**Indice**

**CONCETTI DI BASE**

[Pag. 01] ;

[Pag. 01] Grammatiche Lineari Destre – Precisazioni

[Pag. 02] Grammatiche Ambigue, Monotone, Linguaggi Finiti, Tipi di crescita di L (Lineare, Esponenziale, Senza Pattern)

[Pag. 02] a+b nelle Espressioni Regolari (Inteso come OR); (a+b)\* nelle Espressioni Regolari (Inteso come ∑\*)

[Pag. 03] Stati Finali Automa; Stato Pozza Automa [Serve per il Complemento di un Automa]; Complemento Automa

**CONCETTI AVANZATI**

[Pag. 06] Pumping Lemma CF – caso con 2 terminali

[Pag. 07] Pumping Lemma CF – caso con 3 terminali

[Pag. 10] Pumping Lemma Regolare/Lineare Destro [Metodo Semeraro, Automi; Metodo Fanizzi]

[Pag. 11] Conversione di un Automa NDA --> DFA

[Pag. 13] Proprietà di Chiusura (Tabella Semeraro)

[Pag. 14] Unione e concatenazione grammatiche di tipo 2 (Context free)

[Pag. 14] Grammatiche tipo 2 particolari – (GNF [Greibach Normal Form], NLR, CNF)

[Pag. 15] Intersezione Linguaggi tipo 3 e Intersezione Automi [Leggi di De Morgan]

[Pag. 15] Unione grammatiche di tipo 3 (lineari destre)

[Pag. 15] Automa -> Grammatica tipo 3

[Pag. 15] Unione Linguaggi di tipo 3 (unendo automi)

[Pag. 15] Unione Automi – Metodo 1 (tramite Unione Grammatiche tipo 3) + Esercizio svolto

[Nel caso speciale in cui non posso unirli a occhio, perché un automa itera su q0 o qualcosa torna indietro su q0, o casi simili]

[Pag. 18] Unione Automi – Metodo 2 (Unirli in un ε-NDA) + Conversione da ε-NDA a DFA + 2 Esercizi svolti

[Pag. 22] Da Automa a Espressione Regolare – Metodo 1 (Usato da JFLAP)

[Eliminazione stati intermedi, alternativo a (R\* + SU\*T)\* SU\*]

[Pag. 25] Da Automa a Espressione Regolare – Metodo 2: (R\* + SU\*T)\* SU\*

[Messo da Fanizzi nelle slide, non mi ci sono trovato bene; alternativo a JFLAP]

[Pag. 26] Metodi per Abbreviare le Espressioni Regolari (Accorpare casi simili con Epsilon)

[Pag. 26] Da Espressione Regolare ad Automa Epsilon-NDA (Metodo JFLAP)

[Opzionale, non serve per l’ esame, è per curiosità]

**CASI PARTICOLARI**

[Pag. 28] Automa numeri Divisibili (o Non divisibili) Per n; Automa stringhe che Non terminano con abc

**(Iterazione di un Linguaggio)**

Le infinite combinazioni di parole (inclusa la stringa vuota) ottenibili combinando le parole di L

Esempio: ∑ = {a, b} L = {aba, bb}

∑\* = { ε, a, aa, aaa, […], b, bb, bbb, […], ab, abb, […], ba, baa, […], ababaa, […], babbaaaabaab, […] }

L\* = { ε, aba, abaaba, […], bb, bbbb, […], ababb, ababbbb, […], bbaba, bbabaaba, […] }

QUINDI

aab ∑\*

aab L\*

**(Potenza di un Linguaggio)**

, è un caso speciale di concatenamento

Si prendono due parole qualsiasi, anche diverse fra loro, non è la stessa parola ripetuta due volte

ATTENZIONE:

NON è ,

,

la prima sequenza di a, b, può avere lunghezza diversa dalla seconda, non è l’ insieme di concatenazioni della stessa parola ripetuta due volte

**GRAMMATICHE LINEARI DESTRE (TIPO 3) – PRECISAZIONI**

Due sole forme di produzione accettate:

1) , con b ≠ stringa vuota

2) , con b che può essere stringa vuota ε

OVVERO

1) L’ NT produce uno e un solo terminale (diverso da stringa vuota), seguito da uno e un solo NT a destra

(da ciò il nome della grammatica, perché essendo sempre a destra l’ NT, la stringa cresce verso destra, e producendo ogni NT un solo terminale, cresce in maniera lineare)

2) L’ NT produce uno e un solo terminale (in tal caso può anche essere stringa vuota)

QUINDI

NON è lineare destra (produce due terminali)

NON è lineare destra (non produce nessun terminale)

NON è lineare destra (produce due NT)

NON è lineare destra (il NT è a sinistra)

NON è lineare destra (produce due terminali)

**GRAMMATICHE AMBIGUE, MONOTONE, LINGUAGGI FINITI, TIPI DI CRESCITA DI L**

1) Grammatica Ambigua: si può arrivare a produrre una stessa stringa w applicando diverse produzioni o in vari ordini.

(Un linguaggio ambiguo è un linguaggio generabile solo da grammatiche ambigue)

2) Grammatica Monotona: equivalente a G di tipo 1 (C.S.), ha produzioni in cui nessuno dei terminali ed NT presenti a sx della produzione si trova nella stessa posizione a dx, o non compare proprio:

(Esempi: ).

Può essere riscritta, con un numero maggiore di produzioni e passaggi, come una grammatica C.S.

3) Linguaggio Finito: un linguaggi di cardinalità infinita, ovvero con parole non infinite

4) Tipo di crescita L: Se le parole di L crescono in maniera lineare, L è C.F., se crescono o in maniera esponenziale

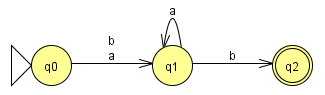
(es. ) o senza un pattern (es. ), L è C.S.

**a+b NELLE ESPRESSIONI REGOLARI (INTESO COME OR)**

Il + NON significa concatenazione.

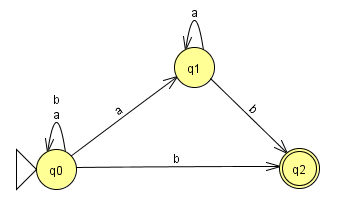
Il + significa OR.

OVVERO:



**(a+b)\* NELLE ESPRESSIONI REGOLARI (INTESO COME ∑\*)**

OVVERO:

(è un NDA, escono 2 a da , vedasi più avanti conversione NDA -> DFA)

**STATI FINALI AUTOMA**

Uno stato è finale quando posso arrivarci solo con una o più parole valide.

NON devo poter arrivare in uno stato finale con parole ancora in costruzione, né con parole non ammesse dal linguaggio.

Se il linguaggio contiene la parola vuota, è anche uno stato finale oltre che iniziale.

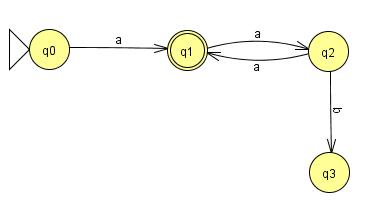
Gli stati finali si denotano sul diagramma dell’ automa con due cerchi concentrici.

**STATO POZZA DI UN AUTOMA [Serve per il Complemento di un Automa]**

Def. Uno stato da cui non è possibile raggiungere nessuno stato finale.

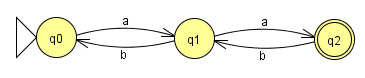
È uno stato opzionale (che posso disegnare per completezza o per usare alcuni algoritmi che lo richiedono, oppure posso trovarne già presenti in automi fornitimi come esercizio)

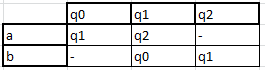
Esempio 1:



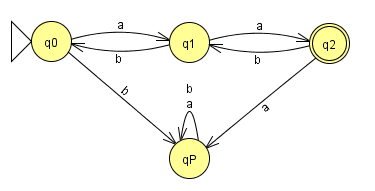
Una volta entrato in non è più possibile raggiungere uno stato finale

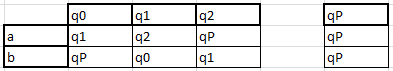
Esempio 2:

L’ automa M: 

Ha come tabella delle transizioni 

È un automa parziale. Posso completarlo aggiungendolo uno stato pozza (opzionale):





1) Collego tutti gli stati con produzioni mancanti allo stato pozza

2) (DA NON DIMENTICARE) Itero lo stato pozza su se stesso.

**COMPLEMENTO DI UN AUTOMA**

1.1) Disegno lo stato pozza (ovvero completo la tabella delle transizioni, rendendo completo l’ automa)

1.2) Non devo dimenticarmi di fare iterare lo stato pozza su se stesso per tutti i simboli dell’ alfabeto dell’ automa

2) Inverto F: gli stati finali diventano non finali, gli stati non finali diventano finali (anche lo stato pozza diventa finale)

**ESERCIZIO SUL COMPLEMENTO DI UN AUTOMA**

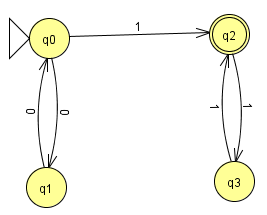
Dato R = (00)\* (11)\* 1, Trovare M tale che T(M) = S(R), e trovare il complemento di M

S(R) = {00}\* unione {11}\* unione {1},

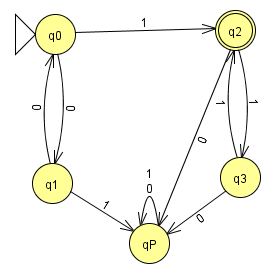
S(R) = { α β 1 | α = (00)\* , β = (11)\* }

ovvero tutte le possibili parole con infinite (o nulle) coppie di zero opzionali, seguite da infinite (o nulle) coppie di 1 opzionali, terminanti con un 1 aggiuntivo (quindi numero dispari di 1)

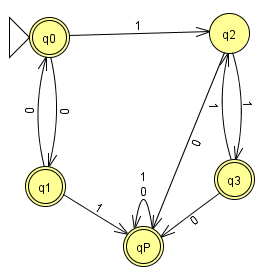
Trovo M:



Disegno lo stato pozza:



Inverto stati finali e non finali:



**PUMPING LEMMA CF – CASO CON 2 TERMINALI**

Dimostrare che non è libero da contesto

[Non commento anche qui i formalismi comuni a tutti i tipi di PL, i commenti generali sono nel PL cf con 3 terminali, qui scrivo solo i commenti specifici di questo tipo di PL]

1. Riscrivo L, come
2. Suppongo L sia libero da contesto. Allora , tale che z ∊ L, |z| > p, si ha:
3. z = uvwxy
4. |vwx| ≤ p
5. v ∙ x ≠ ℇ
6. Considero z = ,

z ∊ L ,

|z| = + p > p

1. [il passo 4 su molte soluzioni è svolto implicitamente, da ciò le perplessità nella lettura delle soluzioni]

[4.1]

Considero *:*

|z’| > |z| = + p [perché || = |uvvwxxy| > |uvwxy|]

[4.2, SOSTITUISCO LE OCCORRENZE DI P CON (P+1), perché se |z’| maggiore stretto |z|, sarà ≥ valori più grandi di |z|]

Sarà quindi |z’| ≥

1. Verifico: [cerco di dimostrare che in realtà |z’| < e non ≥]

|| = |z| + |vx| ≤ |z| +|vwx| ≤ |z| + p = ( + p) + p = + 2p <

[OVVERO, SPIEGATO PASSO PASSO]

|| = |z| + |vx|

|z| + |vx| ≤ |z| +|vwx| [può essere |vx| = |vwx| se w = ℇ]

|z| +|vwx| ≤ |z| + p [per la ① del PL]

|z| + p = ( + p) + p = + 2p [ ho sostituito a |z| il suo valore, calcolato nel punto 3 dell’ esercizio]

< [Non è una formula mi sto solo riconducendo a paragonare il vero valore di z’ con il valore che supposto di z’, che avevo calcolato nel punto 4 dell’ esercizio]

1. Quindi |z’| < , ma doveva essere |z’| ≥ 🡪 z’ L 🡪 L

[In alternativa posso scrivere 🡪ASSURDO 🡪 z’ L 🡪 L ]

**PUMPING LEMMA CF – CASO CON 3 TERMINALI**

Dimostrare che non è libero da contesto.

[Segno con [] le spiegazioni e le maniere equivalenti di scrivere]

**ATTENZIONE:** nel caso con 3 terminali, non posso applicare lo stesso metodo del caso con 2 terminali, cosa succede:

la stringa pompata deve essere ≥ un valore x, operando sulla lunghezza trovo che è < un valore y,

escono sempre y e x non paragonabili, es. x = 2p + 1, y = , non posso dimostrare che in realtà y < x in questo caso,

e non posso dire che per p = 1, < 2p + 1 e il PL è rotto, perché non posso scegliere una p che voglio, lo dico all’ inizio, p è un dato valore che RISPETTA le condizioni del PL; devo necessariamente fare il PL lungo con i sotto-casi

1. Suppongo che L sia libero da contesto [Suppongo che L ∊ ] . Allora , p dipendente solo da L, tale che , si ha:

[N.B. Non significa che ogni parola di L è più lunga di p, ma che per tutte le parole di L che sono più lunghe di p, si ha:]

1. z = uvwxy [ovvero z è scomponibile in 5 sottostringhe, ognuna vuota o piena, tali che:]
2. |vwx| ≤ p
3. v ∙ x ≠ ℇ [ovvero v, x non sono entrambe vuote, possono essere entrambe piene o una piena e una vuota]
4. [Scelgo una parola]

Considero ,

z ∊ L ,

|z| = p + p + p = 3p > p [Devo mostrare che |z| > p]

1. Per la ① del PL, si hanno 5 casi:
2. vwx contiene sole a
3. “ “ “ b
4. “ “ “ c
5. vwx è a cavallo fra a, b
6. “ “ “ “ b, c
7. vwx non può contenere sia a, sia b, sia c,

perché dovrebbe contenere almeno #(b) + 2, ovvero p+2, ma |vwx| ≤ p

[perché dovrebbe essere |vwx| ≥ #(b) + 2 = p + 2, ma |vwx| ≤ p]

1. A)

[A.1]

vwx = , con 0 < k ≤ p

(k > 0 per |vwx| ≥ |vx| > 0 per la ② del PL)

(k ≤ p per la ① del PL)

[specifico perché < k ≤ solo la prima volta, è sempre uguale, non serve riscriverlo nei casi b,c]

[A.2]

, con 0 < ≤ k

[ sarebbe |vx|, ovvero ciò che sto pompando]

( > 0 per la ② del PL)

( ≤ k = |vwx|)

[specifico perché < ≤ solo la prima volta, è sempre uguale, non serve riscriverlo nei casi b,c]

[A.3]

#(b) ≠ max( #(a), #(c) ) 🡪 🡪

B)

vwx = , con 0 < k ≤ p

[ATTENZIONE: qui basta che #(b) sia diverso dal massimo e posso pompare o depompare, è indifferente,

**se fosse stato #(b) ≥ massimo, avrei dovuto necessariamente depompare, per rendere #(b) < max]**

[Per illustrare, ora depompo]

, con 0 < ≤ k

#(b) ≠ max( #(a), #(c) ) 🡪 🡪

C)

vwx = , con 0 < k ≤ p

, con 0 < ≤ k

#(b) ≠ max( #(a), #(c) ) 🡪 🡪

1. [7.1]

D)

vwx = , con 0 < k + r ≤ p

|v| = k’

|x| = r’

[7.2]

Ci sono 3 sotto-casi:

D.1) v ≠ ℇ , x ≠ ℇ [ ovvero v, x non vuote]

D.2) v ≠ ℇ , x = ℇ [ovvero v non vuota, x vuota]

D.3) v = ℇ , x ≠ ℇ

[7.3]

In ogni caso, né v né x possono contenere sia a sia b, o pompandole si otterrebbero stringhe con ripetizioni del tipo abab all’ interno ( tali stringhe non fanno parte di L). v conterrà sole a, x conterrà sole b.

[7.4]

D.1)

**ATTENZIONE: qui devo necessariamente depompare**

, con 0 < ≤ k

k' ≠ 0 , r’ ≠ 0 (per v,x ≠ ℇ)

#(b) ≠ max( #(a), #(c) ) 🡪 🡪

D.2)

, con 0 < ≤ k

k' ≠ 0 , r’ = 0 (per x = ℇ)

#(b) ≠ max( #(a), #(c) ) 🡪 🡪

D.3)

, con 0 < ≤ k

k' = 0 , r’ ≠ 0 (per v = ℇ)

#(b) ≠ max( #(a), #(c) ) 🡪 🡪

[7.5, ripeto]

E)

vwx = , con 0 < k + r ≤ p

|v| = k’

|x| = r’

Ci sono 3 sotto-casi:

E.1) v ≠ ℇ , x ≠ ℇ [ ovvero v, x non vuote]

E.2) v ≠ ℇ , x = ℇ [ovvero v non vuota, x vuota]

E.3) v = ℇ , x ≠ ℇ

v conterrà sole b, x sole c (o, pompando, si otterrebbero ripetizioni del tipo bcbc, inesistenti nel linguaggio)

E.1)

, con 0 < ≤ k

k' ≠ 0 , r’ ≠ 0 (per v,x ≠ ℇ)

#(b) ≠ max( #(a), #(c) ) 🡪 🡪

E.2)

, con 0 < ≤ k

k' ≠ 0 , r’ = 0 (per x = ℇ)

#(b) ≠ max( #(a), #(c) ) 🡪 🡪

E.3)

, con 0 < ≤ k

k' = 0 , r’ ≠ 0 (per v = ℇ)

#(b) ≠ max( #(a), #(c) ) 🡪 🡪

1. Per ogni possibile caso per vwx, la stringa pompata non appartiene al linguaggio. L non è libero da contesto.

[ 🡪

**PUMPING LEMMA REGOLARE / LINEARE DESTRO**

Dimostrare che non è lineare destro.

**[Metodo Semeraro, Automi]**

1. Suppongo che L sia lineare destro. Per il teorema di Kleene, .
2. Allora , con ∑ = {a, b, c}, tale che T(M) = L.

Sia p = |Q| (ovvero il numero di stati di Q).

z ∊ L, |z| > p, si ha:

z = uvw

1. |uv| ≤ p
2. v ≠ ℇ
3. Considero z = ,

z ∊ L ,

|z| > p

1. M possiede solo p stati, ma per produrre le prime p occorrenze di a, servono p+1 stati:

(da a ci sono p+1 stati)

1. Quindi esiste un ciclo, e due stati coincidono.

Il ciclo ha lunghezza ( j - i ), e produce multipli ( j - i ) di a.

1. z =

u v w

|uv| = || ≤ p , per la ① del PL

1. Pompo da stringa v = per k volte

Ottengo z’ =

Se depompo, per k = 0, #(a) < |z’| / 3

Se pompo, per k ≥ 2, #(a) > |z’| / 3

z' L 🡪 L

**[Metodo Fanizzi]**

Uguale al procedimento del PL cf con 3 terminali, con analisi sotto-casi, con le dovute differenze

(uso uvw invece di uvwxy, ecc.)

**CONVERSIONE DI UN AUTOMA: NDA -> DFA (non deterministico -> deterministico)**

1) Traccio la tabella delle transizioni

2.1) Per ogni stato che dalla stessa produzione finisce su più stati, creo un nuovo stato composto uguale all’ unione degli stati

Es. σ( , creo un nuovo stato dal nome

2.2) Uno stato composto produce l’ unione delle produzioni degli stati che lo compongono

Es.

3.1) Aggiorno la tabella delle transizioni

3.2) ATTENZIONE. Se ALLORA

E NON o rimarrebbe un NDA con che produrrebbe due stati

4) Continuo ad aggiornare tabella finchè non trovo più stati composti

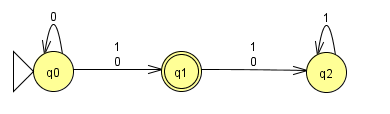
5) Disegno DFA (Può capitare che alcuni vecchi stati siano sovrascritti dai composti o scompaiano, non è un errore)

6) Segno gli stati finali: gli stati finali dell’ automa di partenza () + tutti e soli gli stati composti che contengono almeno uno stato finale dell’ automa di partenza (in questo caso gli stati composti che contengono

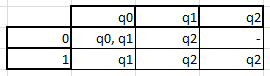
Es. se ci fosse stato anche uno stato non finale, un eventuale stato composto non sarebbe stato finale.

**ESEMPIO SVOLTO**

Dato un automa NDA:



1) Traccio la tabella delle transizioni



finisce su due stati separati, sia su che su , ovvero σ( σ( ,

ovvero σ(

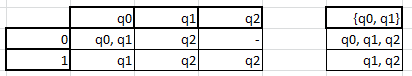
2.1) Creo un nuovo stato che chiamo

2.2) Adesso produce 0 su un unico stato dal nome , invece che su due stati separati

2.3) produce per ogni simbolo l’ unione di ciò che produce con ciò che produce

Ovvero

3.1) Aggiorno la tabella delle transizioni con le produzioni di :



3.2) ATTENZIONE. è comunque un insieme di due stati, non una stringa unica,

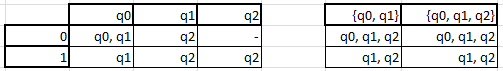
per cui ( , 0 ) ≠ ;

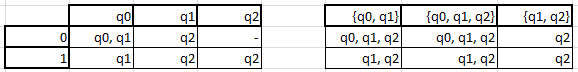
[SBAGLIATO, sarebbe impossibile risolvere la tabella e tradurre in DFA,

rimarrebbe che va su se stesso e su ]

è invece ( , 0 ) =

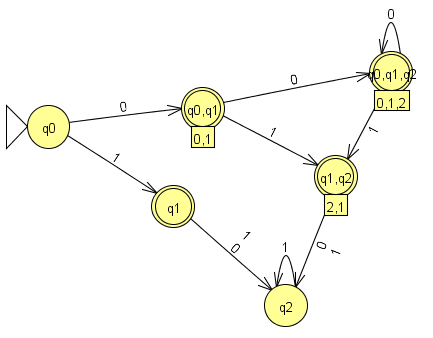
4) Continuo fino a quando non trovo più stati composti e collego ogni stato a ciò che c’è scritto sulla tabella





5) Disegno l’ automa DFA

(Può capitare che alcuni vecchi stati siano sovrascritti dai composti o scompaiano, non è un errore)

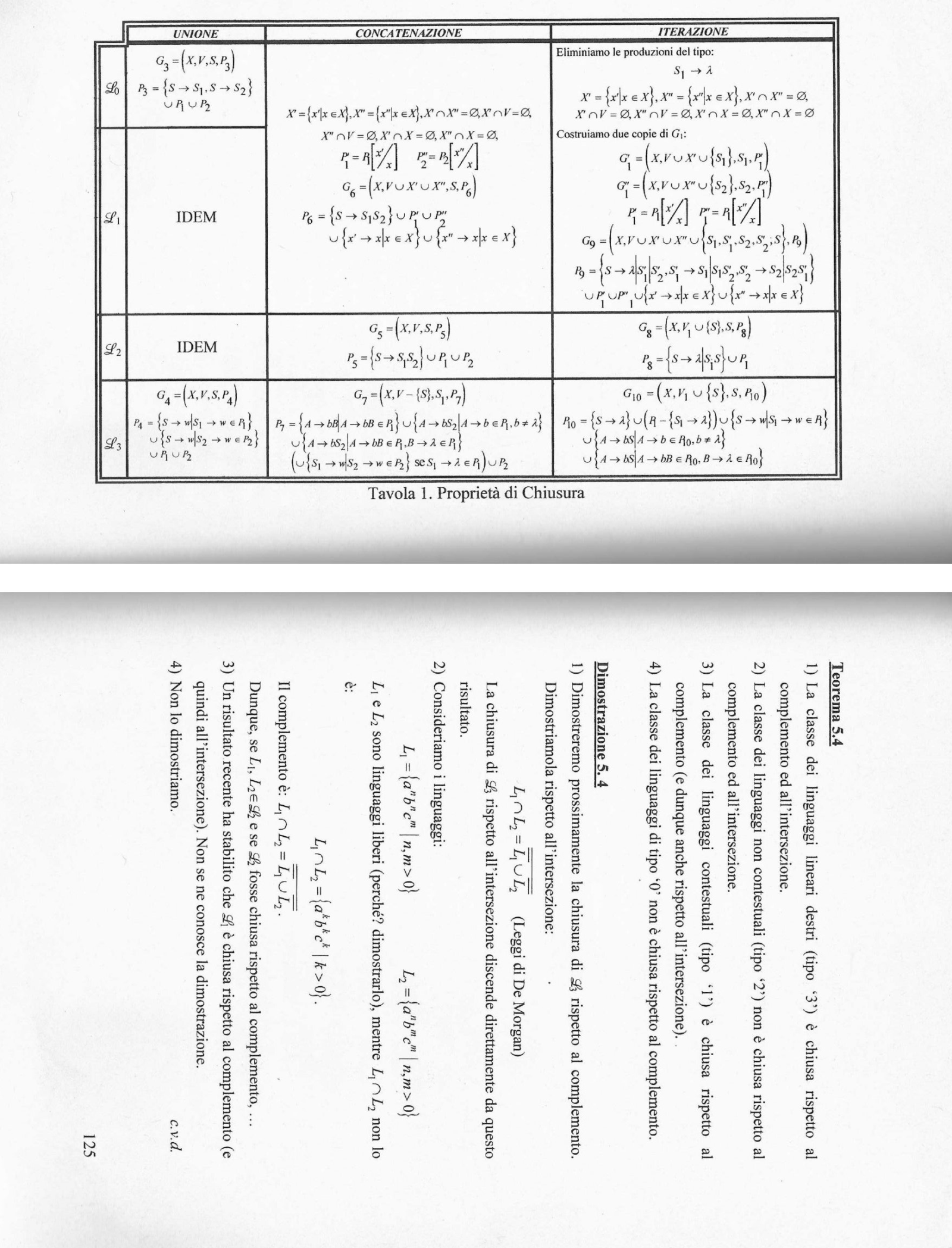


6) Segno gli stati finali: gli stati finali dell’ automa di partenza () + tutti e soli gli stati composti che contengono almeno uno stato finale dell’ automa di partenza (in questo caso gli stati composti che contengono

Es. se ci fosse stato anche uno stato non finale, un eventuale stato composto non sarebbe stato finale.

**PROPRIETÀ DI CHIUSURA**

[N.B. Nei linguaggi di tipo 0 e 1, nella concatenazione (e nell’ iterazione, che è un caso speciale di concatenazione) può portare a generare parole non previste, se , e contiene produzioni del tipo (o viceversa).

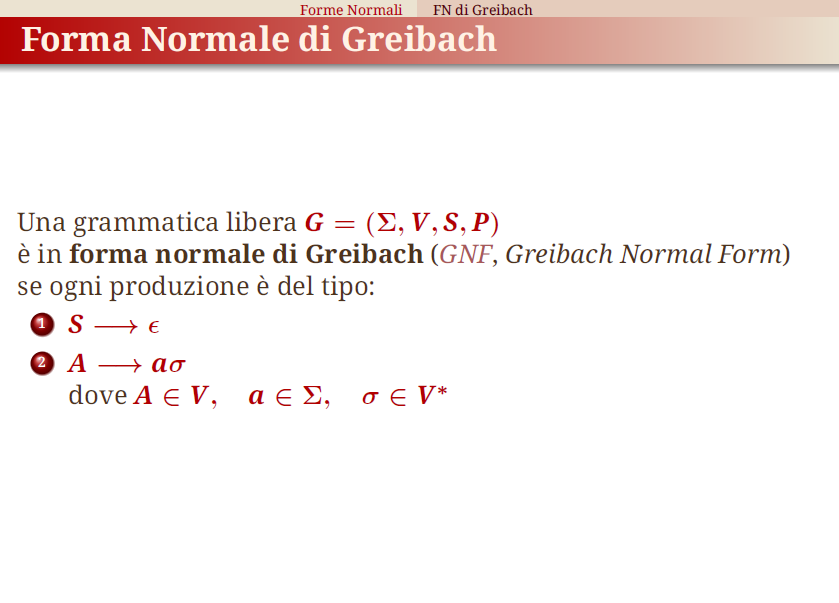
Quindi si creano due cloni dell’ alfabeto diversi fra loro, riconvertiti poi nelle lettere originali alla fine.]

**UNIONE E CONCATENAZIONE GRAMMATICHE DI TIPO 2 (CONTEXT FREE)**

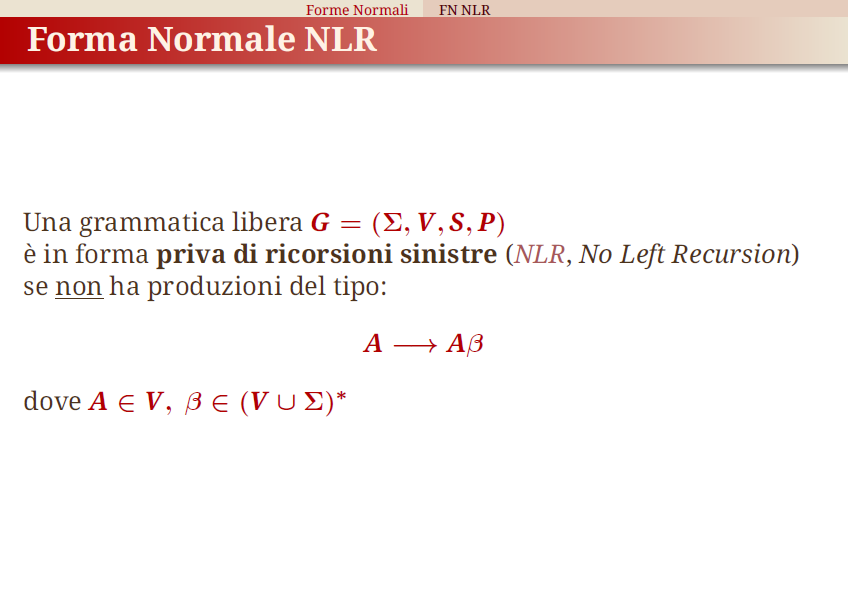
Unione:

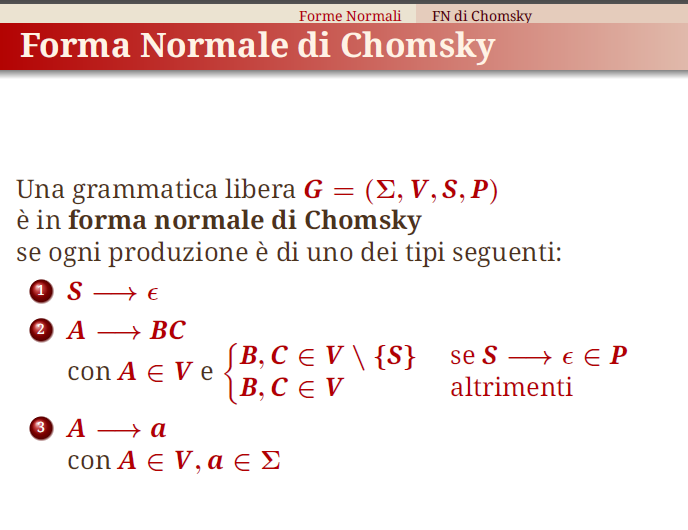
Concatenazione:

**GRAMMATICHE TIPO 2 – SOTTOINSIEMI PARTICOLARI (GNF [Greibach], NLR, CNF)**



**Gnf è presente in alcune tracce d’ esame (es. 17/11/2014)**

****



**INTERSEZIONE LINGUAGGI TIPO 3 e INTERSEZIONE AUTOMI**

**[LEGGI DI DE MORGAN]**

0) Faccio gli automi dei linguaggi

1) Faccio il complemento degli automi [Ricordandomi di aggiungere gli stati pozza se mancano alcune transizioni]

2.A) Unisco gli automi complementati unendone le loro grammatiche e creando automa risultante

2.B) Unisco gli automi complementati con un epsilon-nda che si collega ad entrambi

3) Complemento nuovamente l’ automa ottenuto dall’ unione [ricordandomi di aggiungere eventuali stati pozza

**UNIONE GRAMMATICHE DI TIPO 3 (LINEARI DESTRE)**

1) P = [Unione Linguaggi Tipo 3]

2.2) NOTA BENE: Non devo sostituire le occorrenze di o di con S;

Es. Se in , sarà quindi

Ovvero Parto con S, produco e rimando a .

NON devo sostituire con S, in questa maniera facendo scomparire

USCIREBBE ERRATO

**DA AUTOMA A GRAMMATICA TIPO 3**

1) Mi assicuro che l’ automa sia DFA (in caso contrario, prima lo converto)

2) Rinomino gli stati: = S, gli altri stati saranno A, B, C, ecc.

3) Scrivo le produzioni:

4) Agli stati finali aggiungo la ℇ-produzione (es. Se è finale, sarà fra le possibili produzioni di S)

Perché sugli stati finali posso terminare. Quindi ci sono solo due tipi di produzioni:

1. Uno e un solo terminale seguito da uno e un solo non terminale
2. Una ℇ-produzione

*Non posso scrivere produzioni con solo un terminale (senza NT) perché non posso produrre una lettera nel nulla, devo puntare per forza ad un altro stato se voglio continuare a produrre, oppure fermarmi con una ℇ-produzione.*

*Non ci possono più produzioni da un NT con lo stesso terminale seguito da più NT (tipo A -> bC | bD) perché l’ automa sarebbe un NDA.*

**UNIONE LINGUAGGI DI TIPO 3 (UNENDO AUTOMI)**

1) Faccio gli automi dei due linguaggi

2.A) Unisco le grammatiche degli automi

2.B) Unisco gli automi con l’ epsilon-nda

**UNIONE AUTOMI – METODO 1 (TRAMITE UNIONE GRAMMATICHE DI TIPO 3)**

**[Nel caso speciale in cui non posso unirli a occhio, perché un automa itera su q0 o qualcosa torna indietro su q0, o casi simili]**

1) Definisco le grammatiche dei due automi (con l’ algoritmo per scrivere la grammatica di un automa)

2) Unisco le grammatiche tipo 3

3) Disegno l’ automa risultante dalla grammatica ottenuta

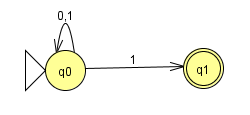
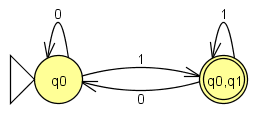
4) Lo converto in DFA

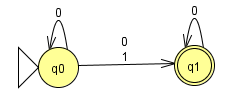
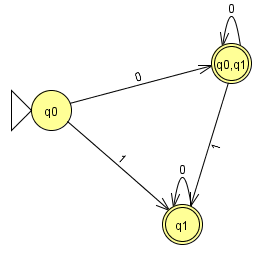
**ESERCIZIO UNIONE AUTOMI (TRAMITE GRAMMATICHE) [Traccia d’esame]**

Date = (0+1)\* 1 ; = 0\* (1+0) 0\*

1) Trovare S(

2) Trovare DFA

Creo :: è NDA. 🡪 Lo converto: applico tabella: […] 🡪

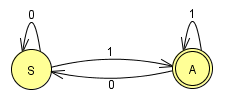
Creo : : è NDA. 🡪 Lo converto: applico tabella: […] 🡪 

Se unissi direttamente i due automi, facendo uscire da le produzioni di entrambi, otterrei parole non valide

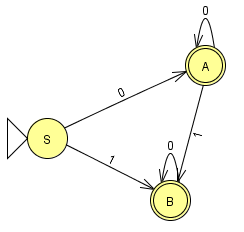
(es. in su posso iterare lo 0, ciò non è accettato dal linguaggio di ).

Definisco la grammatica di :

Riscrivo l’ automa rinominando gli stati, diventa S, diventa A

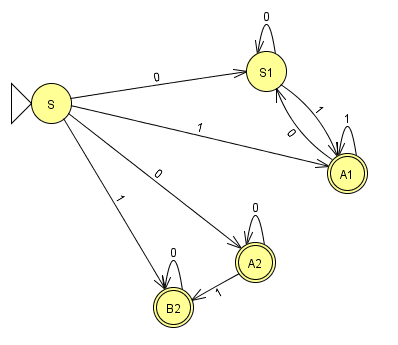


Definisco la grammatica di : ,



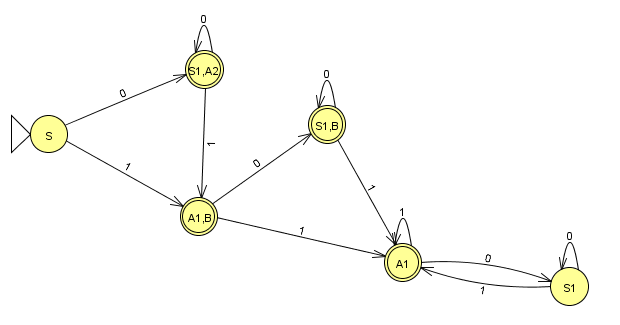
Unisco le due Grammatiche:

Disegno Automa risultante dall’ Unione delle Grammatiche: [N.B. Partendo da S non posso produrre , quindi è superfluo mettere fra gli stati dell’ automa]



: è NDA. 🡪

Lo converto: applico la tabella […] 🡪



**UNIONE AUTOMI – METODO 2 (unirli in un ε-NDA) + CONVERSIONE da ε-NDA a DFA**

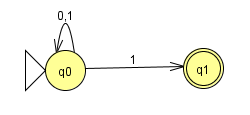
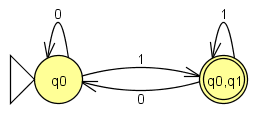
[Si pronuncia Epsilon NDA, ed è diverso da un NDA normale, è un NDA con delle ε-produzioni]

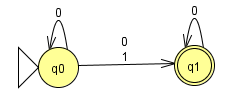
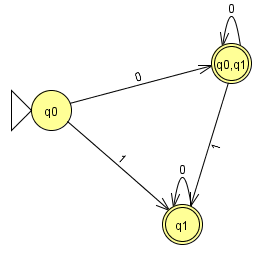
Qui la spiegazione originale del metodo (in inglese) https://www.youtube.com/watch?v=FYk8EpDR3XM

[Prendo lo stesso esercizio usato per l’ unione degli automi tramite le grammatiche, e lo rifaccio con questo metodo]

Devo unire: = (0+1)\* 1 ; = 0\* (1+0) 0\*

E trovare il DFA di S(

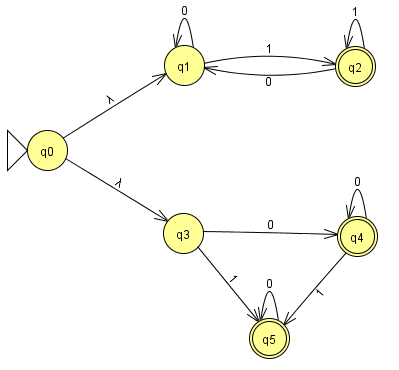
Creo :: è NDA. 🡪 Lo converto: applico tabella: […] 🡪

Creo : : è NDA. 🡪 Lo converto: applico tabella: […] 🡪 

1) Li unisco in un unico ε-NDA

(creo un nuovo stato iniziale, e lo collego con delle ε-produzioni ai vari stati iniziali degli automi da unire.)

2) ATTENZIONE: Controllo se uno dei due precedenti stati iniziali era stato finale (ovvero se il linguaggio accetta stringa vuota) [non è questo il caso]. Se sì, q0 è anche stato finale(o dopo aver eliminato le ε-produzioni, non sarebbe più possibile terminare da q0 su stringa vuota [vedasi esercizio complesso seguente per caso pratico])



3.1) In un ε-NDA, per tracciare la tavola delle transizioni,

non faccio da q0 se produco 0 dove arrivo

ma da q0 se produco ε\*, poi 0, poi ε\* dove arrivo

3.2) Elimino le ε-produzioni

ε\* 0 ε\*

q0 🡪 q0 🡪 Ø

q0 🡪 q1 🡪 q1 🡪 q1

q0 🡪 q3 🡪 q4 🡪 q4

Quindi q0 può passare da ε\*, poi produrre a, arrivando su q2, q4

ε\* 1 ε\*

q0 🡪 q0 🡪 Ø

q0 🡪 q1 🡪 q2 🡪 q2

q0 🡪 q3 🡪 q5 🡪 q5

Quindi q0 può passare da ε\*, poi produrre b, arrivando su q1, q3

4) Traccio la tavola delle transizioni

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | q0 | q1 | q2 | q3 | q4 | q5 |
| 0 | {q1, q4} | q1 | q1 | q4 | q4 | q5 |
| 1 | {q2, q5} | q2 | q2 | q5 | q5 | - |

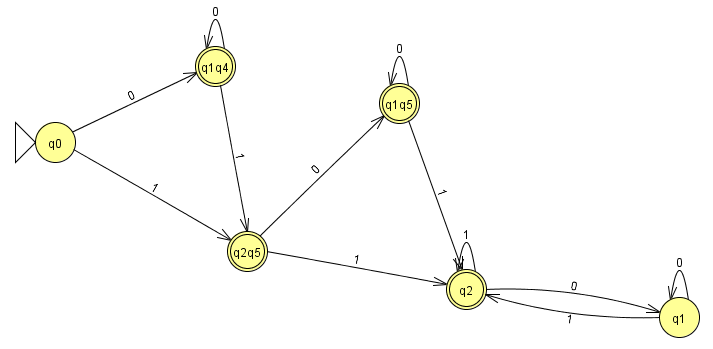
5) Converto in DFA

[N.B. Con l’ unione tramite le grammatiche, arrivo alla stessa tavola delle transizioni, ma in più devo unire le grammatiche, con questo metodo salto l’ unione delle grammatiche e risparmio una decina di minuti]

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | q0 | q1 | q2 | q3 | q4 | q5 |  | {q1, q4} |  | {q2, q5} |  | {q1, q5} |
| 0 | {q1, q4} | q1 | q1 | q4 | q4 | q5 | {q1, q4} | {q1, q5} | {q1, q5} |
| 1 | {q2, q5} | q2 | q2 | q5 | q5 | - | {q2, q5} | q2 | q2 |

6) Traccio il DFA

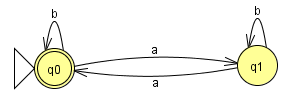
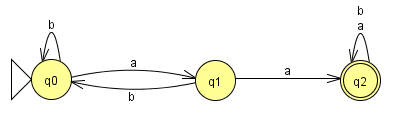
[Ricordiamo: gli stati finali erano q2, q4, q5, ora saranno quelli presenti fra q2, q4, q5 (in questo caso solo q2) + gli stati composti che contengono almeno uno fra q2, q4, q5 (in questo caso tutti, se ci fosse stato uno stato composto tipo {q0,q1} non sarebbe stato finale)]



Risulta lo stesso DFA ottenuto tramite l’ unione delle grammatiche. OK.

**ESERCIZIO COMPLESSO SULL’ UNIONE DI AUTOMI TRAMITE ε-NDA**

Devo unire: ;

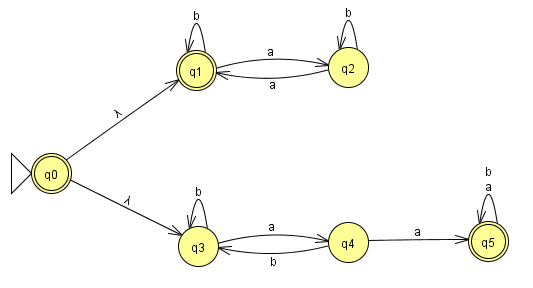
M1 M2 

1) Li unisco in un unico ε-NDA

2) Controllo se uno dei due stati iniziali precedenti è stato finale:

SI 🡪 q0 sarà anche stato finale

(**ATTENZIONE**: in precedenza avevo svolto l’ esercizio senza porre q0 stato finale. Quando andavo a trasformare in DFA, q0 non era stato finale e raggiungeva stati finali solo producendo lettere, non potevo più produrre ε)!!



3.1) In un ε-NDA, per tracciare la tavola delle transizioni,

non faccio da q0 se produco a dove arrivo

ma da q0 se produco ε\*, poi a, poi ε\* dove arrivo

3.2) Elimino le ε-produzioni

ε\* a ε\*

q0 🡪 q0 🡪 Ø

q0 🡪 q1 🡪 q2 🡪 q2

q0 🡪 q3 🡪 q4 🡪 q4

Quindi q0 può passare da ε\*, poi produrre a, arrivando su q2, q4

ε\* b ε\*

q0 🡪 q0 🡪 Ø

q0 🡪 q1 🡪 q1 🡪 q1

q0 🡪 q3 🡪 q3 🡪 q3

Quindi q0 può passare da ε\*, poi produrre b, arrivando su q1, q3

4) Traccio la tavola delle transizioni

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | q0 | q1 | q2 | q3 | q4 | q5 |
| a | {q2, q4} | q2 | q1 | q4 | q5 | q5 |
| b | {q1, q3} | q1 | q2 | q3 | q3 | q5 |

5) Converto in DFA

[N.B. Con l’ unione tramite le grammatiche, arrivo alla stessa tavola delle transizioni, ma in più devo unire le grammatiche, con questo metodo salto l’ unione delle grammatiche e risparmio una decina di minuti]

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | q0 | q1 | q2 | q3 | q4 | q5 |  | {q2, q4} |  | {q1, q3} |
| a | {q2, q4} | q2 | q1 | q4 | q5 | q5 | {q1, q5} | {q2, q4} |
| b | {q1, q3} | q1 | q2 | q3 | q3 | q5 | {q2, q3} | {q1, q3} |

Ci sono ancora stati composti: continuo fino a che non ci sono più produzioni che portano su nuovi stati composti

[Ho spezzato in due la tabella poiché non ci sta in un’ unica riga]

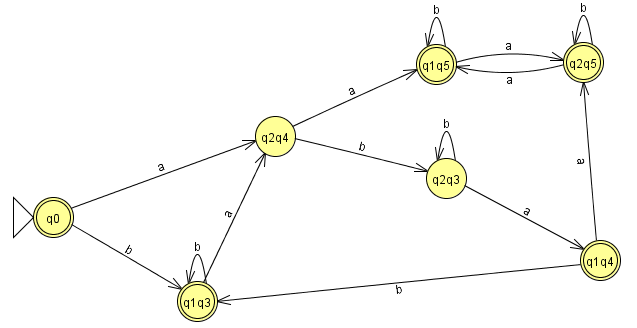
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | {q1, q5} |  | {q2, q3} |  | {q2, q5} |  | {q1, q4} |
| a | {q2, q5} | {q1, q4} | {q1, q5} | {q2, q5} |
| b | {q1, q5} | {q2, q3} | {q2, q5} | {q1, q3} |

6) Traccio il DFA

Gli stati finali erano q0 (!!), q1, q5. Ora saranno quelli presenti fra q0, q1, q5 (in questo caso solo q0)

+ gli stati composti che contengono almeno uno fra q0, q1, q5 (ovvero q1q3, q1q4, q1q5, q2q5).

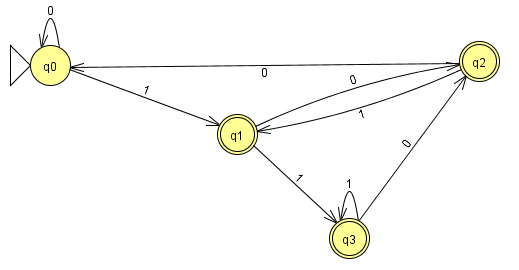
Quindi gli stati composti q2q3 e q2q4, che non contengono né q0, né q1, né q5, non sono stati finali.



**DA AUTOMA A ESPRESSIONE REGOLARE – METODO 1 (Usato da JFLAP)**

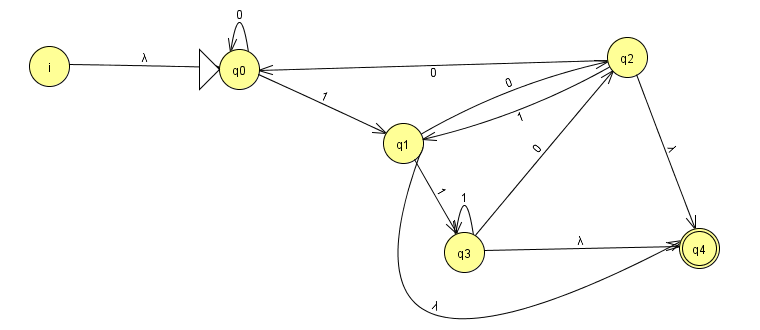
**[ELIMINAZIONE STATI INTERMEDI, ALTERNATIVO A (R\*+SU\*T)SU\*]**

DFA dei binari non divisibili per 100 [4]



1) Aggiungo un nuovo stato iniziale, che si collega a q0 con ε (Passo opzionale, serve per avere uno stato iniziale su cui non arriva nessuna produzione in entrata, utile se voglio eliminare q0 come stato intermedio)

2) Aggiungo un nuovo stato finale, rendo non finali tutti gli stati finali, e li collego con ε al nuovo stato finale.

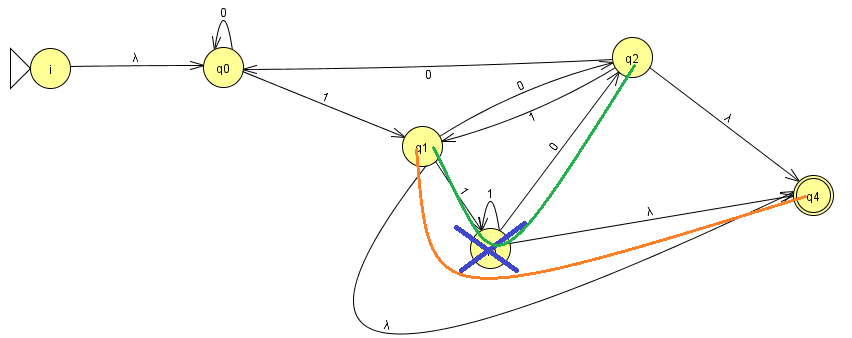


3) Elimino uno stato intermedio alla volta, e trascrivo la tabella dei percorsi (ovvero come ogni stato si collega con tutti gli altri)

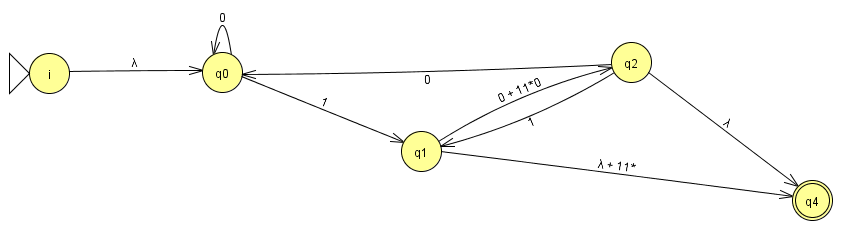
[Non c’è un ordine specifico in cui scegliere gli stati da eliminare, si può scegliere di volta in volta quello più semplice]

3.1) Elimino lo stato intermedio 3, e traccio la nuova tabella delle transizioni

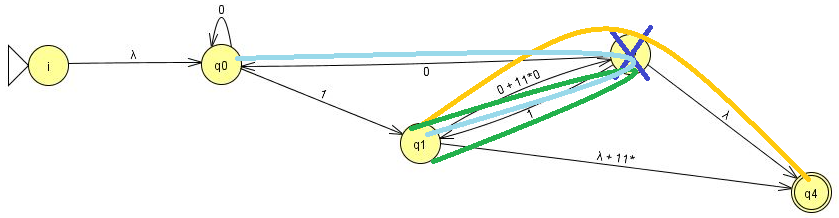
[Considero che ora q1 si deve occupare anche delle transizioni di q3, e deve inglobarle]



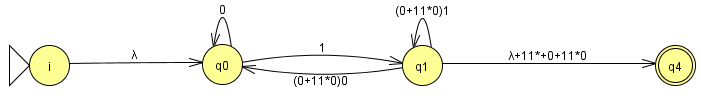
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Partenza | Arrivo |  | Percorso | Commenti |
| 0 | 0 |  | 0 | Non serve scrivere 0\*, l’ iterazione si ottiene dal fatto che vado da 0 su 0 stesso |
| 0 | 1 |  | 1 | Non devo scrivere 0\*1, ovvero i possibili valori con cui posso arrivare su 1 da 0, ma solo il percorso con cui arrivo da 0 a 1 (0\* lo ottengo da 0 su 0) |
| 0 | 2 |  | Ø |  |
| 0 | 4 |  | Ø |  |
| 1 | 0 |  | Ø |  |
| 1 | 1 |  | Ø |  |
| 1 | 2 |  | 0 + 11\*0 | Il percorso originario da 1 a 2 (ovvero 0) + il percorso da 1 a 3 e da 3 a 2, che viene inglobato nel percorso da 1 a 2 eliminando lo stato intermedio 3 |
| 1 | 4 |  | ε + 11\* | Il percorso originario da 1 a 4 (ovvero ε) + il percorso da 1 a 3 e da 3 a 4, che viene inglobato nel percorso da 1 a 4 eliminando lo stato intermedio 3 |
| 2 | 0 |  | 0 |  |
| 2 | 1 |  | 1 |  |
| 2 | 2 |  | Ø |  |
| 2 | 4 |  | ε |  |
| 4 | 0 |  | Ø |  |
| 4 | 1 |  | Ø |  |
| 4 | 2 |  | Ø |  |
| 4 | 4 |  | Ø |  |



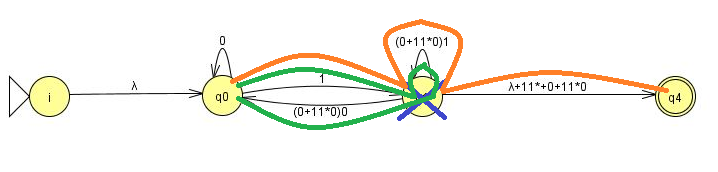
3.2) Elimino lo stato intermedio 2, e traccio la nuova tabella delle transizioni



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Partenza | Arrivo |  | Percorso | Commento |
| 0 | 0 |  | 0 |  |
| 0 | 1 |  | 1 |  |
| 0 | 4 |  | Ø |  |
| 1 | 0 |  | (0 + 11\*0) ∙ 0 |  |
| 1 | 1 |  | (0+11\*0) ∙ 1 | ATTENZIONE, spesso, quando si elimina uno stato, ci si dimentica di controllare se si genera su un altro stato un iterazione su se stesso.  È una frequente fonte di errore. |
| 1 | 4 |  | (ε + 11\*) + (0 + 11\*0) = ε + 11\* + 0 + 11\*0 | Le parentesi qui sono superflue |
| 4 | 0 |  | Ø |  |
| 4 | 1 |  | Ø |  |
| 4 | 4 |  | Ø |  |



3.3) Elimino lo stato intermedio 1, e traccio la nuova tabella delle transizioni



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Partenza | Arrivo |  | Percorso | Commenti |
| 0 | 0 |  | 0 + (1 ∙ ((0+11\*0)1)**\*** ∙ (0+11\*0)0) | ATTENZIONE: non serve mettere tutto fra parentesi e asteriscato perché vado da 0 a 0, ma BISOGNA mettere asteriscata la parte che iterava su 1, perché significa che all’ interno di questa stringa che itera su 0, la sottostringa ((0+11\*0)1)\* può esserci N volte, a differenza di 1 e di (0+11\*0)0 che ci sono sempre e solo 1 volta per iterazione  [L’ asterisco grande è solo per renderlo evidente, non ha alcun significato particolare] |
| 0 | 4 |  | 1 ∙ ((0+11\*0)1)\* ∙ (ε+11\*+0+11\*0) | Stessa cosa qui, devo asteriscare che la parte che iterava su 1, può esserci N volte |
| 4 | 0 |  | Ø |  |
| 4 | 4 |  | Ø |  |

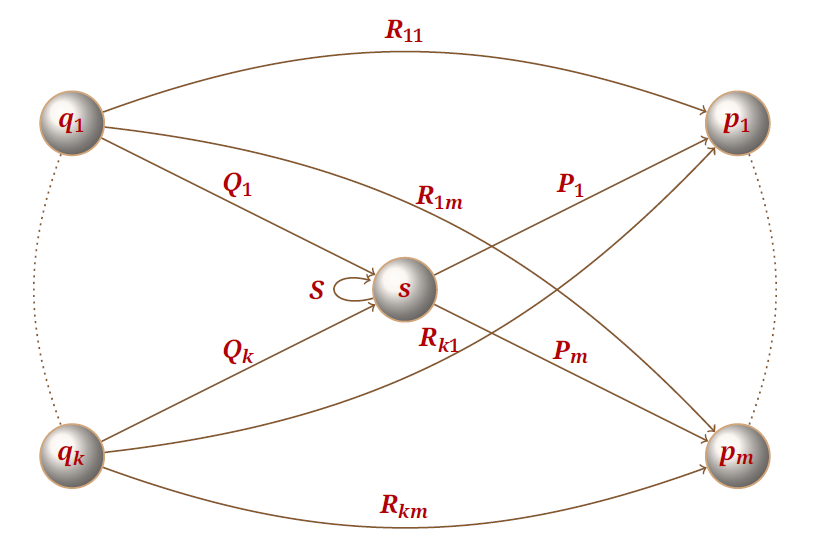
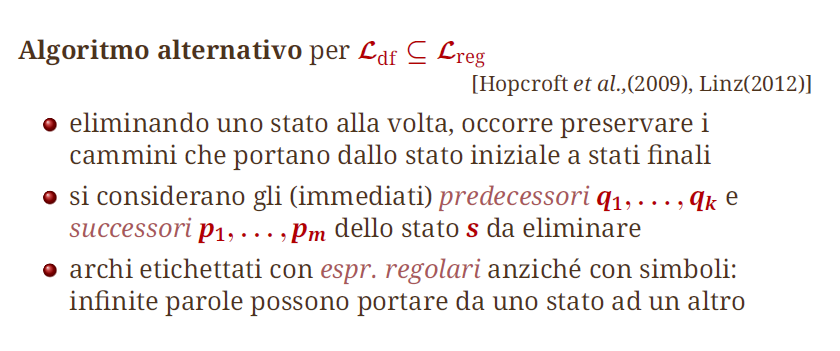
4) Da quest’ ultima tabella deduco che l’ espressione regolare è:

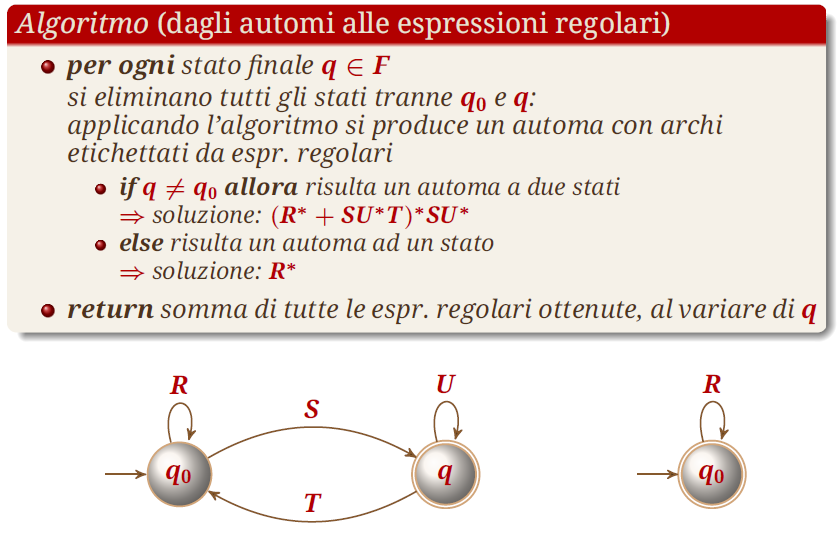
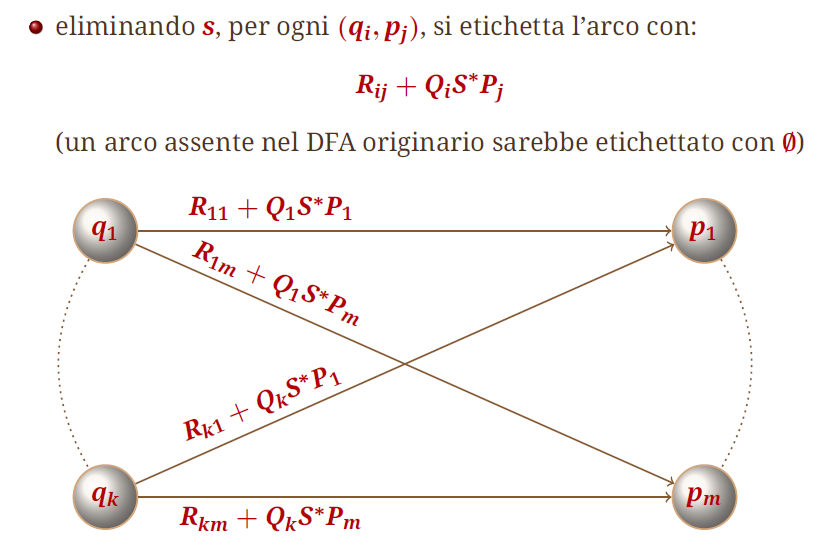
(0 + (1 ∙ ((0+11\*0)1)\* ∙ (0+11\*0)0))**\*** ∙ (1 ∙ ((0+11\*0)1)\* ∙ (ε+11\*+0+11\*0)

5) ATTENZIONE: Controllare sempre a fine esercizio di aver messo asteriscata la parte dell’ espressione regolare che va da 0 a 0 [Nel punto qui sopra ho ingrandito l’ asterisco per far notare]

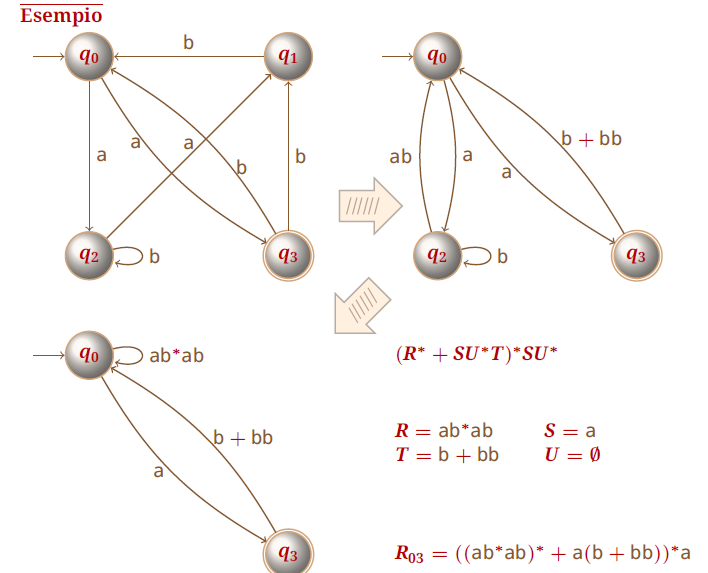
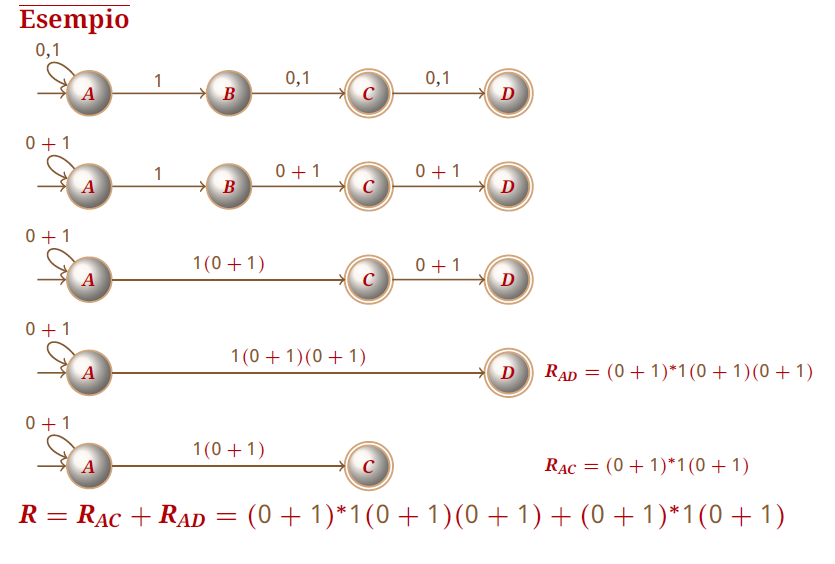
**DA AUTOMA A ESPRESSIONE REGOLARE – METODO 2: (R\* + SU\*T)\* SU\***

**[Messo da Fanizzi nelle slide, non mi ci sono trovato bene; alternativo a JFLAP]**

1,2

3,4

5,6



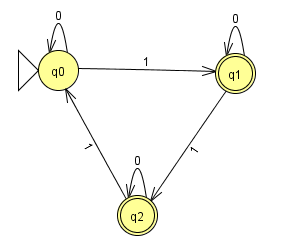
**METODI PER ABBREVIARE LE ESPRESSIONI REGOLARI**

**(ACCORPARE CASI SIMILI CON EPSILON)**

1) Linguaggio numeri binari con massimo 3 uno:

R =

2) Linguaggio binari con #(1) mod 3 ≠ 0



R =

<---->

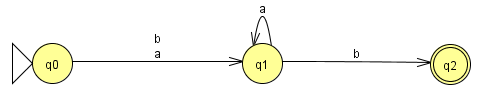
3) […]

**DA ESPRESSIONE REGOLARE AD AUTOMA EPSILON-NDA (Metodo JFLAP)**

**[Non serve per esame, è per curiosità]**

R = (a+b) ∙ a\* ∙ b

Questa è fattibile anche ad occhio, ed è



Un approccio algoritmico invece (valido per regexp anche complesse, per le quali non è possibile disegnare l’ automa ad occhio) è il seguente:

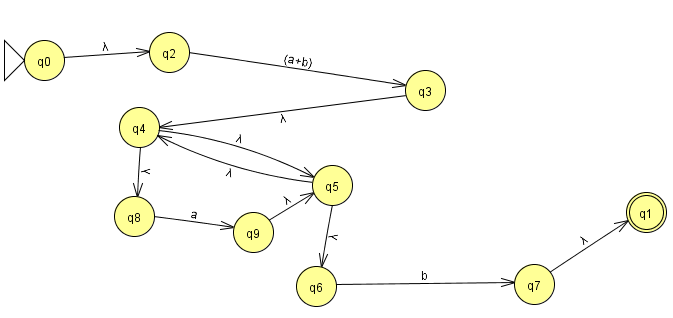
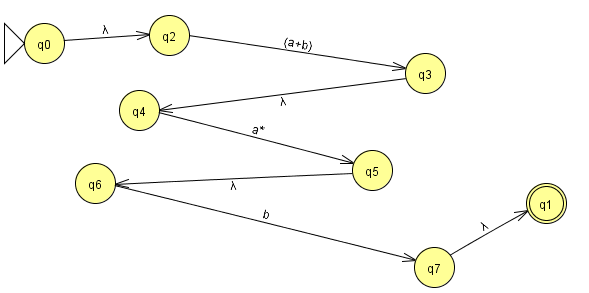
1) Traccio un automa a due stati, iniziale e finale, con l’ iniziale che produce la regexp sul finale



2) Scelgo fra A) metodo lungo JFLAP, o B) metodo veloce

2.A) Scompongo la regexp, e fra un elemento e l’ altro, per evitare possibili errori, ci metto delle epsilon produzioni

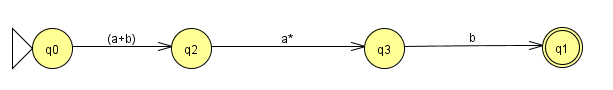
[Questo è il metodo in cui lo fa JFLAP, essendo un computer e non potendo riflettere razionalmente sull’ inutilità di alcune epsilon produzioni, le mette per sicurezza, poi le leva durante la conversione ad automa DFA]

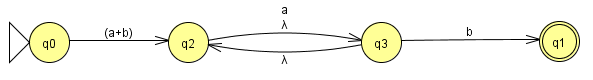


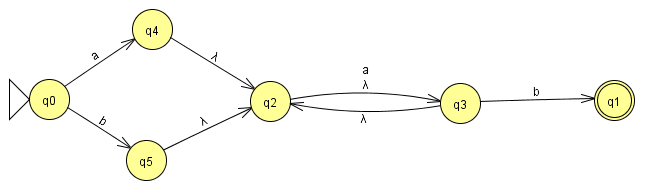
Ecc. [Con le epsilon produzioni persino in casi semplici come questo vengono una quindicina di stati]

**OPPURE**

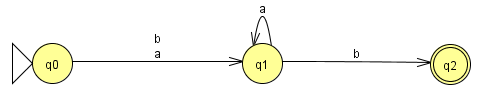
2.B)







Fatto. Ovviamente da qui è facile levare le epsilon produzioni e convertirlo in DFA, e minimizzarlo, e diventa:



**ALGORITMO PER LA COSTRUZIONE DI UN AUTOMA DEL LINGUAGGIO IN QUALSIASI BASE**

**DEI NUMERI DIVISIBILI (O NON DIVISIBILI) PER UN DATO NUMERO n**

0) Determino la base (binaria/decimale/altro), e quindi l’ alfabeto ∑

1.1) Determino i possibili resti della divisione fratto n, che vanno da 0 a n-1

1.2) Converto ogni resto nella base del linguaggio (es. in binario)

1.3) L’ automa avrà n stati, da a

(li disegno uno alla volta quando vado a tracciare i resti, non tutti all’ inizio, perché possono non essere tutti in fila)

1.4) Ogni stato rappresenta un possibile resto, e ogni stringa numerica che genero deve terminare sullo stato dal resto appropriato

2) Traccio ogni resto (ovviamente convertito nella base del linguaggio, es. in binario) assicurandomi che ogni numero sullo stato col resto appropriato

3) Continuo a tracciare i numeri da n in su, fino a quando l’ automa è completo (ovvero fino a quando da ogni stato escono tutti i simboli dell’ alfabeto, es. in binario quando da ogni stato escono sia 0 sia 1)

[Un automa DFA è completo quando non ha spazi vuoti nella tabella di transizione; quando ne ha, è parziale]

4) Segno gli stati finali:

4.A) Se è il linguaggio dei numeri divisibili per n, è finale solo (resto 0)

4.B) Se è il linguaggio dei numeri non divisibili per n, sono finali tutti tranne (resto diverso da 0)

Guida presa (in inglese) da:

http://stackoverflow.com/questions/21897554/design-dfa-accepting-binary-strings-divisible-by-a-number-n

**OSSERVAZIONI SUI NUMERI BINARI (UTILI PER AUTOMA DEI BINARI DIVISIBILI PER n)**

1) Se aggiungo uno 0 alla fine di un numero binario n, ottengo 2n (perché sto moltiplicando per 2)

Es. 1101 = 13; 11010 = 26; 110100 = 52

2) Se aggiungo un 1 alla fine di un numero binario n, ottengo 2n+1 (perché sto moltiplicando per 2 e aggiungendo 1)

Es. 1101 = 13; 11011 = 27; 110111 = 55

**ESERCIZIO SUGLI AUTOMI DEI NUMERI NON DIVISIBILI PER n**

L = { w ∈ {0,1}\* | w non è divisibile per 100 (ovvero per 4) }

0) ∑ = { 0, 1 }

1.1) Possibili resti = { 0, 1, 2, 3 }

1.2) 0 ---> 0

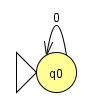
1 ---> 1

2 ---> 10

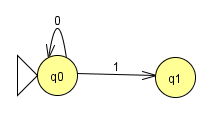
3 ---> 11

1.3) L’ automa avrà 4 stati:

2) Traccio 0: 0 dà resto 0, quindi devo produrlo terminando su

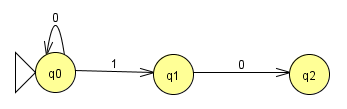


Traccio 1: 1 dà resto 1, quindi devo produrlo terminando su



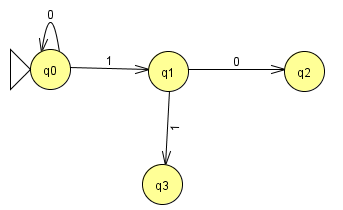
Traccio 10 (ovvero 2):

10 dà resto 2, quindi devo produrlo terminando su



Traccio 11 (ovvero 3)

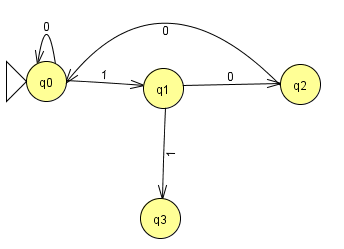
11 dà resto 3, quindi devo produrlo terminando su

Ma non posso mettere dopo , o verrebbe 101, quindi: 

3) Traccio i numeri da 4 in su fino a rendere completo il DFA (ovvero fino a che da ogni stato escono sia 0 sia 1)

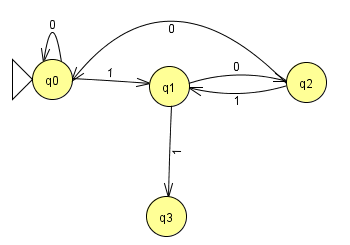
Traccio 100 (ovvero 4)

100 dà resto 0, devo produrlo terminando su



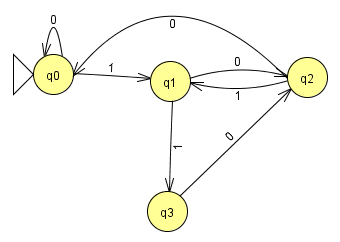
Traccio 101 (ovvero 5)

101 dà resto 1, devo produrlo terminando su



Traccio 110 (ovvero 6)

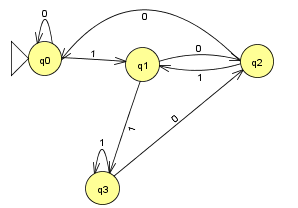
110 dà resto 2, devo produrlo terminando su



Traccio 111 (ovvero 7)

111 dà resto 3, devo produrlo terminando su

Posso far iterare 1 su , perché anche 1111 = 15, 15 % 4 = 3; 11111 = 31, 31 % 4 = 3; ecc.



4) w non deve essere divisibile fratto 4, quindi sono stati finali, no (perché ci finiscono i multipli di 4)

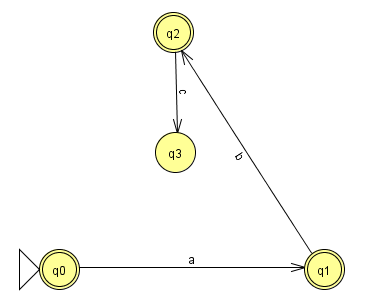
**AUTOMA STRINGHE CHE NON TERMINANO CON abc O SIMILI**

L = { w ϵ {a,b,c}\* | w ≠ vabc, v ϵ {a,b,c}\* }, ovvero il linguaggio delle stringhe che non terminano con abc

Definire l’ automa M tale che T(M) = L.

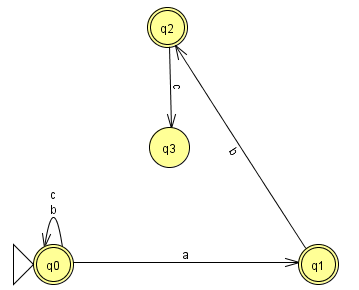
1) Prendo la fine vietata in questione, abc

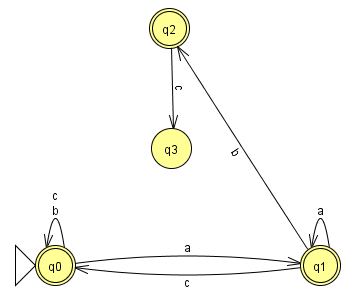
2) Strutturo l’ automa in modo da non poter finire su abc, rendendo q3 non finale, con tutti gli altri possibili casi finali



3) Completo facendo uscire da ogni stato tutte le parole dell’ alfabeto dell’ automa, stando sempre attento a non violare la condizione vietata dal linguaggio.

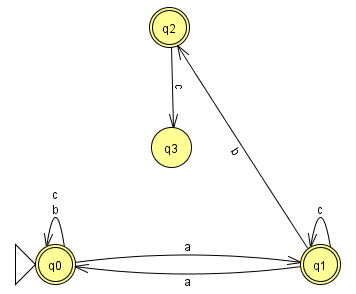
3.1) Da q0 escono: b, c



3.2) Da q1 escono: a, c

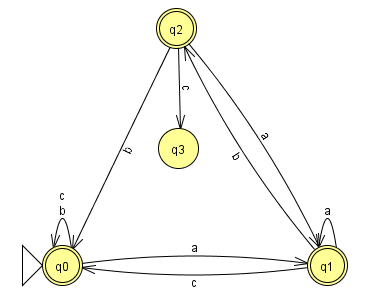
ATTENZIONE: Qui avrei potuto fare andare a su q0, e poi produrre b, poi c, iterando su q0, e finire poiche q0 è stato finale, e avrei violato la condizione.

**ESEMPIO DI ERRORE**



Errore: così potrei finire con abc. Attenzione!

3.3) Da q2 escono: a, b



ATTENZIONE:

Se avessi invertito, q2 manda b su q1, a su q0, avrei potuto fare abc.

Se avessi iterato a,b su q2 stesso, non avrei potuto fare alcune stringhe; controllare sempre.

3.4) Da q3 escono: a, b, c

