contenere un singolo stato ma saranno comunque insiemi di stati.

- Un NFA N accetta

$$w \in \Sigma_\varepsilon^* \Leftrightarrow \exists q \in F \text{ t.c. } (q_0, w) \vdash_N^* (q, \varepsilon)$$

1.4. Equivalenza tra DFA e NFA

Dimostriamo come in realtá le due classi di automi sono equivalenti. Per prima cosa definiamo in modo formale le due classi:

- $\mathcal{L}(\text{DFA}) = \text{REG}$, quindi la classe dei linguaggi regolari, riconosciuti da almeno un DFA.
- $\mathcal{L}(\text{NFA}) = \{L | \exists \text{ NFA } N \text{ t.c. } L(N) = L\}$, la classe dei linguaggi riconosciuti da almeno un NFA.

Teorema

$\text{REG} := \mathcal{L}(\text{DFA}) = \mathcal{L}(\text{NFA})$

Dimostrazione - Formalmente vogliamo dimostrare $\mathcal{L}(\text{DFA}) \subseteq \mathcal{L}(\text{NFA})$ e $\mathcal{L}(\text{NFA}) \subseteq \mathcal{L}(\text{DFA})$

1. **Prima implicazione** $\mathcal{L}(\text{DFA}) \subseteq \mathcal{L}(\text{NFA})$

Dato $L \in \mathcal{L}(\text{DFA})$ e $D \coloneqq (Q, \Sigma, \delta, q_0, F)$ t.c. $L = L(D)$. Notiamo che il concetto di NFA, in realtà, non è altro che una generalizzazione di un DFA, infatti se prendiamo un qualsiasi DFA possiamo dire che è anche un NFA, quindi D è un NFA e $L \in \mathcal{L}(\text{NFA})$.

2. **Seconda implicazione** $\mathcal{L}(\text{NFA}) \subseteq \mathcal{L}(\text{DFA})$

   Per dimostrare questa implicazione prendiamo un NFA e costruiamo un DFA che lo simula. Sia
   - $N = \{Q, \Sigma, \delta, q_0, F\}$ l'NFA t.c. $L(N) = A$

   costruiamo il DFA
   - $M = \{Q_M, \Sigma, \delta_M, q_0^M, F_M\}$ t.c. $L(M) = A$

   Prima facciamo una costruzione senza considerare gli ε-archi:

   - $Q_M = \mathcal{P}(Q)$ - Ogni stato di M è un insieme di stati N
   - $\delta_M(R, a) = \bigcup_{r \in R} \delta(r, a)$

     Dove $R \in Q_M, a \in \Sigma, \delta(R, a) = \{q \in Q \mid q \in \delta(r, a) \text{ con } r \in R\}$, stiamo dicendo che la funzione di transizione si applica su un insieme di stati e restituisce un altro insieme di stati, precisamente quelli che si ottengono applicando la funzione di transizione su tutti gli elementi dell'insieme in input.
   - $q_0^M = \{q_0\}$
   - $F_M = \{R \in Q_M \mid R \cap F \neq \varnothing\}$ ovvero quegli insiemi di stati che hanno al loro interno almeno uno stato accettante del DFA.

   A questo punto introduciamo gli ε-archi e per farlo definiamo

   $$E(R) = \{q \mid q \text{ può essere raggiunto da } R \text{ attraverso } 0 \text{ o più } \varepsilon - \text{archi}\}$$

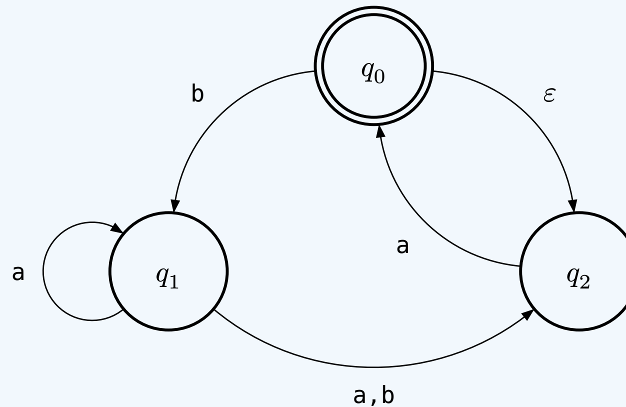
   Scriviamo 0 o più perchè in questo modo includiamo anche il nodo su cui ci troviamo. Con questa definizione possiamo modificare la funzione di transizione del DFA e lo stato iniziale del DFA in:

   - $\delta_M(R, a) = \{q \in Q \mid q \in E(\delta(r, a)), \exists r \in R\}$. In questo modo includiamo sia gli archi visti con la definizione di prima ma anche quelli che raggiungiamo tramite gli ε-archi.
   - $q_0^M = \{q_0\}$

Vediamo un esempio di trasformazione da NFA a DFA per chiarire i concetti.

Esempio - Da NFA a DFA

Prendiamo questo NFA come esempio, ha come linguaggio $\Sigma = \{a, b\}$



Iniziamo a dare le prime definizioni:

- $Q_D = \left\{q_\varnothing, q_{\{0\}}, q_{\{1\}}, q_{\{2\}}, q_{\{0,1\}}, q_{\{0,2\}}, q_{\{1,2\}}, q_{\{0,1,2\}}\right\}$. Con la notazione $q_{\{0,1\}}$ indichiamo l'insieme di stati $\{q_0, q_1\}$
- $q_0^D = E\left(q_{\{0\}}\right) = q_{\{0,2\}}$. Infatti da q_0 ci muoviamo con un ε-arco in q_2 ma manteniamo anche il ramo che si trova in q_0.
- $F_D = \left\{q_{\{0\}}, q_{\{0,1\}}, q_{\{0,2\}}, q_{\{0,1,2\}}\right\}$. Ovvero tutti gli stati che contengono lo stato q_0

Adesso ci manca da definire la funzione di transizione, è un passaggio lungo quindi vediamo solo alcuni casi.

Per lo stato $q_{\{0\}}$ abbiamo:

- $\delta_D\left(q_{\{0\}}, a\right) = E(\delta(q_0, a)) = q_{\{0,2\}}$
- $\delta_D\left(q_{\{0\}}, b\right) = E(\delta(q_0, b)) = q_{\{1,2\}}$

Per lo stato $q_{\{1\}}$:

- $\delta_D\left(q_{\{1\}}, a\right) = E(\delta(q_1, a)) = q_{\{1,2\}}$
- $\delta_D\left(q_{\{1\}}, b\right) = E(\delta(q_1, b)) = q_{\{2\}}$

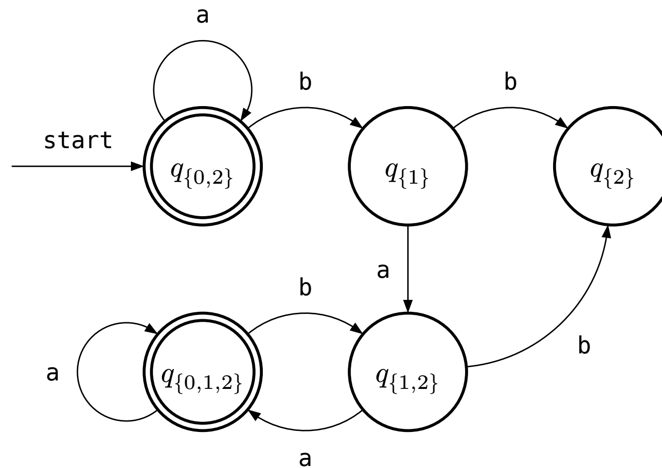
…

Per lo stato $q_{\{0,1\}}$:

- $\delta_D\left(q_{\{0,1\}}, a\right) = E(\delta(q_0, a)) \cup E(\delta(q_1, a)) = q_{\{0,2\}} \cup q_{\{1,2\}} = q_{\{0,1,2\}}$
- …

…

Basandoci sull'esempio di prima possiamo anche provare a costruire il DFA che abbiamo soltanto definito. (semplificando anche alcuni nodi mai raggiunti)



Da adesso in poi quindi non ci importa più distinguere NFA e DFA dato che abbiamo visto essere equivalenti, possiamo usare benissimo entrambi in qualsiasi situazione, ovviamente in alcuni casi potrebbe essere più vantaggioso un determinato tipo.

1.5. Chiusura per Linguaggi Regolari

I linguaggi regolari sono **chiusi** rispetto alle loro operazione? Ovvero, se applichiamo un'operazione a due linguaggi regolari otteniamo come risultato un altro linguaggio regolare?

Teorema

L'insieme REG è chiuso per unione. Ovvero $L_1, L_2 \in \text{REG} \Rightarrow L_1 \cup L_2 \in \text{REG}$

Dimostrazione - Possiamo svolgere la dimostrazione in due modi:

1. Senza NFA

   Per dimostrare il teorema utilizzando i DFA dobbiamo costruire un automa M che riconosce il linguaggio $L_1 \cup L_2$. Per farlo dobbiamo fare in modo che M simuli il comportamento di altri due automi M_1, M_2 tali che $L(M_1) = L_1$ e $L(M_2) = L_2$, e che accetti soltanto se almeno una delle due simulazioni accetta.

   Dati

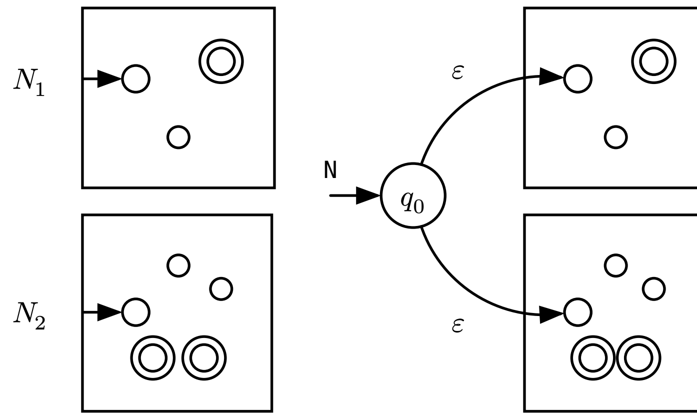
   - $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ t.c. $L(M_1) = L_1$
   - $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ t.c. $L(M_2) = L_2$

   Costruiamo quindi $M = (Q, \Sigma, \delta, q_0, F)$ dove:

   - $Q = Q_1 \times Q_2 = \{(r_1, r_2) \mid r_1 \in Q_1 \wedge r_2 \in Q_2\}$
   - $\delta : Q \times \Sigma \to Q$. Ad esempio $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta(r_2, a))$
   - $q_0 = (q_1, q_2)$
   - $F = \{(r_1, r_2) \mid r_1 \in F_1 \vee r_2 \in F_2\} = (F_1 \times Q_2) \cup (Q_1 \times F_2)$

2. Con NFA

   In questo caso la dimostrazione è molto semplice, infatti ci basiamo sempre sull'idea di prima di usare due simulazioni N_1, N_2 per i due linguaggi ma le inglobiamo in un NFA N che avrá come nodo iniziale un nuovo nodo collegato ai nodi iniziali delle simulazioni tramite ε-archi:



Più formalmente, dati

- $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ t.c. $L(N_1) = L_1$
- $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ t.c. $L(N_2) = L_2$

Costruiamo $N = (Q, \Sigma, \delta, q_0, F)$ dove:

- $Q = \{q_0\} \cup Q_1 \cup Q_2$
- $F = F_1 \cup F_2$
- $\delta : Q \times \Sigma \to Q$ dove

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0, a = \varepsilon \\ \varnothing & q = q_0, a \neq \varepsilon \end{cases}$$

Teorema

REG è chiuso per complemento. Ovvero $\forall L \in \text{REG}, \overline{L} \in \text{REG}$

Dimostrazione - Costruisco un automa $\overline{L}$ che accetta tutto tranne le stringhe accettate da L. Per farlo ci basta avere come stati finali il complemento dell'insieme degli stati finali.

Dato $L = (Q, \Sigma, \delta, q_0, F)$ t.c. $L(L) = A$ costruisco $\overline{L} = (Q, \Sigma, \delta, q_0, Q - F)$

Teorema

REG è chiuso per intersezione. Ovvero $\forall L_1, L_2 \in \text{REG}, L_1 \cap L_2 \in \text{REG}$

Dimostrazione - Dati $D_1 = (Q, \Sigma, \delta_1, q_1, F_1)$ t.c. $\mathcal{L}(D_1) = L_1$ e $D_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ t.c. $\mathcal{L}(D_2) = L_2$ costruisco $M = (Q, \Sigma, \delta, q_0, F)$ t.c.:

- $Q = Q_1 \times Q_2$
- $F = F_1 \times F_2$
- $q_0 = (q_1, q_2)$
- $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$

Teorema

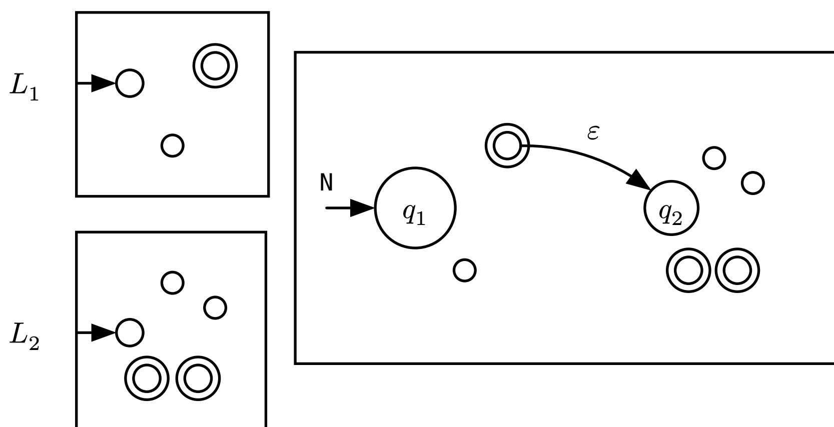
REG è chiuso per concatenazione. $\forall L_1, L_2 \in \text{REG}, L_1 \circ L_2 \in \text{REG}$

Dati $D_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ t.c. $\mathcal{L}(D_1) = L_1$ e $D_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ t.c. $\mathcal{L}(D_2) = L_2$ costruisco $N = (Q, \Sigma, \delta, q_0, F)$ t.c.:

- $Q = Q_1 \cup Q_2$
- $q_0 = q_1$
- $F = F_2$
- $\forall q \in Q, a \in \Sigma$ si ha che

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 - F_1 \\ \delta_1(q,a) & q \in F_1 \wedge a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1 \wedge a = \varepsilon \\ \delta_2(q,a) & q \in Q_2 \end{cases}$$

Graficamente ci basta prendere il primo automa L_1 e collegare i suoi stati finali allo stato iniziale di L_2 tramite ε-archi:



Da questa dimostrazione possiamo anche dire che REG è chiuso per potenza, dato che la potenza si esprime in funzione della concatenazione. *Es* $L^2 = L \circ L$

Corollario

REG è chiuso per l'operatore potenza. $\forall L \in \text{REG}, n \in \mathbb{N}, L^n \in \text{REG}$.

Teorema

REG è chiuso per l'operatore $*$. $\forall L \in \text{REG}, L^* \in \text{REG}$.

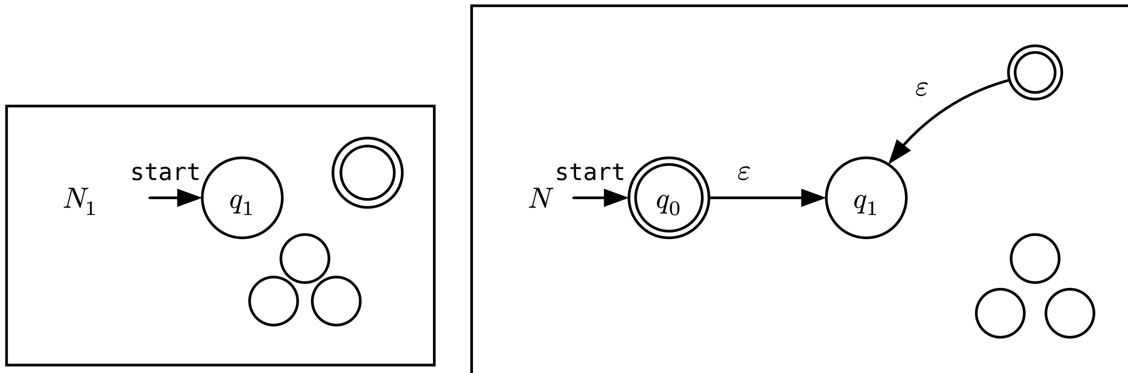
Dimostrazione - Dato $L \in \text{REG}$, sia $N_1 = (Q_1, \Sigma, \delta_1, q_1, F)$ l'NFA t.c. $L(N_1) = L$ costruisco $N = (Q, \Sigma, \delta, q_0, F)$ NFA t.c.:

- $q_0 \neq q_1$ è un nuovo stato iniziale
- $Q = Q_1 \cup \{q_0\}$
- $F = F_1 \cup \{q_0\}$. Dobbiamo aggiungere anche il nuovo stato iniziale dato che l'operatore $*$ comprende anche la stringa vuota ε.

- $\forall q \in Q, a \in \Sigma$ si ha che

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q - F \\ \delta_1(q,a) & q \in F \land a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F \land a = \varepsilon \\ \{q_1\} & q = q_0 \land a = \varepsilon \\ \varnothing & q = q_0 \land a \neq \varepsilon \end{cases}$$

Questo nuovo automa accetta tutto quello che accetta N_1 ma ogni volta che accetta torna indietro con un ε-arco per provare ad accettare un'altra stringa accettata sempre da N_1. Proviamo a rappresentare questo automa:



Come prima, grazie a questa dimostrazione possiamo anche dire che REG è chiuso per +. L'operatore + é come l'operatore $*$ ma esclude la potenza 0, posiamo definirlo quindi come $L^+ = L^1 \cup L^2 \cup \ldots$

Notazione

L^+ equivale quindi a scrivere LL^*, ovvero almeno una concatenazione di L.

Corollario

REG è chiuso per $+$. La dimostrazione è uguale a $*$ ma ci basta escludere q_0 da F.

1.6. Espressioni Regolari

Le espressioni regolari sono come le espressioni matematiche ma invece di rappresentare numeri, rappresentano linguaggi. Diamo una definizione formale.

Definizione - Espressione Regolare

Sia Σ un alfabeto. Un'espressione regolare su Σ che indichiamo con $re(\Sigma)$ è definita ricorsivamente come segue:

$$\text{caso base: } \begin{cases} \varnothing & \in re(\Sigma) \\ \varepsilon & \in re(\Sigma) \\ a & \in re(\Sigma), \forall a \in \Sigma \end{cases}$$

$$\text{caso induttivo: } \begin{cases} R_1 \cup R_2 & R_1, R_2 \in re(\Sigma) \\ R_1 \circ R_2 & R_1, R_2 \in re(\Sigma) \\ (R_1)^* & R_1 \in re(\Sigma) \end{cases}$$

Dove R_i sono espressioni regolari.

Stiamo quindi dicendo che, preso un alfabeto, i suoi singoli caratteri, la stringa vuota e l'insieme vuoto appartengono all'espressione regolare che descrive questo alfabeto. L'unione, l'intersezione o la $*$ di espressioni regolari su questo alfabeto sono comunque espressioni regolari dell'alfabeto.

Osservazione - Ogni espressione regolare $R \in re(\Sigma)$ ha associato un linguaggio $L(R)$:

$$\text{caso base: } \begin{cases} L(R) = \varnothing & \text{se } R = \varnothing \\ L(R) = \{\varepsilon\} & \text{se } R = \varepsilon \\ L(R) = \{a\} & \text{se } R = a \end{cases}$$

$$\text{caso induttivo: } \begin{cases} L(R) = L(R_1) \cup L(R_2) & \text{se } R = R_1 \cup R_2 \\ L(R) = L(R_1) \circ L(R_2) & \text{se } R = R_1 \circ R_2 \\ L(R) = (L(R_1))^* & \text{se } R = (R_1)^* \end{cases}$$

Vediamo qualche esempio di espressione regolare.

Esempi

- $0^*10^* = \{w \mid w \text{ contiene un solo } 1\}$
- $(0 \cup 1) = \{0\} \cup \{1\} = \{0, 1\}$
- $(\Sigma\Sigma)^* = \{w \mid w \text{ ha lunghezza pari}\}$. Prima concatena 2 simboli qualsiasi dell'alfabeto e poi li eleva, quindi avremo semrpe $2 * n$ simboli

Teorema

Un linguaggio è regolare **se e solo se** esiste un'espressione regolare che lo descrive, ovvero:

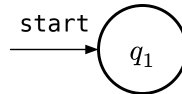
$$\text{REG} \equiv \mathcal{L}(re)$$

Dimostrazione - Dimostriamo il teorema tramite due lemmi.

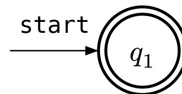
1. **Lemma** $\mathcal{L}(re) \subseteq \text{REG}$

Data un'espressione regolare R, costruisco un NFA/DFA N_R t.c. $L(N_R) = L(R)$. Per convertire R in un NFA consideriamo i casi della definizione ricorsiva di R. Prima vediamo i 3 casi base e poi i 3 induttivi.

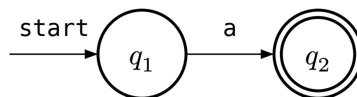
1. $R = \varnothing$. È un NFA che non accetta niente.
   - $N_R = \{\{q_1\}, \Sigma, \delta, q_1, \varnothing\}$
   - $\delta(q_1, b) = \varnothing, \forall b \in \Sigma$



2. $R = \varepsilon$. È un NFA che accetta soltanto la stringa vuota.
   - $N_R = \{\{q_1\}, \Sigma, \delta, q_1, \{q_1\}\}$
   - $\delta(q_1, b) = \varnothing, \forall b \in \Sigma$



3. $R = a, a \in \Sigma$. È un NFA che accetta soltanto il carattere a.
   - $N_R = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$
   - $\delta(q_1, a) = q_2$
   - $\delta(q, b) = \varnothing, b \neq a$



Passiamo adesso ai 3 casi ricorsivi.

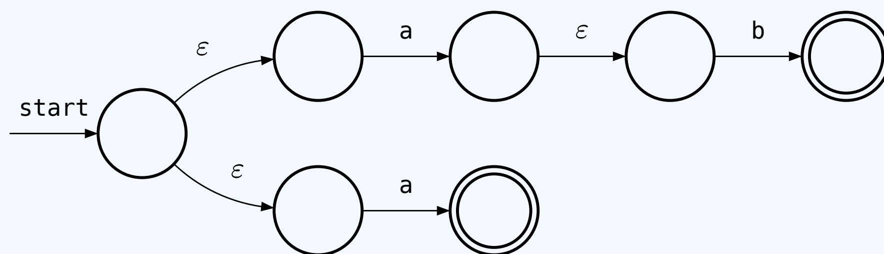
4.5.6. $R = R_1 \cup R_2; R = R_1 \circ R_2; R = (R_1)^*$. Per costruire i casi induttivi ci basta utilizzare le chiusure dei linguaggi regolari. Ad esempio se dobbiamo costruire $R = R_1 \cup R_2$ ci basta prima costruire i due automi per R_1, R_2 e poi utilizzare la chiusura come visto nelle dimostrazioni precedenti.

Esempio

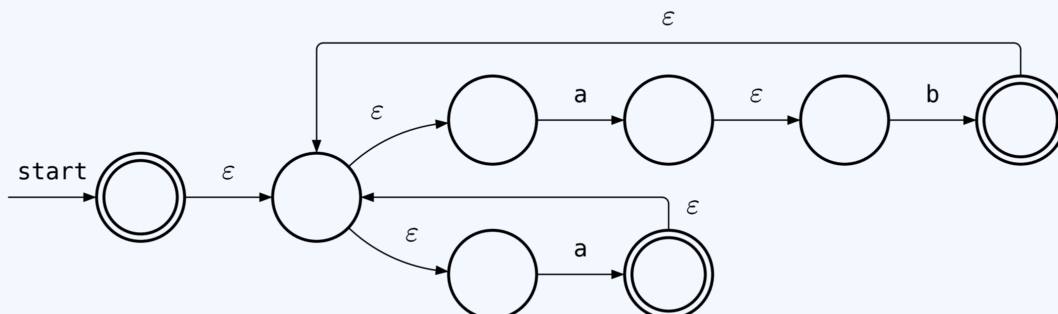
Convertiamo l'espressione regolare $(ab \cup a)^*$ in un NFA: Per prima cosa costruiamo gli automi per riconoscere i caratter a e b:

Poi costruiamo quello per riconoscere ab usando appunto la concatenazione:

Adesso usiamo l'unione per costruire l'NFA che riconosce $ab \cup a$:



Possiamo finalmente usare la chiusura per $*$ e costruire l'NFA:



2. **Lemma** $\text{REG} \subseteq \mathcal{L}(re)$. Adesso dobbiamo mostrare come passare da un linguaggio regolare ad un'espressione regolare. Per farlo dobbiamo imparare a convertire gli NFA in espressioni regolari.

   Per fare questo ci serve il concetto di **NFA Generalizzato (GNFA)**:

   - Le etichette degli archi sono espressioni regolari
   - Lo stato iniziale ha solo archi uscenti verso ogni altro stato
   - C'è un solo stato finale, che ha solo archi entranti da altri stati ed è diverso dallo stato iniziale
   - Eccetto gli stati iniziale e finale, presi due stati qualsiasi (anche lo stesso) esiste un arco tra di essi.

   Per prima cosa convertiamo l'NFA in un GNFA tramite il seguente algoritmo:

   - Se lo stato iniziale ha archi entranti, aggiungere un nuovo stato iniziale con un ε-arco al vecchio stato iniziale.
   - Se c'è più di uno stato accettante o esiste uno stato accettante con archi uscenti, aggiungere un nuovo stato finale con ε-archi provenienti da ogni $q \in F$.

- Se qualche arco ha più di un input, usare come input per quell'arco l'unione tra gli input precedenti.
- Aggiungere archi con input $\varnothing$ tra le coppie di stati non unite da archi.

Possiamo anche fornire una definizione formale di **GNFA**.

Definizione - GNFA

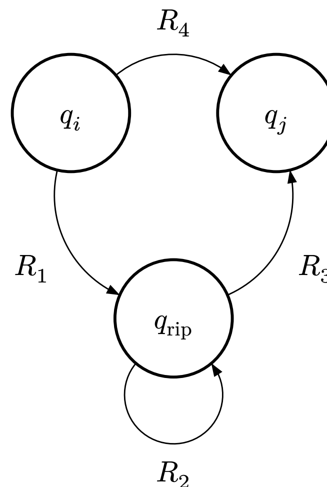
Un GNFA è una tupla $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$ dove gli elementi sono definiti come negli NFA ad eccezione della funzione di transizione, infatti:

$$\delta : (Q \setminus \{q_{\text{accept}}\}) \times (Q \setminus \{q_{\text{start}}\}) \to \mathcal{R} = re(\Sigma)$$

Conversione da GNFA a Espressione Regolare - Adesso possiamo convertire il GNFA in espressione regolare. Definiamo la funzione `CONVERT(G)` che prende in input un GNFA e restituisce un'espressione regolare:

1. Sia $k = \#$stati di G
2. Se $k = 2$ allora G ha solo i due stati q_{start} e q_{accept} e un singolo arco con etichetta $R \in \mathcal{R}$, l'output sarà quindi R.
3. Se $k > 2$ scegliamo un $q_{\text{rip}} \in Q$ diverso da q_{start} e q_{accept}. Definiamo $G' = (Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$ dove:
   - $Q' = Q \setminus \{q_{\text{rip}}\}$
   - $\delta' : Q' \setminus \{q_{\text{acc}}\} \times Q' \setminus \{q_{\text{start}}\} \to \mathcal{R}$

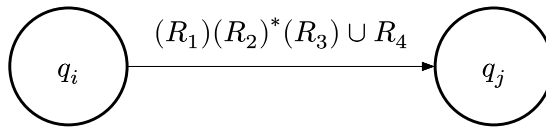
   Quindi prendiamo uno stato q_{rip} a caso e lo rimuoviamo. Come si rimuove? Vediamo un esempio con il seguente automa:



Dobbiamo vedere quali sono tutti i modi che abbiamo per andare da q_i a q_j, ovvero:
- Direttamente fra i due nodi: espressione R_4
- Considerando anche il passaggio per q_{rip}: prima si attraversa R_1 poi R_2 quante volte vogliamo e infine R_3

Scriviamo quindi la nuova espressione che andrà sull'arco da q_i a q_j:
- $\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup R_4$

E possiamo infine ridisegnare il nuovo automa:



4. Si lancia ricorsivamente `CONVERT(G')` fino ad arrivare ad un automa con 2 stati.

Adesso però dobbiamo dimostrare che il risultato di `CONVERT(G)` è equivalente a G.

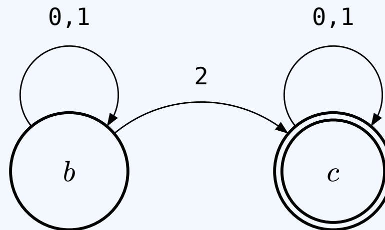
Dimostrazione - Si dimostra per induzione su k numero di stati di G

- **Caso Base**: $k = 2$ è vero perché banalmente ci sono solo due stati e un solo arco che li collega quindi otteniamo lo stesso identico automa applicando la funzione.
- **Passo Induttivo**: Supponendo quindi per $k - 1$ stati valga $G \equiv$ `CONVERT(G)`, se G accetta $w \in \Sigma^*$ esiste un ramo di computazione t.c. G percorre $q_{\text{start}}, q_1, ..., q_{\text{acc}}$ e in questo percorso facciamo due distinzioni:
  - Se la sequenza non contiene q_{rip} allora banalmente i linguaggi degli automi sono uguali $L(G) = L(G')$ dato che non abbiamo rimosso nulla dal percorso.
  - Se la sequenza contiene q_{rip}, gli stati adiacenti ad esso che chiamiamo q_1 e q_2, in G' avranno un nuovo arco che tiene conto di tutti i modi per passare da q_1 a q_2 anche senza q_{rip} e quindi anche in questo caso manteniamo il linguaggio uguale.
- Quindi $G \equiv$ `CONVERT(G)` oppure $L(G) = L(G')$

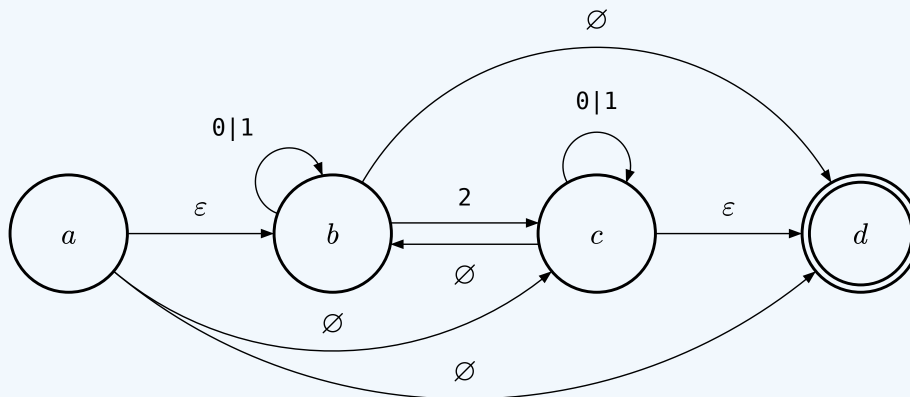
Facciamo un esempio completo per chiarire il procedimento.

Esempio - Conversione da NFA a Esp. Regolare

Dobbiamo trovare l'espressione regolare associata all'automa:



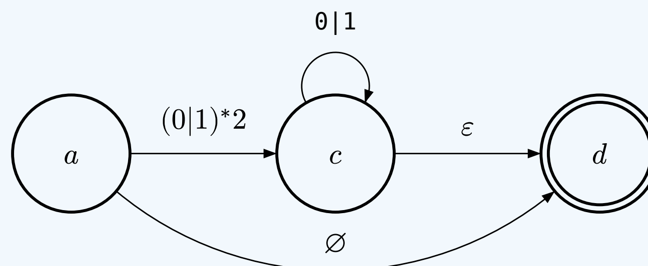
Per prima cosa generalizziamo l'automa:



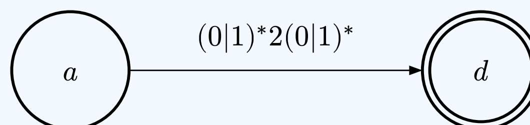
Abbiamo quindi:

- Rimosso archi entranti dallo stato iniziale
- Rimosso archi uscenti dallo stato finale
- Collegato ogni stato con ogni altro stato

Adesso iniziamo la rimozione degli stati, iniziamo rimuovendo b:



Rimuoviamo c:



Otteniamo quindi l'espressione regolare $(0|1)^*2(0|1)^*$

1.7. Pumping Lemma

Il *pumping lemma* ci serve per rispondere alla domanda "tutti i linguaggi sono regolari?". L'idea dietro il lemma è che tutte le stringhe di un linguaggio regolare hanno una sezione che può essere

ripetuta infinite volte e tutte le stringhe risultanti da questa ripetizione apparterranno ancora al linguaggio.

Teorema - Pumping Lemma

Se L è regolare allora $\exists p$ (*pumping length*) t.c. presa $w \in L$ con $|w| \geq p$ allora w può essere scomposta in $w = xyz$ in modo che:
1. $\forall i \geq 0, xy^i z \in L$
2. $|y| > 0$. *Altrimenti sarebbe banale infatti* $y^0 = \varepsilon$.
3. $|xy| \leq p$

Prendiamo come pumping length p il numero degli stati e mostriamo che qualsiasi stringa s di lunghezza maggiore o uguale a p può essere spezzata in xyz. Ovviamente se nessuna stringa è lunga almeno p allora il teorema risulta vero. Se $|s| = n$ allora possiamo dire che gli stati attraversati dall'automa saranno $n + 1$ e dato che $n \geq p$ allora $n + 1 > p$, quindi c'è almeno uno stato che si ripete.

Dividiamo s in xyz in questo modo:
- Individuiamo il primo stato che si ripete e lo chiamiamo q_r
- x è la parte di s che viene prima della **prima apparizione** di q_r.
- y è la parte prima della **prima ripetizione** di q_r.
- z è la parte rimanente della stringa.

Con questa divisione rispettiamo le 3 condizioni:
1. Visto che y ci riporta alla fine di x possiamo ripetere y quante volte vogliamo e quindi se l'automa accetta xyz accetterà anche $xy^i z$ con $i \geq 0$.
2. $|y| > 0$ per costruzione
3. Sappiamo che $|xy| \leq p$ perché y finisce prima della prima ripetizione.

Adesso che abbiamo visto come funziona il lemma "a parole", dimostriamolo in modo formale.

Dimostrazione:

- Sia $M = (Q, \Sigma, \delta, q_1, F)$ t.c. $L(M) = A$ e $p = |Q|$
- Sia $w = w_1 w_2 ... w_n \in A$ t.c. $n \geq p$
- Siano $r_1, ..., r_{n+1}$ gli stati attraversati da M durante il processamento di w (quindi la funzione fa i passaggi in modo che $\delta(r_i, w_i) = r_{i+1}$).

La stringa ha lunghezza $n \geq p$ quindi la sequenza di stati è lunga $n + 1 \geq p + 1$ e quindi uno stato appare almeno due volte, chiamiamo le occorrenze di questo stato r_p (la prima) e r_s (la seconda). Visto che r_s appare tra i primi $p + 1$ (infatti ho p stati ma ne attraverso $\geq p + 1$) stati si ha $s \leq p + 1$.

Adesso siano:
- $x = w_1, ..., w_{p-1}$
- $y = w_p, ..., w_{s-1}$
- $z = w_s, ..., w_n$

Otteniamo che:
1. Visto che x porta M da r_1 a r_p e y porta M da r_p a r_p, M accetterà $xy^i z$.
2. Sappiamo che $p \neq s$ quindi $|y| > 0$.
3. Sappiamo che $s \leq p + 1$ qundi $|xy| < p$.

Esercizio - Dimostriamo utilizzando il pumping lemma che $L = \{0^n 1^n\}$ non è regolare. Scegliamo $w = 0^p 1^p$ con p =pumping length. Abbiamo $|w| \geq p$. Dato che abbiamo $w = 0^p 1^p$ allora per qualsiasi scomposizione scegliamo $w = xya$ con $|xy| \leq p$ avremo sempre che y è composta da solo "0":

$$w = \underbrace{00..}_{x}...\underbrace{..00}_{y}01...1$$

Per $i \geq 2, \hat{w} = xy^i z$ è $0^q 1^p$ t.c. $q > p$ che non è in L. Quindi il pumping lemma è contraddetto $\Rightarrow L$ non è regolare. Infatti per qualsiasi numero di volte aumenti il numero di 0 non avrò mai lo stesso numero di 1.

Esercizio - Dimostriamo che il linguaggio $L = \{w \in \{0,1\}^* : \#_0 w = \#_1 w\}$ non è regolare. Prendiamo la stringa $w = 0^p 1^p \in L$ e analizziamo due casi:

1. $\underbrace{0...0}_{x}\ \underbrace{0...0}_{y}\ \underbrace{1...1}_{z}$
2. $\underbrace{0...0}_{x}\ \underbrace{0...0}_{y}\ \underbrace{001...1}_{z}$

Analizziamoli:

1. Nel primo caso abbiamo che:
   - $|y| = k > 0$ con $k \leq p$
   - $|x| = p - k$
   - $|z| = p$

   In questo modo però otteniamo che $|xy^2 z| = (p - k) + 2k + p = 2p + k$ ma allora $\#_0 = (p - k) + 2k = p + k > p$ e quindi $\hat{w} \notin L$.
2. Nel secondo invece:
   - $|y| = k > 0$
   - $|z| = p + l$. Dove p sono il numero di 1 e l il numero di 0.
   - $|x| = p - k - l$. Ovvero tutti gli 0 rimasti.

   Adesso però otteniamo che $|xy^2 z| = p - k - l + 2k + p + l = 2p + k$ ma sappiamo che $\#_0 = 2k + p - k - l + l = k + p > p$ e quindi $\hat{w} \notin L$.

Scelta della stringa

È importante scegliere la stringa "giusta" per effettuare la dimostrazione, infatti se ad esempio avessimo scelto $w = (01)^p \in L$ allora non saremmo riusciti a trovare una scomposizione che invalidasse il lemma.

1.8. Linguaggi Acontestuali

I linguaggi acontestuali sono "più potenti" dei linguaggi regolari, quest'ultimi infatti non sono in grado di avere una memoria per, ad esempio, ricordarsi quante occorrenze di un carattere sono state incontrate, questi nuovi linguaggi invece ci permettono di descrivere anche alcune strutture ricorsive.

Introduciamo il concetto di **Grammatica Acontestuale (CFG)**, queste sono formate da regole di sostituzione dette "produzioni" composte da **variabili** (di cui una speciale ovvero quella iniziale) e **terminali**. Vediamo un esempio:

1. $A \to 0A1$

2. $A \to B$
3. $B \to \#$

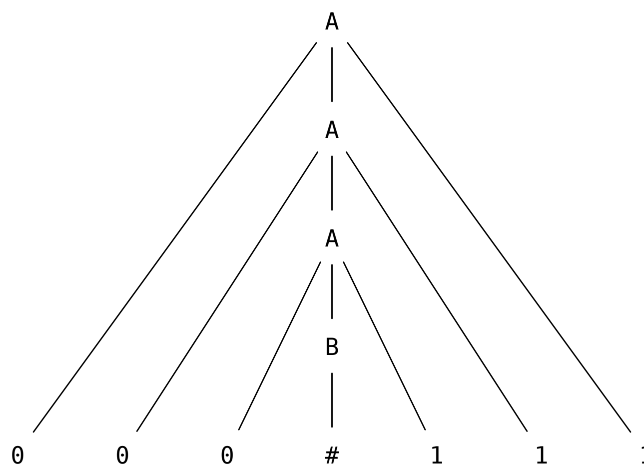
In questo caso le variabili sono A, B mentre i terminali $0, 1, \#$. Spesso i terminali li indichiamo anche con lettere minuscole.

Per generare stringhe con una grammatica acontestuale si usa il seguente procedimento:
1. Si scrive la variabile iniziale
2. Si usano le regole della grammatica per sostituire (a piacimento) le variabili con le espressioni che producono.
3. Si ripete lo step 2 fino a quando non ci sono più variabili.

Esempio - Con la grammatica di prima possiamo generare:

$$A \to 0A1 \to 00A11 \to 000A111 \to 000B111 \to 000\#111$$

Per le stringhe generata da una grammatica possiamo disegnare l'**albero di derivazione**:



Definizione - Grammatiche Acontestuali

Una CFG è una tupla (V, Σ, R, S) dove:
- V è un insieme finito di variabili.
- Σ è un insieme finito di terminali $(V \cap \Sigma = \varnothing)$
- R è un insieme finito di regole.
- S è la variabile iniziale.

Notazione

Due o più regole del tipo $A \to x, A \to y$, quindi con la stessa variabile che porta a più cose si possono unire per compattezza e scrivere quindi $A \to x \mid y$.

La terminologia che utilizzeremo invece è la seguente:
- Se $u, v, w \in \Sigma \cup V$ e $(A \to w) \in R$ diciamo che uAv **produce** uwv e lo scriviamo come $uAv \Rightarrow uwv$.
- Diciamo che u **deriva** w, scritto come $u \Rightarrow^* w$ se:
  - $u = w$, *oppure*
  - $u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_k \Rightarrow v$

Il **linguaggio di una grammatica** è quindi definito da:

$$\{w \in \Sigma \mid S \Rightarrow^* w\}$$

Esempio

Prendiamo la grammatica $G = (\{S\}, \{a, b\}, R, S)$ dove:
- $R = \{S \to aSb \mid SS \mid \varepsilon\}$

Allora, ad esempio, $aabb \in L(G)$ infatti:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

Esistono diversi modi per costruire una CFG, noi vedremo: **unione**, **trasformazione da DFA in CFG** e **ricorsione**.

1.8.1. Unione tra CFG

Supponiamo di avere una CFG $G_i = (V_i, \Sigma_i, R_i, S_i), \forall i \in [k]$ e vogliamo creare una CFG $G = (V, \Sigma, R, S)$ t.c. $L(G) = \bigcup_i L(G_i)$. Allora definiamo:
- $V = \bigsqcup_i V_i \cup \{S\}$. Assumendo quindi (wlog) che $V_i \cap V_j = \varnothing, \forall i \neq j$.
- S nuova variabile iniziale
- $\Sigma = \bigcup_i \Sigma_i$
- $R = \bigcup_i R_i \cup \{S \to S_1 \mid S_2 \mid \dots \mid S_k\}$

Dimostrazione - Dimostriamo che $\cup_i L(G_i) = L(G)$.

- La prima implicazione è $\cup_i L(G_i) \subseteq L(G)$. Sia $w \in \cup_i L(G_i)$ allora $\exists j \in [k]$ t.c. $w \in L(G_j)$ ovvero $S_j \Rightarrow^*_{G_j} w$, allora per costruzione di G abbiamo che $S \Rightarrow S_j \Rightarrow^*_{G_j} w$ ovvero $w \in L(G)$.

- La seconda implicazione è $L(G) \subseteq \cup_i L(G_i)$. Sia $w \in L(G)$ ovvero $S \Rightarrow^*_G w$ allora per costruzione $\exists j \in [k]$ t.c. $S \Rightarrow S_j \Rightarrow^*_{G_j} w$. Siccome V_j è disgiunto da tutti gli altri V_i possiamo dire che:

$$S \Rightarrow \underbrace{S_j \Rightarrow^*_{G_j} w}_{w \in L(G_j) \subseteq \cup_i L(G_i)}$$

1.8.2. Trasformare un DFA in un CFG

Dato un DFA $D = (Q, \Sigma, \delta, q_0, F)$ voglio definire $G = (V, \Sigma, R, S)$ t.c. $L(G) = L(D)$, costruiamola nel seguente modo:
- $V = \{V_q \mid q \in Q\}$
- $S = V_{q_0}$
- $\forall q \in F$ aggiungo la regola $V_q \to \varepsilon$
- $\forall p, q \in Q$ t.c. $\delta(q, a) = p$ aggiungo la regola $V_q \to aV_p$

Dimostrazione - Dobbiamo dimostrare che $L(D) = L(G)$ quindi vediamo le due implicazioni.

1. $L(D) \subseteq L(G)$. Sia $w = w_1 w_2 \dots w_k \in L(D)$ allora esiste una sequenza di stati

$$\underbrace{q_0, q_1, \dots, q_k, q_{k+1}}_{S} \text{ con } q_{k+1} \in F$$

   tali che $\delta(q_i, w_i) = q_{i+1}, \forall i < k$. Per costruzione possiamo dire che:

   - $\forall q_1, q_2 \in S$ t.c. $\delta(q_1, a) = q_2, \exists V_{q_1} \to aV_{q_2} \in R$.
   - $q_{k+1} \in F$, quindi $\left(V_{q_{k+1}} \to \varepsilon\right) \in R$

   La stringa viene quindi generata dalla CFG G.

2. $L(G) \subseteq L(D)$. Sia $w \in L(G)$ allora esiste una derivazione $S \Rightarrow^* w$ ma le produzioni in G sono tutte del tipo ($A \to aB$ o $A \to \varepsilon$) quindi la derivazione avrà forma

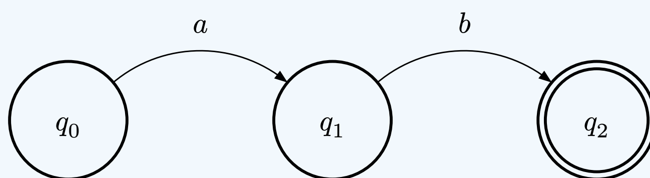
$$V_{q_0} \Rightarrow w_1 V_{q_1} \Rightarrow \ldots \Rightarrow w_1 \ldots w_k V_{q_k} \Rightarrow w_1 \ldots w_k$$

Ogni passo della derivazione è possibile per la regola $\left(V_{q_i} \to w_{i+1} V_{q_{i+1}}\right) \in R$ e per costruzione abbiamo che:
- $\delta(q_i, w_{i+1}) = q_{i+1} \in \delta$
- $\left(V_{q_k} \to \varepsilon\right) \in R \Leftrightarrow q_k \in F$ù

La stringa viene quindi riconosciuta correttamente dall'automa.

Facciamo un esempio.

Esempio



Costruiamo la grammatica G con:
- $V = \left\{V_{q_0}, V_{q_1}, V_{q_2}\right\}$
- $\Sigma = \{a, b\}$
- $R = \left\{V_{q_0} \to aV_{q_1}, V_{q_1} \to bV_{q_2}, V_{q_2} \to \varepsilon\right\}$
- $S = V_{q_0}$

Prendiamo la stringa $w = ab \in L(D)$ e vediamo che possiamo infatti generarla con la grammatica:

$$S \Rightarrow aV_{q_1} \Rightarrow abV_{q_2} \Rightarrow ab$$

1.8.3. Forma Normale di Chomsky

Definizione - Forma Normale di Chomsky

Una CFG è in forma normale se ogni regola è del tipo:
- $A \to BC$
- $A \to a$
- $S \to \varepsilon$

Con $A, B, C \in V, a \in \Sigma, B, C \neq S$

Teorema

Ogni CFG ammette una CFG equivalente in forma normale.

Dimostrazione - Facciamo una "dimostrazione" tramite un esempio (?), prendiamo la grammatica con le seguenti regole:

$$R = \{S \to ASA \mid aB; A \to B \mid S; B \to b \mid \varepsilon\}$$

1. Si aggiunge S_0 variabile iniziale insieme alla regola $S_0 \to S$:

$$R = \begin{cases} S & \rightarrow ASA \mid aB \\ A & \rightarrow B \mid S \\ B & \rightarrow b \mid \varepsilon \\ S_0 & \rightarrow S \end{cases}$$

2. Si eliminano le ε-regole del tipo $A \rightarrow \varepsilon$, quindi per ogni occorrenza di A a destra di una regola si aggiunge una nuova regola in cui $A = \varepsilon$:

$$\text{Elimino } B \rightarrow \varepsilon = \begin{cases} S & \rightarrow ASA \mid aB \mid a \\ A & \rightarrow B \mid S \mid \varepsilon \\ B & \rightarrow b \cancel{\mid \varepsilon} \\ S_0 & \rightarrow S \end{cases} \quad \text{Elimino } A \rightarrow \varepsilon = \begin{cases} S & \rightarrow ASA \mid aB \mid a \mid SA \mid AS \mid S \\ A & \rightarrow B \mid S \cancel{\mid \varepsilon} \\ B & \rightarrow b \\ S_0 & \rightarrow S \end{cases}$$

3. Si eliminano le regole unitarie $A \rightarrow B$ e ogni regola $B \rightarrow x$ con $x \in (V \cup \Sigma)$ viene sostituita da $A \rightarrow x$

$$\text{Elimino } S_0 \rightarrow S \text{ e } S \rightarrow S = \begin{cases} S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \cancel{\mid S} \\ A \rightarrow B \mid S \\ B \rightarrow b \\ S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS \\ \cancel{S_0 \rightarrow S} \end{cases}$$

$$\text{Elimino } A \rightarrow B = \begin{cases} S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \\ A \rightarrow \cancel{B} \mid S \mid b \\ B \rightarrow b \\ S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS \end{cases}$$

$$\text{Elimino } A \rightarrow S = \begin{cases} S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \\ A \rightarrow \cancel{S} \mid b \mid ASA \mid aB \mid a \mid SA \mid AS \\ B \rightarrow b \\ S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS \end{cases}$$

4. Per ogni regola $A \rightarrow x_1 ... x_k$ con $k \geq 3$ e $x_i \in (V \cup \Sigma)$ si elimina la regola iniziale e si aggiungono le regole:

$$\begin{cases} A & \rightarrow x_1 A_1 \\ A_1 & \rightarrow x_2 A_2 \\ \vdots \\ A_{k-2} & \rightarrow x_{k-1} x_k \end{cases}$$

$$\text{Otteniamo quindi} = \begin{cases} S \rightarrow \cancel{ASA} \mid aB \mid a \mid SA \mid AS \mid AA_1 \\ A \rightarrow b \mid \cancel{ASA} \mid aB \mid a \mid SA \mid AS \mid AA_1 \\ B \rightarrow b \\ S_0 \rightarrow \cancel{ASA} \mid aB \mid a \mid SA \mid AS \mid AA_1 \\ A_1 \rightarrow SA \end{cases}$$

5. Per ogni regola della forma $A \rightarrow x_1 x_2$, se $x_1 \in \Sigma$ si aggiunge una variabile X_1 ed una regola $X_1 \rightarrow x_1$. Si sostituisce $A \rightarrow x_1 x_2$ con $A \rightarrow X_1 x_2$. Facciamo lo stesso anche se $x_2 \in \Sigma$:

$$
\begin{cases}
S \to \cancel{aB} \mid a \mid SA \mid AS \mid AA_1 \mid XB \\
A \to b \mid \cancel{aB} \mid a \mid SA \mid AS \mid AA_1 \mid XB \\
B \to b \\
S_0 \to \cancel{aB} \mid a \mid SA \mid AS \mid AA_1 \mid XB \\
A_1 \to SA \\
X \to a
\end{cases}
$$

Abbiamo quindi ottenuto una grammatica equivalente in forma normale di Chomsky:

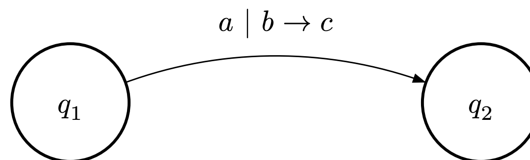
$$
\begin{cases}
S \to a \mid SA \mid AS \mid AA_1 \mid XB \\
A \to b \mid a \mid SA \mid AS \mid AA_1 \mid XB \\
B \to b \\
S_0 \to \mid a \mid SA \mid AS \mid AA_1 \mid XB \\
A_1 \to SA \\
X \to a
\end{cases}
$$

1.9. Pushdown Automata - PDA

I PDA sono degli automi in grado di riconoscere i linguaggi delle CFG, questi sono simili agli NFA ma hanno una pila (stack) LIFO. Ad ogni passo di computazione il PDA può svolgere 3 operazioni sulla cima della pila:

- PUSH (inserimento)
- POP (rimozione)
- Sostituzione (POP e poi PUSH in un unico passaggio)

Vediamo un esempio grafico per definire la notazione:



Questo significa che se ci troviamo nello stato q_1 possiamo percorrere l'arco e andare in q_2 soltanto se:

- Leggiamo a
- In cima alla pila abbiamo b
- Inoltre, Per effettuare l'operazione dobbiamo sostituire b con c.

Per PUSH e POP invece scriviamo:

- $\varepsilon \to x$: PUSH di x sulla cima
- $x \to \varepsilon$: POP di x dalla cima

Definizione - PDA

Un PDA è una tupla $(Q, \Sigma, \Gamma, \delta, q_0, F)$ dove:

- Q, Σ, q_0, F sono come gli NFA
- Γ è l'alfabeto della pila
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q_\varepsilon \times \Gamma_\varepsilon)$

I PDA non deterministici e deterministici sono **equivalenti**!

Vediamo come funzionano le transizioni, dato:

$$(q, c) \in \delta(p, a, b)$$
$$\text{con } p, q \in Q; a \in \Sigma_\varepsilon; b, c \in \Gamma_\varepsilon$$

Abbiamo appunto che p, q indicano gli stati, rispettivamente il precedente e il successivo. a è il carattere letto in input mentre b, c sono i caratteri in cima alla pila, rispettivamente il precedente e il successivo. Abbiamo quindi che:

- Se $a, b, c \neq \varepsilon$ l'automa legge a ed effettua la sostituzione $b \to c$.
- Se $b = \varepsilon$ l'automa legge a ed effettua il PUSH $\varepsilon \to c$.
- Se $b \neq \varepsilon$ e $c = \varepsilon$ l'automa legge a ed effettua il POP $b \to \varepsilon$
- Se $b = c = \varepsilon$ l'automa legge a senza modificare la pila.

Quand'è che un PDA **accetta**? Un PDA accetta una stringa $w = w_1, ..., w_n$ t.c. $w_i \in \Sigma$ se $\exists r_0, ..., r_m \in Q$ e $s_1, ..., s_m \in \Gamma^*$ t.c.:

- $r_0 = q_o$ (inizia dallo stato iniziale)
- $s_0 = \varepsilon$ (stack vuoto all'inizio)
- $r_m \in F$ (termina in uno stato accettante)
- $\forall i \in [m]$ si ha che:
  - $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$
  - $s_i = at$
  - $s_{i+1} = bt$

Con $a, b \in \Gamma_\varepsilon; t \in \Gamma^*$ stringa della pila.

Accettazione - Da notare che un PDA accetta indipendentemente dal contenuto della pila ma possiamo assumere che la pila debba essere vuota, ad esempio aggiungendo un ε-arco che la svuoti.

Per quanto riguarda le relazioni $\vdash_M$ e $\vdash_M^*$ invece, funzionano come per gli NFA:

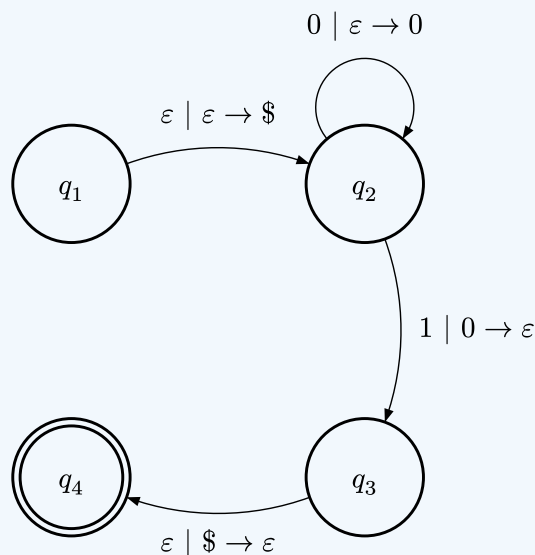
$$L(M) = \{w \in \Sigma^* \mid (q_o, w, \varepsilon) \vdash_M^* (q, \varepsilon, y), q \in F, y \in \Gamma^*\}$$

Come esempio vediamo il precedente linguaggio che abbiamo dimostrato non essere regolare, vediamo quindi come invece un PDA è in grado di riconoscerlo.

Esempi

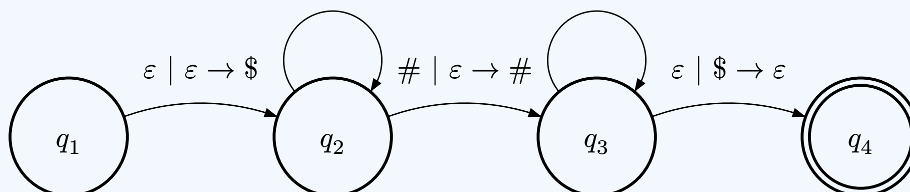
Costruiamo un PDA per il lingaggio $L = \{0^n 1^n \mid n \geq 0\}$. L'idea è quella di fare PUSH di 0 quando leggiamo 0 e POP di 0 quando leggiamo 1, se la pila è vuota a fine computazione allora accetta. Inseriamo inoltre un "segnalibro" \$ per indicare l'inizio della stringa e riconoscere quindi che è vuota. Abbiamo quindi:

- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, \$\}$
- $F = q_4$
- $q_0 = q_1$



Costruiamo un PDA per il lingaggio $L = \{w\#w^R \mid w \in \{0,1\}^*\}$. Dobbiamo quindi riconoscere le stringhe composte da una stringa w seguite dal carattere $\#$ e poi la stringa w al contrario, ad esempio *mela#alem*. L'idea è quella di scrivere w nella pila fino $\#$, controlliamo i restanti caratteri con quello che abbiamo nella pila:

$\cdot\ Q = \{q_1, q_2, q_3, q_4\}\ \cdot\ \Sigma = \{0, 1, \#\}\ \cdot\ \Gamma = \{0, 1, \$\}\ \cdot\ F = \{q_4\}\ \cdot\ q_0 = q_1$



Teorema

Un linguaggio è acontestuale **se e solo se** esiste un PDA M che lo riconosce.

Dimostrazione - Per fare questa dimostrazione dividiamo in due **lemmi**.

Lemma - Se un linguaggio è acontestuale allora esiste un PDA M che lo riconosce.

Dimostrazione - Per farlo pensiamo al fatto che se A è un linguaggio acontestuale (CFL) allora esiste una CFG G che lo genera, possiamo convertire questa CFG in un PDA:

- Ad ogni step di derivazione, il non determinismo del PDA P seleziona tutte le regole per una variabile in modo parallelo.
- P inizia scrivendo la variabile iniziale sullo stack e accetta se arriva a una stringa composta solo da terminali e che corrisponde a quella in input.
- Per ogni stringa intermedia il PDA salva solo dalla prima variabile in poi andando a fare un match carattere per carattere dei terminali.

Quello che fa P è quindi:

1. Inserire \$ e la variabile iniziale in cima allo stack.
2. Ripete i seguenti passaggi:
   1. Se la cima dello stack è una variabile A la sostituisce in modo non deterministico percorrendo tutte le regole in modo parallelo.
   2. Se la cima è un terminale a legge il prossimo simbolo dell'input e lo confronta con a, se combaciano va avanti altrimenti rifiuta quel ramo di computazione.
   3. Se la cima é \$ va allo stato di accettazione.

Iniziamo la **dimostrazione formale**, per farlo però introduciamo una "scorciatoia" ovvero che in un passo di computazione possiamo scrivere sullo stack un'intera stringa invece che un solo simbolo:

$$(r, u) \in \delta(q, a, s)$$

Ovvero P legge a dallo stato q, fa pop di s e scrive la stringa u. Questo è possibile perché equivale ad aggiungere gli stati $q_1, ..., q_{l-1}$ con $l = |u|$ t.c.:

- $\delta(q, a, s) = (q_1, u_l)$
- $\delta(q_1, \varepsilon, \varepsilon) = (q_2, u_{l-1})$
- $\vdots$
- $\delta(q_l, \varepsilon, \varepsilon) = (r, u_1)$

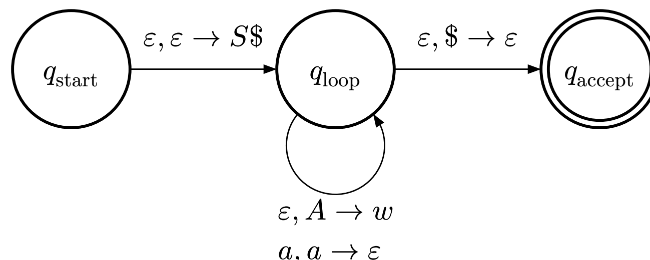
Dato un PDA $P = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, F)$ con:

- $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\} \cup E$. Dove E sono gli stati aggiunti grazie a al "trucco" di prima
- δ :
  - Creo il setup iniziale $\delta(q_{\text{start}}, \varepsilon, \varepsilon) = \{(q_{\text{loop}}, S\$)\}$

  Poi definiamo 3 regole in base a cosa abbiamo in cima allo stack:

  - **Variabile**: $\delta(q_{\text{loop}}, \varepsilon, A) \ni (q_{\text{loop}}, w)$ con $A \to w \in R$
  - **Terminale**: $\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \varepsilon)\}$
  - **\$**: $\delta(q_{\text{loop}}, \varepsilon, \$) = \{(q_{\text{accept}}, \varepsilon)\}$

Quindi, in generale, abbiamo quest situazione:



Esempio

Prendiamo la grammatica G con le regole $\{(S \to aTb \mid b), (T \to Ta \mid \varepsilon)\}$

Lemma - Se esiste un PDA P che riconosce L allora L è un linguaggio acontestuale.

Dimostrazione - Costruiamo una CFG che imita M. Assumiamo *wlog* che l'automa P:
- Ha un solo stato accettante q_{acc}
- Svuoti lo stack prima di accettare
- Abbia solo transizioni che fanno PUSH o POP non entrambe.

Per ogni coppia di stati (p, q) la CFG ha una variabile A_{pq} che genera tutte le stringhe che portano M da p a q senza cambiare l'altezza dello stack. Le regole della grammatica si basano sul comportamento della pila nel passare da p a q e ce ne sono di 3 tipi:
- **Stare Fermi**: $\forall p, A_{pp} \to \varepsilon$. Infatti per andare da p a p senza modificare la pia non dobbiamo leggere nulla.
- Se nell'andare da p a q l'automa passa per un punto r in cui la pila torna vuota allora il passaggio è la somma dei due passaggi. (Ovvero il simbolo pushato all'inizio non è quello che viene pop-ato alla fine).

$$A_{pq} \to A_{pr}A_{rq}$$

- Se il simbolo pop-ato alla fine, su input b, è quello pushato all'inizio su input a:

$$A_{pq} \to aA_{rs}b$$

Formalmente abbiamo che:

- $\forall p, q, r, s \in Q, u \in \Gamma, a, b \in \Sigma_\varepsilon$ se $(r, u) \in \delta(p, a, \varepsilon)$ e $(q, \varepsilon) \in \delta(s, b, u)$ allora mettiamo la regola $A_{pq} \to aA_{rs}b$ in G
- $\forall p, q, r \in Q$ mettiamo la regola $A_{pq} \to A_{pr}A_{rq}$ in G.
- $\forall p \in Q$ mettiamo $A_{pp} \to \varepsilon$ in G.

Claim - A_{pq} genera x **se e solo se** x porta P da p con stack vuoto a q con stack vuoto. Dobbiamo dimostrare entrambi i lati dell'implicazione.

Lato 1 (⇒) - Dimostriamolo per induzione sui k passi di derivazione di x da A_{pq}.

- **Caso base**: La derivazione ha 1 passo allora l'unica regola possibile è $A_{pp} \to \varepsilon$ che banalmente porta P da p a q con stack vuoto.
- **Passo Induttivo**: Supponendo che sia vero per derivazioni che hanno fino a k passi. Data $A_{pq} \Rightarrow^* x$ con $k + 1$ passi allora il primo passo può essere di 2 tipi:
  - $A_{pq} \Rightarrow aA_{rs}b$ e allora abbiamo $x = ayb$ con y che è stata generata da A_{rs} in k passi e quindi porta P da r a s con stack vuoto. Per costruzione applichiamo $A_{pq} \Rightarrow aA_{rs}b$ se $(r, u) \in \delta(p, a, \varepsilon)$ e $(q, \varepsilon) \in \delta(s, b, u)$
  - $A_{pq} \Rightarrow A_{pr}A_{rq}$ e abbiamo $x = yz$ con y generato da A_{pr} in massimo k passi e z generato da A_{rq} in massimo k passi. Quindi anche qui vale la proprietà di prima.

Lato 2 (⇐) - Dimostriamo per induzione sui k passi di computazione di P da p a q.

- **Caso Base**: Se facciamo 0 step allora la computazione inizia e finisce sullo stesso stato, P non può aver letto nulla per costruzione, allora G ha la regola $A_{pp} \to \varepsilon$.
- **Passo Induttivo**: Assumendo che sia vero per computazioni lunghe fino a k, ci sono 2 casi:
  - Lo stack è vuoto solo a inizio e fine computazione, allora il simbolo pushato a inizio computazione è lo stesso simbolo pop-ato a fine computazione. Chiamiamo u il simbolo, a l'input letto al primo passo, b l'input dell'ultimo passo, r lo stato dopo il primo passo e s il penultimo. Allora $(r, u) \in \delta(p, a, \varepsilon)$ e $(q, \varepsilon) \in \delta(s, b, u)$ e quindi la regola $A_{pq} \to aA_{rs}b \in G$.

    Possiamo scrivere $x = ayb$ dove y porta P da r a s senza toccare u in $k - 1$ passi, quindi $A_{rs} \Rightarrow^* y$ che comporta $\Rightarrow A_{pq} \Rightarrow^* x$
  - Lo stack si svuota durante la computazione. Sia r lo stato in cui lo stack si svuota allora le computazioni che portano P da p a r e da r a q hanno massimo k passi. Quindi scrivendo $x = yz$ abbiamo $A_{pr} \Rightarrow^* y$ e $A_{rs} \Rightarrow^* z$.

    Visto che la regola $A_{pq} \to A_{pr}A_{rq}$ è in G allora $A_{pq} \Rightarrow^* x$.

1.10. Pumping Lemma per i CFL

Dato un CFL L, esiste p t.c. $\forall w \in L$ con $|w| \geq p$ è possibile scomporre w in $w = uvxyz$ in modo che:

1. $\forall i \geq 0, uv^ixy^iz \in L$
2. $|vy| > 0$
3. $|vxy| \leq p$

Dimostrazione - *wlog* Assumiamo che G sia in forma normale di Chomsky, questo ci permette di dire che l'albero di derivazione di w sarà binario, infatti le uniche regole possibili sono $\{(A \to BC), (A \to a), (S \to \varepsilon)\}$.

Claim

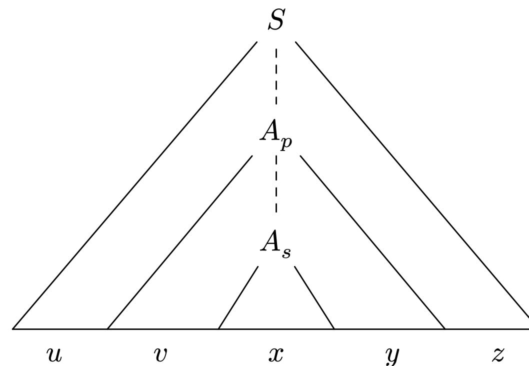
Se il cammino più lungo nell'albero ha lunghezza i, la stringa da esso generata ha lunghezza $\leq 2^{i-1}$. Come idea ci basti pensare al caso limite ovvero che ad ogni ramo una variabile si sdoppia in altre due ma ovviamente al livello finale dell'albero tutte le variabili diventano terminali ed è per questo che eleviamo con -1.

Dimostriamo per induzione:
- **Caso Base**: $i = 1$, l'albero è $S \to a$ e infatti $|a| = 1 = 2^{1-1} = 2^0 = 1$
- **Passo Induttivo**: Supponendo sia vero fino a $i - 1$, se abbiamo un albero di altezza $i > 1$ la radice S deve avere una regola $S \to AB$, i due sottoalberi radicati in A e B avranno lunghezza $\leq i - 1$.

Quindi per ipotesi induttiva A e B generano stringhe lunghe al massimo $2^{i-1-1} = 2^{i-2}$. Questo significa che S genera una stringa di lunghezza $\leq 2 \cdot 2^{i-2} = 2^{i-1}$.

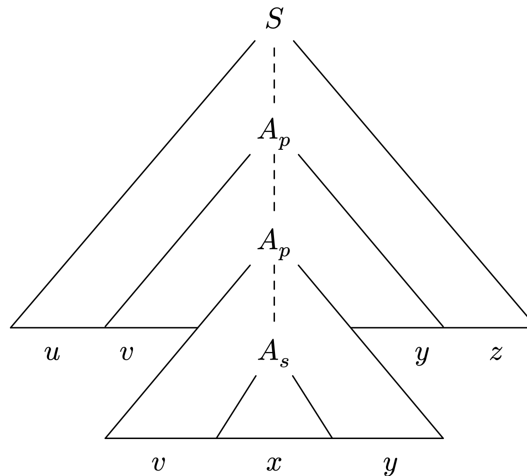
Dimostrazione - Sia $m = \#V$ in G poniamo $p = 2^m$. Segue che w t.c. $|w| \geq p$ è generata da un cammino lungo $\geq m + 1$. Il cammino ha quindi almeno $m + 2$ nodi compreso il terminale di cui almeno $m + 1$ variabili. Visto che $|V| = m$ allora almeno una variabile si deve ripetere.

Partendo dalle foglie dell'albero prendiamo la prima variabile che si ripete e chiamiamo A_s la prima occorrenza e A_p la successiva:



E la computazione si divide in $w = uvxyz$
- $S \Rightarrow^* uA_pz$
- $A_p \Rightarrow vA_sy$
- $A_s \Rightarrow x$

Ma sappiamo che $A_p = A_s$ e quindi è come se fosse $A_p \Rightarrow vA_py$ e possiamo quindi sostituire il suo sottoalbero e la derivazione rimarrà uguale:



In questo modo otteniamo la stringa uv^2xy^2z, e otteniamo quindi che:

1. $uv^ixy^iz \in A, \forall i \geq 0$
2. $|vy| > 0$ perché non ci sono ε-produzioni né produzioni unitarie quindi la derivazione $A_p \Rightarrow^* vA_sy$ deve aver usato una regola del tipo $A_p \to BC$ dove $B \Rightarrow^* vA_s$ e $C \Rightarrow^* y$ oppure $B \Rightarrow^* v$ e $C \Rightarrow^* A_sy$.
   Visto che non ci sono ε-regole, in entrambi i casi si ha che $y \neq \varepsilon$ o $v \neq \varepsilon$.
3. $|vxy| \leq p$ dato che A_p si trova tra le ultime $m+1$ variabili sappiamo che il sottoalbero di A_p ha altezza massimo $m+1$ quindi per il claim genera stringhe da lunghezza massima $2^{m+1-1} = 2^m = p$

2. Calcolabilità

Iniziamo introducendo il concetto di **Macchina di Turing**, queste corrispondono al modello astratto di computer, tuttio ciò che è calcolabile per un computer lo è anche per una TM.

Una Turing Machine possiede un nastro infinito e una testina con la quale può leggere e scrivere sul nastro. Inizializziamo il nastro scrivendoci sopra la stringa di input, i restanti slot sono riempiti con il carattere *blank* $\sqcup$. I movimenti possibili per la testina sono *dx, sx, lettura, scrittura* e continua a computare fino al raggiungimento lo stato di accettazione o rifiuto, se non raggiunge nessuno dei due allora va in loop.

Esempio

Costruiamo una TM per il linguaggio

$$\{w\#w : w \in \{0,1\}^*\}$$

La TM si muove a zig-zag tra i simboli a dx e sx di # e controlla se il simbolo a sx combacia con quello a dx, se si li cancella. Se trova un confronto non valido rifiuta altrimenti se sovrascrive tutti i simboli accetta.

Diamo adesso la definizione formale di TM.

Definizione - Turing Machine

Una TM è una tupla $M = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{acc}}, q_{\text{rej}})$ dove:

- Q è l'insieme finito degli stati
- Σ è l'insieme finito di input $\sqcup \notin \Sigma$
- Γ è l'alfabeto del nastro $(\sqcup \in \Gamma, \Sigma \subseteq \Gamma$
- q_{acc} è lo stato di accettazione
- $q_{\text{rej}} \neq q_{\text{acc}}$ è lo stato di rifiuto
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ definita come, se $\delta(p, a) = (q, b, X)$ allora:
  - a viene letto dal nastro e sostituito da b
  - la macchina passa dallo stato p allo stato q
  - Il nastro viene spostato a sx se $X = L$ e a dx se $X = R$

  La transizione avrà quindi come etichetta $a \to b; X$

Inoltre le transizione in realtà avvengono da $Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}$ dato che se la TM va in uno dei due termina subito.

Vediamo quindi adesso il concetto di **configurazione**. La configurazione di una TM si rappresenta in questo modo:

$$uqv$$

Dove:

- La posizione della testina è il primo simbolo di v
- q è lo stato corrente
- u è il restante contenuto del nastro precedente

Quindi ad esempio, una configurazione potrebbe essere:

$$1011q_7 01111$$

, inoltre la configurazione iniziale è $q_{\text{start}} w$ e quella finale $q \in \{q_{\text{acc}}, q_{\text{rej}}\}$.

Possiamo quindi dire che:

- $uaq_i bv \vdash uq_j acv \Leftrightarrow \delta(q_i, b) = (q_j, c, L)$
- $uaq_i bv \vdash uacq_j v \Leftrightarrow \delta(q_i, b) = (q_j, c, R)$

Diciamo che M **accetta** $w \in \Sigma^* \Leftrightarrow c_{\text{start}} = q_{\text{start}} w$ è t.c. $c_{\text{start}} \vdash^* c_{\text{acc}}$ dove c_{acc} è la configurazione di accettazione.

Definizione - Linguaggi Turing Riconoscibili

Un linguaggio L è Turing-Riconoscibile se $\exists$ TM M che lo riconosce, ovvero se $\forall w \in L, M$ accetta w.

Definizione - Decisore

Una TM è detta **decisore** se termina sempre, ovvero non va in loop.

Definizione - Decidibilità

Un linguaggio L è Turing-Decidibile se $\exists M$ decisore t.c. $L = L(M)$

Esistono delle varianti di TM che mantengono lo stesso potere espressivo:
- Nastro infinito a 2 direzioni
- TM che muovono a R, L, S - "stay-put". Infatti basta che ad ogni transizione fanno prima uno spostamento a sinistra e poi a destra.
- TM con k nastri
- TM non deterministiche

2.1. TM Multinastro

Formalmente abbiamo che $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$ con $k = \#$ nastri, abbiamo quindi

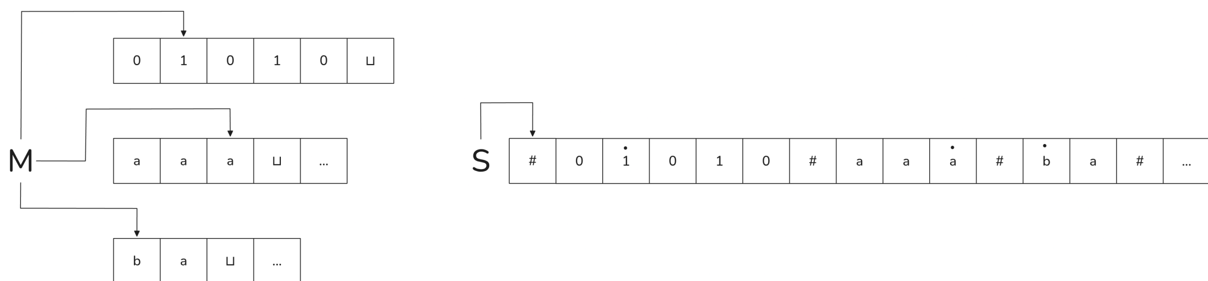
$$\delta(q_i, a_1, ..., a_k) = (q_j, b_1, b_k, L, R, ..., L)$$

Se la macchina è nello stato q_i e le k testine leggono i simboli $a_1, ..., a_k$ la macchina va nello stato q_j, scrive i simboli $b_1, ..., b_k$ e ogni testina si muove come specificato.

Teorema

Ogni TM multinastro ha una TM singolo-nastro equivalente.

Dimostrazione - S simula M mettendo il suo contenuto in un singolo nastro in questo modo:
- Scrive sequenzialmente i contenuti dei k nastri con $\backslash\#$ come delimitatore
- Per tenere traccia di dove si trovano le testine, introduce un simbolo ° sopra al carattere corrispondente, le chiamiamo "testine virtuali".



Quindi S su input $w = w_1, ..., w_n$ si comporta così:
1. Pone il nastro nella configurazione "iniziale" $\#\mathring{w}_1 w_2 ... w_n \# \mathring{\sqcup} \# ... \#$
2. S scansiona il nastro dal primo all'ultimo $\#$, per capire quali siano i simboli puntati dalle testine virtuali. Poi fa una seconda scansione per aggiornare il nastro secondo δ' di M.
3. Se in qualsiasi momento una testina finisce su un $\#$ significa che ha raggiunto un $'\sqcup'$ e quindi vi scrive $'\sqcup'$ e sposta di una posizione a destra l'intero contenuto del nastro.

Corollario

Un linguaggio è Turing-Riconoscibile ⇔ una TM multinastro lo riconosce.

Dimostrazione - Dato che la singolo-nastro è una caso speciale di multinastro e che ogni multinastro ha una singolo-nastro equivalente la dimostrazione è completa.

2.2. TM non Deterministica

Formalmente abbiamo che:

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Il concetto di non determinismo è uguale a quelllo che troviamo negli NFA. Semplicemente in questo caso quando la TM deve fare una mossa potrebbe avere più azioni possibili da svolgere e come per gli NFA, creiamo più percorsi indipendenti.

Simuliamo una NTM N con una TM D, per farlo consideriamo l'albero di computazione di N che esploriamo con una BFS. Non effettuiamo la ricerca con una DFS perché dato che vogliamo simulare il parallelismo se eseguissimo una DFS questa potrebbe continuare all'infinito su un ramo.

L'albero ha questa struttura:
- La radice è ε
- Ogni altro nodo ha un indirizzo $\in \Gamma_b = \{1, ..., b\}$ con b il massimo numero di scelte tra tutti gli stati di N, nello specifico l'indirizzo è formato da xy con x indirizzo del padre e $y \in \Gamma_b$ "numero" del figlio, quindi ad esempio 231 indica il secondo figlio della radice -> il suo terzo figlio -> il suo primo figlio.

La TM D che simula N è una multi-nastro con 3 nastri:
1. Il nastro 1 contiene l'input w e non viene mai alterato. Ci serve come riferimento per quando cambiamo ramificazione.
2. Il nastro 2 simula un ramo della computazione di N, quando ha più scelte effettua quella presente in 3.
3. Il nastro 3 contiene l'indirizzo del nodo fino a cui simulare, quindi conterrà ad esempio "1", "2", "3" poi passa ai figli del primo nodo quindi "11", "12", "13" e poi a quelli del primo nodo ancora "111", "112"...

La TM D si comporta quindi in questo modo:
1. Inizialmente il nastro 1 contiene w e gli altri 2 sono vuoti.
2. Copia il nastro 1 sul nastro 2
3. Usa il nastro 2 per simulare N con input w. Ad ogni step N consula il prossimo simbolo sul nastro 3
   - Se il nastro 3 è vuoto va allo step 4
   - Se la simulazione rifiuta va allo step 4
   - Se la simulazione accetta, accetta
4. Sostituisce la stringa sul nastro 3 con la successiva in ordine lessicografico. Va al passo 2.

Esempio

Una TM non deterministica può essere una TM che su input binario:
- input 0:
  - Si muove a dx
  - Si muove a sx
- input 1: accetta

Quindi quando legge uno 0 crea due rami, se simuliamo questa TM non deterministica con una deterministica avremo che eseguirà entrambe le scelte prima o poi.

Tesi di Church-Turing

Qualunque problema calcolabile da un algoritmo può essere risolto da una macchina di Turing.

2.3. Linguaggi Decidibili

Sono quei linguaggi per i quali esiste una TM che per ogni stringa termina sempre in un tempo finito e stabilisce se questa appartiene o no al linguaggio.

Problema dell'accettazione per i DFA - Definiamo

$$A_{\text{DFA}} = \{< B, w > \mid B \text{ è un DFA che accetta } w\}$$

E presentiamo una TM che accetta A_{DFA}:

M = "Su input $< B, w >$, con B DFA e w stringa:
- Simula B su input w
- Se la simulazione finisce con uno stato di accettazione, accetta; altrimenti rifiuta."

Problema dell'accettazione per gli NFA - Definiamo

$$A_{\text{NFA}} = \{< B, w > \mid B \text{ è un NFA che accetta } w\}$$

E presentiamo una TM che accetta A_{NFA}:

N = "Su input $< B, w >$, con B NFA e w stringa:
- Converti B in un DFA C
- Esegui la TM 'M' (per A_{DFA}) su C

-Se M accetta, accetta; altrimenti rifiuta."

Determinare se una regex genera una stringa - Definiamo

$$A_{\text{REX}} = \{< R, w > \mid R \text{ regex che genera } w\}$$

E costruiamo una TM che accetta A_{REX}:

P = "Su input $< R, w >$ con R regex e w stringa:
- Converti R in NFA A
- Esegui la TM N (per A_{NFA} su A)
- Se N accetta, accetta; altrimenti rifiuta"

Emptiness-testing per i DFA - Definiamo

$$E_{\text{DFA}} = \{< A > \mid A \text{ DFA e } \; L(A) = \varnothing\}$$

E costruiamo una TM che accetta E_{DFA}:

T = "Su input $< A >$ con A DFA:
- Marca lo stato iniziale
- Finchè non vengono marcati nuovi stati, marca ogni stato che ha una transizione entrante da uno stato già marcato.
- Se almeno uno stato accettante è stato marcato, rifiuta; altrimenti accetta."

Determinare se due DFA accettano lo stesso linguaggio - Definiamo

$$EQ_{\text{DFA}} = \{< A, B > \mid A \text{ e } B \text{ DFA e } L(A) = L(B)\}$$

Utilizziamo la differenza simmetrica:

$$L(C) = L(A) \bigtriangleup L(B)$$

, se è vuota abbiamo che $L(A) = L(B)$. Si ha

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$$

Dobbiamo verificare se $< C > \in E_{\text{DFA}}$, costruiamo

F = "Dato un input:

- Costruisci il DFA C come descritto
- Simula la TM T (per E_{DFA})
- Se T accetta, accetta; altrimenti rifiuta“

Determinare se una CFG genera una stringa - Definiamo

$$A_{\text{CFG}} = \{< G, w > \mid G \text{ CFG genera } w\}$$

Costruiamo la TM

M = “Su input $< G, w >$ con G CFG e w stringa:
- Converti G in CNF (Chomsky)
- Enumera le derivazioni con $2n - 1$ passi dove $n = |w|$, se $n = 0$ enumera solo quelle con 1 passo. Da questa possiamo ricavare il successivo *claim*
- Se una delle derivazioni genera w, accetta; altrimenti rifiuta.“

Claim - Sia $G = (V, \Sigma, R, S)$ una CFG in CNF. Data $w \in G$ con $|w| = n$, la sua derivazione è di $2n - 1$ passi.

Dimostrazione - Dimostriamolo per induzione su n.
- Caso Base: $n = 1$ allora $|w| = 1$ significa che $w \in \Sigma$. Quindi w è derivata con la regola $S \to w$ in $1 = 2 \cdot 1 - 1$ passi.
- Passo Induttivo: Assumiamo sia vero per stringhe di lunghezza $\leq n - 1$. Sia $w \in G$ t.c. $|w| = n$. Visto che G è in CNF, $S \overset{*}{\Rightarrow} w$ implica che $S \to AB \overset{*}{\Rightarrow} w$. Quindi possiamo scrivere $w = xy$ con $A \overset{*}{\Rightarrow} x$ e $B \overset{*}{\Rightarrow} y$. Visto che non ci sono ε-regole in CNF abbiamo che $x, y \neq \varepsilon$. Quindi siano $|x| = k$ e $|y| = n - k$, per ipotesi induttiva x, y sono derivabili in $2k - 1$ e $2(n - k) - 1 = 2n - 2k - 1$ produzioni. Quindi $AB \overset{*}{\Rightarrow} w$ ha $2k - 1 + 2n - 2k - 1 = 2n - 2$ produzioni, aggiungendo $S \to AB$ otteniamo che $S \overset{*}{\Rightarrow} w$ in $2n - 1$ produzioni.

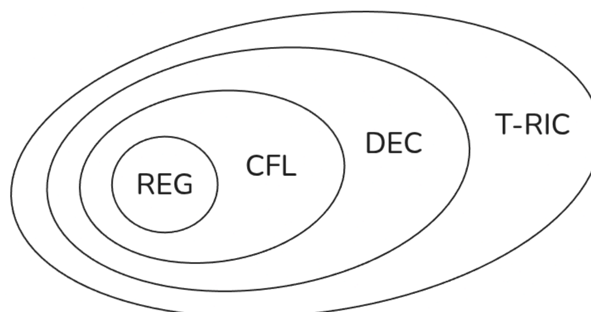
Emptiness-testing per CFG - Definiamo

$$E_{\text{CFG}} = \{< G > \mid G \text{ CFG e } L(G) = \varnothing\}$$

Costruiamo la TM

M = “Su input G, dove G è una CFG:
- Marca tutti i terminali in Σ
- Finchè non vengono marcate nuove variabili:
  - Marca ogni $A \in V$ per cui $\exists A \to U_1 U_2 ... U_l \in R$ dove ogni U_i è già stato marcato
- Se S è marcata, rifiuta; altrimenti accetta“.

Otteniamo quindi questa situazione per i linguaggi:



Esistono però linguaggi non TURING-DEC / RIC, introduciamo quindi il problema A_{TM} che determina se una TM accetta un determinato input.

$$A_{\text{TM}} = \{< M, w > \mid M \text{ TM e } w \in L(M)\}$$

Definizione - Turing Machine Universale

Una TM universale è una TM in grado di simulare ogni altra TM. Si dice infatti che un modello di calcolo è Turing-complete se è equivalente ad una TM universale.

Una qualsiasi TM M può essere completamente descritta da un insieme finito di transizioni, e quindi può essere codificata come una stringa finita di simboli. Una TM universale prende in input la descrizione codificata della TM da simulare $< M >$ e l'input w su cui dovrebbe lavorare e simula M.

Notiamo che A_{TM} è *T-REC*, infatti abbiamo la TM U che lo riconosce:

U = "Su input $< M, w >$ con M TM e w stringa:
- Simula M su w
- Se M raggiunge uno stato accettante, accetta; altrimenti rifiuta"

Con questa costruzione se M va in loop ci andrà anche U.

Teorema

A_{TM} non è decidibile.

Ma per dimostrarlo abbiamo prima bisogno di un altro teorema.

Teorema

Esistono linguaggi non Turing-riconoscibili.

Dimostrazione - Notiamo che l'insieme di tutte le TM $\mathcal{M}$ non è numerabile in quanto l'insieme di tutte le stringhe Σ^* su un alfabeto Σ è numerabile, queste rappresentano gli encoding di ogni TM in stringe $< M >$.

Notiamo anche che l'insieme delle stringhe binarie infinite $\mathcal{B}$ non è numerabile. Adesso consideriamo $\mathcal{L} = \mathcal{P}(\Sigma^*)$ l'insieme di tutti i linguaggi definiti su Σ. Mostriamolo fornendo una corrispondenza biunivoca con $\mathcal{B}$, quindi ogni linguaggio $A \in \mathcal{L}$ corrisponde ad una sequenza univoca in $\mathcal{B}$.

Definiamo il concetto **Sequenza Caratteristica** di A, ovvero una sequenza binaria definita come segue:

$$\mathcal{X}_A = b_1 b_2 \ldots \text{ t.c. } b_i = \begin{cases} 1 & s_i \in A \\ 0 & s_i \notin A \end{cases}$$

Esempio

Prendiamo il linguaggio A = "Stringhe binarie che iniziano con 0":
- $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, \ldots\}$
- $A = \{\ , 0,\ , 00, 01,\ ,\ , \ldots\}$
- $\mathcal{X}_A = 0101100\ldots$

Notiamo che $f : \mathcal{L} \to \mathcal{B} : A \to \mathcal{X}_A$ è biunivoca, perciò visto che $\mathcal{B}$ non è numerabile non lo è neanche $\mathcal{L}$. Ovvero $\exists A \in \mathcal{L}$ t.c. $\nexists M$ TM t.c. $L(M) = A$.

Quindi la funzione $h : \mathcal{M} \to \mathcal{L} : M \to L(M)$ non è biiettiva, quindi $\exists L \in \mathcal{L}$ t.c. $\nexists M$ per cui $L(M) = L$.

Adesso possiamo tornare a dimostrare che A_{TM} non è decidibile.

Dimostrazione - Supponiamo per assurdo che $A_{\text{TM}} \in \text{DEC}$ e sia H un decisore per A_{TM}:

$$H(< M, w >) = \begin{cases} \text{accept} & \text{se } M \text{ accetta } w \\ \text{reject} & \text{se } M \text{ non accetta } w \end{cases}$$

Adesso costruiamo una TM

D = "Su input $< M >$ con M TM:
1. Esegui H con input $< M, < M \gg$
2. Dai in output l'opposto dell'output di H"

$$D(< M >) = \begin{cases} \text{accetta} & \text{se M non accetta <M>} \\ \text{rifiuta} & \text{se M accetta <M>} \end{cases}$$

Ma adesso cosa succede se eseguiamo D con la sua stessa descrizione $< D >$ in input?

$$D(< D >) = \begin{cases} \text{accetta} & \text{se D non accetta <D>} \\ \text{rifiuta} & \text{se D accetta <D>} \end{cases}$$

Ma abbiamo ottenuto una contraddizione, se D accetta $< D >$ allora D deve rifiutare $< D >$ e viceversa.

Definizione - Coturing-Riconoscibile

Un linguaggio si dicec *Co-Turing-Riconoscibile* se è il complemento di un linguaggio *Turing-Riconoscibile*

Teorema

Un linguaggio L è decidibile $\Leftrightarrow L \in \text{REC}$ e $L \in \text{CO-REC}$

Dimostrazione
- $\Rightarrow$ se L è decidibile, $\overline{L}$ è decidibile quindi L e $\overline{L}$ sono *Turing-Riconoscibili.*
- $\Leftarrow$ sia M_1 un riconoscitore per A e M_2 un riconoscitore per $\overline{A}$

Costruiamo M = "Su input w:
1. Esegui M_1 e M_2 in parallelo su w
2. Se M_1 accetta, accetta; se M_2 accetta, rifiuta"

Ogni stringa w è in A oppure in $\overline{A}$, quindi uno tra M_1 e M_2 deve accettare. Visto che M si ferma se M_1 o M_2 accettano, termina sempre e quindi è un decisore. Inoltre, accetta le stringhe in A e rifiuta quelle in $\overline{A}$ quindi è un decisore per A.

Corollario 1

Se L non è decidibile allora almeno uno tra L e $\overline{L}$ non è *Turing-Riconoscibile.*

Corollario 2

$\overline{A_{\text{TM}}}$ non è *Turing-Riconoscibile.*

2.4. Riducibilità

Vediamo una tecnica per dimostrare che alcuni problemi non sono decidibili o riconoscibili. Una **riduzione** è un modo di convertire un problema in un altro problema in modo che una soluzione per il secondo problema possa essere usata per il primo. *(Ci serve aumentare il modello delle TM aggiungendo un nastro di output WLOG)*

Diremo che una TM calcola una funzione su un dato input se inizia con w sul nastro di input e termina con $f(w)$ sul nastro di output.

Definizione - Calcolabilità

Una funzione $f : \Sigma^* \to \Sigma^*$ è calcolabile se $\exists$TM M che calcola f $\forall w \in \Sigma^*$