Indice

1 Introduzione

1.1 Motivazione

Un linguaggio è uno strumento per descrivere come risolvere i problemi in maniera rigorosa, in modo tale che sia eseguibile da un calcolatore Perché è utile studiare come creare un linguaggio di programmazione?

- non rimanere degli utilizzatori passivi
- capire il funzionamento dietro le quinte di un linguaggio
- domain-specific language (DSL): è un linguaggio pensato per uno specifico problema
- model drivern software development: modo complesso per dire UML e simili
- model checking

1.2 Definizioni base

Un linguaggio è composto da:

- lessico e sintassi
- compilatore: parser + generatore di codice oggetto

La generazione automatica di codice può essere dichiarativa lessico (espressioni regolari o automa a stati finite) o sintassi(grammatiche o automa a pile). Un automa a stati finiti consuma informazioni una alla volta, ne salva una quantità finita. Alcuni esempi di applicazione di automa a stati finiti: software di progettazione di circuiti, analizzatore lessicale, ricerca di parole sul web e protocolli di comunicazione.

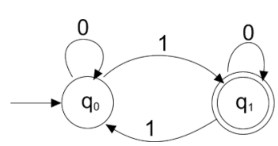


Figura 1: Semplice automa

1.3 Contenuti del corso

- Linguaggi formali e Automi:
  - Automi a stati finiti, espressioni regolari, grammatiche libere, automi a pila, Macchine di Turing, calcolabilità
- Compilatori:
  - Analisi lessicale, analisi sintattica, analisi semantica, generazione di codice
- Logica di base:
  - Logica delle proposizioni e dei predicati
- Modelli computazionali:
  - Specifica di sistemi tramite sistemi di transizione, logiche temporali per la specifica e verifica di proprietà dei sistemi (model checking), sistemi concorrenti (algebre di processi e reti di Petri)

1.4 Informazioni utili

Parte integrante del corso:

- Supporto alla parte teorica usando tool specifici.
  - JFLAP 7.1: http://www.jflap.org (automi/grammatiche)
  - Tina 3.7.5: http://projects.laas.fr/tina (model checking di sistemi di transizione e reti di Petri)
  - LTSA 3.0: http://www.doc.ic.ac.uk/ltsa (sistemi di transizione definiti tramite algebre di processi)
- Nel resto del corso utilizzeremo un ambiente di sviluppo per generare parser/compilatori
  - IntelliJ esteso con plug-in ANTLRv4, ultima versione 1.20 (generatore ANTLR: http://www.antlr.org/)

Libri di testo suggeriti:

- J. E. Hopcroft, R. Motwani e J. D. Ullman: Automi, linguaggi e calcolabilita', Addison-Wesley, Terza Edizione, 2009. Cap. 1–9
- A. V. Aho, M. S. Lam, R. Sethi e J. D. Ullman: Compilatori: principi tecniche e strumenti, Addison Wesley, Seconda Edizione, 2009. Cap. 1–5
- M. Huth e M. Ryan: Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge University Press, Second Edition, 2004. Cap. 1–3

2 Linguaggi regolari

2.1 Alfabeti

Un *alfabeto* è un insieme finito e non vuoto di simboli, comunemente indicato con Σ. Seguono alcuni esempi di alfabeti:

- $\Sigma = \{0{,}1\}$ alfabeto binario
- $\Sigma = \{a,b,...,z\}$ alfabeto di tutte lettere minuscole
- L'insieme ASCII

2.1.1 Stringhe

Una stringa/parola è un insieme di simboli di un alfabeto, 0010 è una stringa che appartiene $\Sigma = \{0{,}1\}$.
La *stringa vuota* è una stringa composta da 0 simboli.
La lunghezza della stringa sono il numeri di caratteri che la compongono (non devono essere unici). La sintassi per la lunghezza di una stringa w è $|w|$, quindi $|001| = 3$ oppure $|\epsilon| = 0$ (nota bene, $\epsilon \neq 0$ ma è di lunghezza 0).

Potenze di un alfabeto

Se Σ è un alfabeto si può esprimere l'insieme di tutte le stringhe di una certa lunghezza con una notazione esponenziale: Σ^k denota tutte le stringhe di lunghezza k con simboli che appartengono a Σ.
Per esempio:
$\Sigma^1 = \{0{,}1\}$
$\Sigma^2 = \{00, 01, 10, 11\}$
$\Sigma^2 = \{000, 001, 010, 011, 100, 101, 110, 111\}$
L'insieme delle stringhe meno quella vuota è segnato come Σ^+, mentre l'insieme che include la stringa vuota è Σ^*,

2.1.2 Concatenazione di stringhe

Siano x e y stringhe, dove i è la lunghezza di x e j è la lunghezza di y, la stringa xy è la stringa risultata dalla concatenazione delle stringhe xy di lunghezza i+j.

2.2 Definizione di linguaggio

Un insieme di stringhe a scelta L $\subseteq \Sigma^*$ si definisce linguaggio su Σ.
Un modo formale per definire un alfabeto è il seguente {w | enunciato su w}, che si traduce in "w tale che enunciato su w".
$\{0^n 1^n | n \geq 1\}$ si traduce in "l'insieme di 0 elevato alla n, 1 alla n tale che n è maggiore o uguale a 1"

3 Automa a stati finiti deterministico

Un automa a stati finiti deterministico consiste in:

1. Un insieme di stati finiti Q
2. Un insieme di simboli di input, Σ
3. Una funzione di transizione, che prende in input uno stato e un simbolo e restituisce uno stato. Tale funzione è spesso indicato con δ ed è usata per rappresentare i archi nella rappresentazione grafica. Ovvero sia q uno stato, a un input allora δ(q,a) è lo stato p tale che esista un arco da q a p.
4. Uno stato iniziale (naturalmente che appartiene a Q)
5. Un insieme di stati accettati finali F. Questo è un sottoinsieme di Q.

Un automa a stati finiti deterministico è spesso chiamato con l'acronimo DFA e viene può essere rappresentato nella seguente maniera concisa:

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

Dove A rappresenta il DFA.

3.1 Elaborazione di stringhe

Per elaborare una stringa è si definisce lo stato iniziale, quello finale e una serie di regole di transizione per poterci arrivare. Se dovessi decodificare la stringa 01 il DFA risulterebbe:

$$A = (Q = \{q1, q2, q3\}, \{0, 1\}, \delta, q_0, \{q1\})$$

I stati sono i sequenti:
$\delta(q_0, 1) = q_0$: leggo come primo stato 1, nessun progresso fatto
$\delta(q_0, 0) = q_2$: leggo come primo stato 0, posso andare avanti e cercare un 1
$\delta(q_2, 1) = q_1$: leggo 1 dopo lo 0, ho trovato la stringa
$\delta(q_2, 0) = q_2$: leggo 0 dopo lo 0, non ho fatto progresso
Nota bene: questa è una notazione arbitraria del libro, q1 e q2 si possono invertire.

3.1.1 Notazioni semplici per DFA

Diagramma di transizione

Dato un DFA $A = (Q, \Sigma, \delta, q_0, F)$ un suo diagramma di transizione è composto da:

- Ogni stato Q è un nodo
- Ogni funzione δ è una freccia
- La freccia Start che denota il primo input
- Gli stati accettati F hanno un doppio cerchio

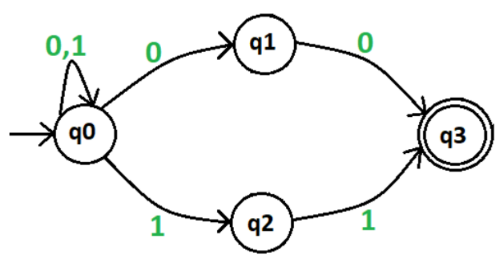


Figura 2: Diagramma di transizione

Tabelle di transizione

Una tabella di transizione è costituita nelle riga dalle funzioni δ e nelle colonne dagli input. Ogni incrocio equivale a uno stato della funzione δ con un input generico a.

	0	1
$\rightarrow$ q$_0$	q$_2$	q$_0$
*q$_1$	q$_1$	q$_1$
q$_2$	q$_2$	q$_1$

Tabella 1: Esempio di tabella

La freccia è lo start e l'asterisco è lo stato finale.

3.1.2 Estensione della funzione di transizione di stringhe

Allo scopo di poter seguire una sequenza di input ci serve definire una funzione di transizione estesa. Se δ è una funzione di transizione, chiameremo $\hat{\delta}$ la sua funzione estesa. La funzione estesa prende in input q e una stringa w e ritorna uno stato p.
Ogni stato viene calcolato grazie allo stato esteso precedente:

$$\hat{\delta}(q,w) = \delta(\hat{\delta}(q,x), a)$$

Esempio

L = { w | w ha un numero pari di 0 e di 1 }
Nota bene: 0 (numero di simboli) è pari quindi conta come stato accettato, ed è l'unico stato accettato.

q_0: 0 e 1 sono pari
q_1: 0 pari 1 dispari
q_2: 1 pari 0 dispari
q_3: 0 dispari 1 dispari

$$A = (\{q0, q1, q2, q3\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

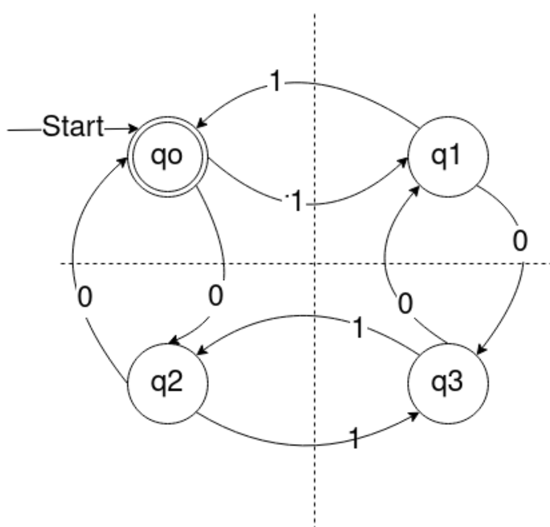


Figura 3: Diagramma

	0	1
$\rightarrow * q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Tabella 2: Esempio funzioni

Ora applichiamo le funzione di transizione estesa per verificare che 110101 abbia 0 e 1 pari:

- $\hat{\delta}(q_0, \epsilon) = q_0$
- $\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \epsilon), 1) = \delta(q_0, 1) = q_1$
- $\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_1, 1) = q_0$
- $\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_1, 11), 0) = \delta(q_0, 1) = q_2$
- $\hat{\delta}(q_0, 1101) = \delta(\hat{\delta}(q_0, 110), 1) = \delta(q_2, 1) = q_3$
- $\hat{\delta}(q_0, 11010) = \delta(\hat{\delta}(q_0, 1101), 0) = \delta(q_3, 0) = q_1$
- $\hat{\delta}(q_0, 110101) = \delta(\hat{\delta}(q_0, 11010), 1) = \delta(q_1, 1) = q_0$

A ogni simbolo aggiunto posso usare la funzione estesa precedente per calcolare il prossimo stato, in questo caso la sequenza ha un numero pari di 0 e 1.

4 Automa a stati finiti non deterministici

Un NFA (nondeterministic finite automaton) può trovarsi contemporaneamente in diversi stati. L'automa "scommette" sul input su certe proprietà dell'input.
I NFA sono spesso più succinti e facili da definire rispetto ai DFA, un DFA può avere un numero di stati addirittura esponenziale rispetto a un NFA. Ogni NFA può essere convertito in un DFA.

4.1 Descrizione informale

A differenza di un DFA, una funzione di stato in un NFA può restituire 0 o più stati. Immaginiamo di dover identificare se una stringa finisce con 01. Di seguito il diagramma di transizione sarà il seguente. Come è possibile

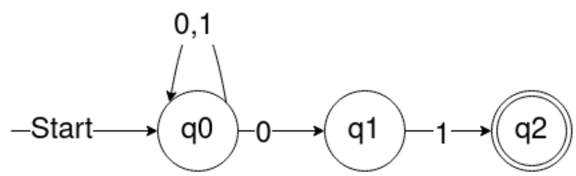


Figura 4: NFA che accetta stringa che finisce con 01

notare q_0 può può restituire due stati se riceve uno 0. Il NFA esegue molteplici stadi alla ricerca del pattern (simile a un processo che si moltiplica).

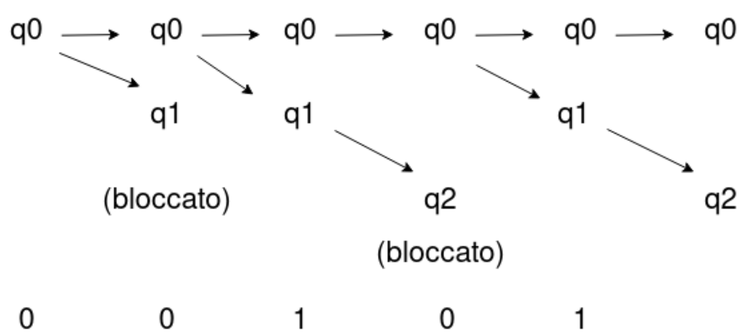


Figura 5: Gli stati del NFA

Ogni volta che il NFA accetta uno stato 0 crea due processi, un q_1 e q_0 A ogni successivo input tutti i processi vanno avanti, nel nostro caso il q_1 "muore". Al secondo giro viene creato q_1 che muore alla quarta iterazione perché non è l'ultimo simbolo. Durante la quarta iterazione nasce q_1 che alla quinta ci porta uno stato accettato.

4.2 Definizione formale

Formalmente un NFA si definisce come un DFA.

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

1. Un insieme di stati finiti Q
2. Un insieme di simboli di input, Σ
3. Una funzione di transizione, che prende in input uno stato e un simbolo e restituisce ***un insieme di stati***. Questa è l'unica differenza rispetto al DFA, dove ci viene restituito un singolo stato.
4. Uno stato iniziale (naturalmente che appartiene a Q)
5. Un insieme di stati accettati finali F. Questo è un sottoinsieme di Q.

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

Tabella 3: Tabella di transizione di una NFA che accetta una stringa che finisce con 01

L'unica differenza con una tabella DFA è che negli incroci ci sono dei insiemi di stati di output (singoletto quanto è uno solo), mentre se la transizione non esiste viene segnata con $\emptyset$.

4.3 Funzione di transizione estesa

Come per i DFA bisogna prendere la funzione di transizione e renderla estesa. In questo caso lo stato precedente può ritorna un insieme di stati, quindi bisognare fare l'unione di questi. La funzione estesa di δ si chiamerà $\hat{\delta}$.

$$\bigcup_{x=2}^{k} \delta(p_i, a) = \{r_1, r_2, ..., r_m\}$$

Usiamo $\hat{\delta}$ per calcolare se la stringa 00101 finisce con 01.

1. $\hat{\delta}(q_0, \epsilon) = \{q_0\}$
2. $\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$
3. $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
4. $\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
5. $\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
6. $\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

Abbiamo un risultato positivo, q_2 mentre q_0 viene scartato

4.4 Linguaggio NFA

Come abbiamo visto sopra, il fatto di avere uno stato non accettabile al termine dell'operazione non significa che non abbia avuto successo.
Formalmente se $A = (Q, \Sigma, \delta, q_0, F)$ è un NFA allora:

$$L(A) = \{w | \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

In parole povere L(A) è l'insieme delle stringhe w in Σ^* tale che $\hat{\delta}(q_0, w)$ contenga almeno uno stato accettante.

4.5 Equivalenza tra DFA e NFA

Di solito è più facile ottenere un NFA piuttosto che un DFA per un linguaggio. Nel migliori dei casi un DFA ha circa tanti stati quanti un NFA, ma più transizioni. Nel caso peggiore un DFA ha 2^n stati, mentre un NFA n.
Come detto in precedenza ogni NFA può essere ricondotto a un DFA, questo andrà dimostrato costruendo un DFA per insiemi a partire da un NFA.
Dato un NFA $A = (Q_N, \Sigma, \delta_N, q_0, F_N)$ possiamo costruire un DFA
$A = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ tale che L(D)=L(N) (che i linguaggio sono uguali).
Si noti che i due linguaggi condividono lo stesso alfabeto.
Gli altri D componenti sono fatti nel seguente modo:

- Q_D è formato da un insieme di insiemi di Q_N, in termini formali Q_D è l'insieme potenza di Q_N. Quindi se Q_N ha n stati allora Q_D ha 2^n stati, questo è vero nella teoria, nella pratica gli stati non raggiungibili non contano quindi tendono a essere meno di 2^n.

- F_D è l'insieme dei sottoinsiemi di S di Q_N tale che $S \cap F_N \neq \emptyset$. F_D è quindi formato dagli sottoinsiemi di stati N che includono almeno uno stato accettante.

- Per ogni insieme $S \subseteq Q_N$ e per ogni simbolo a in Σ,

$$\delta_D(S, a) = \bigcup_{p\ in\ S} \delta_N(p, a)$$

Ovvero l'insieme $\delta_D(S, a)$ è calcolato tramite l'unione di tutti gli insiemi p in S.

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Tabella 4: Stringa che termina con 01, NFA $\rightarrow$ DFA

La tabella precedente era deterministica nonostante fosse formata da insiemi, *ogni insieme è uno stato*, e non sono insieme di stati. Per rendere più chiara l'idea possiamo cambiare notazione.

	0	1
A	A	A
→B	E	B
C	A	D
∗D	A	A
E	E	F
∗F	E	B
∗G	A	D
∗H	E	F

Tabella 5: Stringa che termina con 01, notazione nuova

Tra gli 8 stati presenti in tabella possiamo raggiungere: B, E e F. Gli atri stati sono irraggiungibili o non esistenti. È possibile evitare di costruire questi stati compiendo una "valuta differita".
Trattando i l'insieme di stati come un unico stato composto da un insieme è possibile riscrivere la DFA in questo modo:

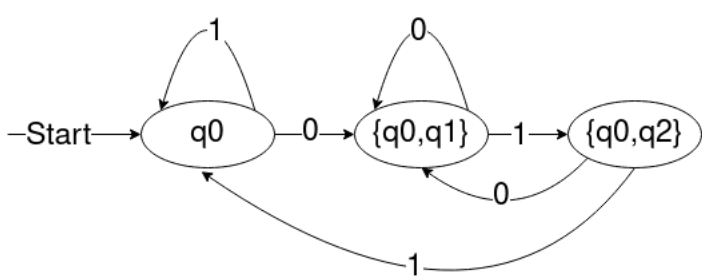


Figura 6: Grafico DFA convertito da NFA

Teorema

Se $D = (Q_N, \Sigma, \delta_N, q_0, F_N)$ è il DFA trovato per costruzione a partire dal NFA $N = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ allora L(D)=L(N).

Teorema

Un linguaggio L è accettato da un DFA se e solo se L è accettato da un NFA.

5 Automa con epsilon-transazioni

Un estensione degli automa è la capacità di poter ammettere come input la stringa vuota ϵ. È come se l'NFA compisse una transizioni spontaneamente. Tale NFA si chiamerà ϵ-NFA

5.1 Uso delle epsilon-transizioni

L'esempio di seguito tratta le ϵ come invisibili, possono mutare lo stato ma non sono contante nella catena.

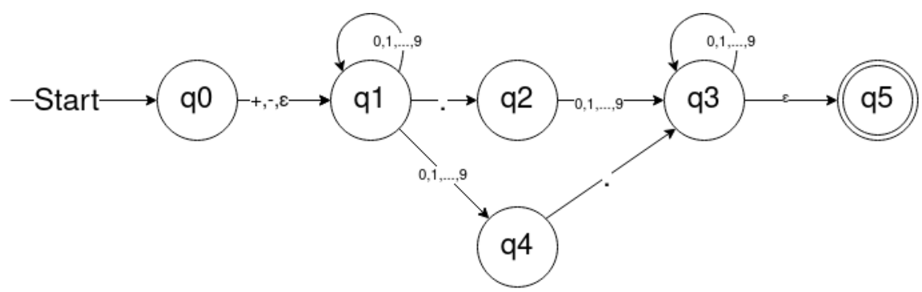


Figura 7: epsilon-NFA che accetta numeri decimali

L'ϵ-NFA in figura accetta numeri decimali formati da:

1. un segno +,- facoltativo
2. una sequenza di cifre
3. un punto decimale
4. una seconda sequenza di cifre

È possibile avere input vuoti prima della virgola $\delta(q_1, .) = q_2$ e dopo la virgola $\delta(q_4, .) = q_3$ ma non entrambi. Il segno è facoltativo $\delta(q_0, \epsilon) = q_1$.
In q_3 l'automa può "scommettere" che la sequenza sia finita oppure può andare avanti a leggere.

5.2 Notazione formale di epsilon-NFA

La definizione forma di un ϵ-NFA è uguale a quella di un NFA, va solo specificate le informazioni relative alla transizione ϵ.
Una ϵ-NFA è definita con $A = (Q, \Sigma, \delta, q_0, F)$, dove δ è una funzione di transizione che richiede come input:

1. uno stato Q
2. un elemento $\Sigma \cup \{\epsilon\}$, ovvero un simbolo di input oppure il simbolo ϵ. Questa distinzione viene fatta per evitare confusione.

ϵ-NFA per riconoscere un numero decimale

$$E = (\{q_0, q_1, ..., q_5\}, \{., +, -, 1, ..., 9\}, \delta, q_0, \{q_5\})$$

	ϵ	+,-	.	0,1,...,9
q_0	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
q_5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Tabella 6: Tabella di transizione per un numero decimale

5.3 Epsilon chiusure

Un ϵ-chiusura è un cammino fatto solo di transizioni ϵ. Formalmente tale stato si scrive ENCLOSE(q) = insieme di stati.

5.4 Transizioni estese di epsilon-NFA

Grazie alle ϵ-chiusure possiamo definire cosa significa accettare un input.
Supponiamo $E = (Q, \Sigma, \delta, q_0, F)$ un σ-NFA, $\hat{\delta}$(q,w) è la funzione di transizione estesa le cui etichette concatenate descrivono la stringa w.
BASE $\hat{\delta}$(q,w) = ENCLOSE(q), se l'etichetta è ϵ posso seguire solo cammini ϵ, definizione di ENCLOSE.
INDUZIONE Supponiamo w abbia forma xa, dove a è l'ultimo simbolo, che non può essere ϵ perché non appartiene a Σ:

1. Poniamo $\hat{\delta}$(q,x) = $\{p_1, p_2, ..., p_k\}$ in questo modo tutti i cammini p_i sono tutti gli stati raggiungibili da q a x. Questi stati possono terminare con ϵ oppure contenere altre ϵ transizioni

2. Sia $\bigcup_{i=1}^{k} \delta(p_i, a)$ l'insieme $\{r_1, r_2, ..., r_m\}$, ovvero tutte le transizioni da a a x.

3. Infine $\hat{\delta}$(q,w) = $\bigcup_{j=1}^{m} ENCLOSE(r_j)$, questo chiude gli archi rimasti dopo a

Forma contratta

$$\hat{\delta}(q, xa) = \bigcup_{p \in \hat{\delta}(q,x)} (\bigcup_{t \in \delta(q,a)} ENCLOSE(t))$$

Il linguaggio accettato è L(E) = $\{w | \hat{\delta}(q_0, w) \cap F \neq 0\}$

5.5 Da epsilon-NFA a DFA

Dato un ϵ-NFA possiamo costruire un equivalente DFA per sottoinsiemi. Sia $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ un ϵ-NFA il suo equivalente DFA è

$$D = (Q_D, \Sigma, \delta_D, q_0, F_D)$$

ovvero:

1. Q_D è l'insieme di sottoinsiemi Q_E. Ogni stato accessibile in D è un sottoinsieme ϵ-chiuso di Q_E, in termini formali S$\subseteq$ Q_E tale che S = ENCLOSE(S).

2. q_D=ENCLOSE(q_0)

3. F_D contiene almeno uno stato accettante in E.
   $F_D = \{S | S$ è in Q_D e $S \cap F_E \neq 0\}$

4. $\hat{\delta}(q, xa) = \bigcup_{p \in \hat{\delta}(q,x)} (\bigcup_{t \in \delta(q,a)} ENCLOSE(t))$

Teorema Un linguaggio è linguaggio L è accetto da un ϵ-NFA se è solo se è accettato da un DFA.

6 Espressioni regolari

Le espressioni regolari definiscono gli stessi linguaggi definiti dai vari automi: *linguaggi regolari*. A differenza degli automi, le espressioni regolari descrivono linguaggi in maniera dichiarativa. Per questo motivo le espressioni regolari sono molto diffuse, per esempio nel commando unix *grep* oppure negli analizzatori lessicali.

6.1 Operatori lessicali

L'espressione lessicale 01*+10* denota il linguaggio 0 seguito da qualsiasi numero di 1 oppure 1 seguito da qualsiasi numero di 0.
Per poter definire le operazioni sulle regex (sinonimo di espressione regolare) dobbiamo definire tali operazioni sui linguaggi che esse rappresentano:

1. *Unione* di due linguaggi L ed M, L∪M, indica tutte le stringhe che appartengono ad L e ad M oppure a entrambi.

2. *Concatenazione* di due linguaggi L ed M è l'insieme di stringhe formate dalla concatenazione di una qualsiasi stringa L con una qualsiasi stringa M. Tale operazione è indicata così: L$\cdot M$ oppure semplicemente LM. Per L={001,10,111} e M={ϵ,001} LM={001,10,111,001001, 10001, 111001}

3. *Chiusura*(o *star* o chiusura di Kleene) di un linguaggio L, indicata come L*, rappresenta l'insieme delle stringhe che si possono formare tramite concatenazione e ripetizione di qualsiasi stringa in L. Nel caso L={0,1} L* rappresenta l'alfabeto binario, qualsiasi combo di 0 e 1. Nel caso L={0, 11} L* rappresenta qualsiasi stringa che abbia una o più coppie di 1, NB 011 è valido ma come 01111, mentre 101 non è valido, non abbiamo né la stringa 10 né la stringa 01. Formalmente L* è l'unione infinita $\bigcup_{i>0} L_i$ dove $L^0 = \{\epsilon\}$, $L^1 = L$, $L^i = LL...L$.

6.2 Proprietà regex

- $L \cup M = M \cup L$ L'unione è commutativa
- $(L \cup M) \cup M = L \cup (M \cup L)$ L'unione è associativa
- $(LM)M = L(ML)$ La concatenazione è associativa (LM $\neq$ ML)
- $\emptyset \cup L = L \cup \emptyset = L$
- $\{\epsilon\} \cup L = L \cup \{\epsilon\} = L$
- $\emptyset L = L\emptyset = \emptyset$
- $L(M \cup N) = LM \cup LN$
- $(M \cup N)L = ML \cup NL$
- $L \cup L = L$
- $\emptyset^* = \{\epsilon\}, \{\epsilon\}^* = \{\epsilon\}$
- $L^+ = LL^* = L^*L,\ L^* = L^* \cup \{\epsilon\}$

6.3 Costruzione di regex

Servono modi per raggruppare le espressioni regolari, in questo caso vengono usati operatori algebrici comuni. Di seguito verranno definite regex lecite E con il loro corrispondente linguaggio L(E).

BASE

1. le costanti ϵ e $\emptyset$ sono regex, rispettivamente del linguaggio $\{\epsilon\}$ e $\{\emptyset\}$, in altri termini L(ϵ) =$\{ \epsilon\}$ e L($\emptyset$) = $\{\emptyset\}$.
2. Se a è un simbolo allora **a** è una regex che denota il linguaggio {a}, ovvero L(**a**) = {a}. (si usa il grassetto per distinguere simboli da regex)
3. Una lettera maiuscola qualsiasi, di solito L, viene usata per indicare un linguaggio arbitrario

INDUZIONE

1. Data E ed L regex, allora $E + L$ è una regex che indica l'unione dei due linguaggi L(E) e L(L), in altre parole L(E+F) = L(E) $\cup$ L(F)
2. Date E e F due regex, EF indica la concatenazione tra i due linguaggio L(E) e L(F), in altri termini L(EF) = L(E)L(F).
3. Data E una regex, E* indica la chiusura del linguaggio L(E), in altri termini L(E*) = (L(E))*
4. Data E una regex, allora anche (E) è una regex valida che appartiene sempre al linguaggio E, in termini formali L((E)) = L(E)

Esempio di regex

Si crei una regex che descriva una linguaggio che è fatto di 0 e 1 alternati. Intuitivamente si potrebbe provare **01***, che è errato, questo indica tutte le stringhe che hanno uno 0 e un numero arbitrario di 1. (**01**)* è corretto, però indica per forza un linguaggio di 01 alternati, quindi 101010 non sarebbe valido

Uniamo regex per descrivere il caso: (**10***) 10 alternato, **0(10)*** 10 con 0 all'inizio, **1(01)*** 01 con 1 all'inizio, in conclusione

$$\mathbf{(01)^*+(10)^*+0(10)^*+1(01)^*}$$

Un modo più contratto sarebbe quello di aggiungere un 1 facoltativo all'inizio e uno 0 facoltativo alla fine

$$\mathbf{(\epsilon+1)(01)^*)(\epsilon+0)}$$

6.4 Precedenza degli operatori

1. Star ha la precedenza massima
2. concatenazione
3. unione

Naturalmente si possono usare parentesi per decidere il proprio ordine e inoltre è consigliato farlo anche se non fosse necessario per rendere più chiara l'espressione.

7 Automa a stati finite e regex

Abbiamo visto che le regex e gli automi a stati finiti possono descrive gli stessi linguaggi, va solo dimostrato che formalmente.
Dobbiamo dimostrare che:

1. Ogni linguaggio definito da un automa è definito anche da una regex, useremo un DFA per comodità
2. Ogni linguaggio definito da una regex è definita da un automa, useremo un ϵ-NFA per comodità

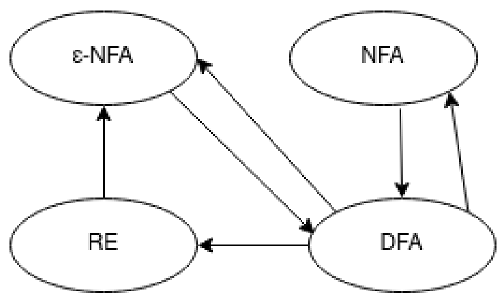


Figura 8: Conversioni

7.1 Da DFA a regex

Teorema
Se L = L(A) per un DFA A, allora esiste una regex R tale che L = L(R).
Il procedimento formale e matematico è formato dal espansione di ogni singolo stato tramite la formula:

$$R_{ij}^{k} = R_{ij}^{k-1} + R_{ik}^{k-1}(R_{kk}^{k-1}R_{kj}^{k-1})$$

In parole povere sto calcolando l'espressione regolare da uno stato j a uno stato i k volte, una per ogni stato.
Questo procedimento è molto lungo, perché l'espressione va effettuata per ogni transizione, unita e poi ridotta.
Tenendo però a mente questa formalità è possibile usare un metodo più gestibile, ovvero *l'eliminazione per stati.*

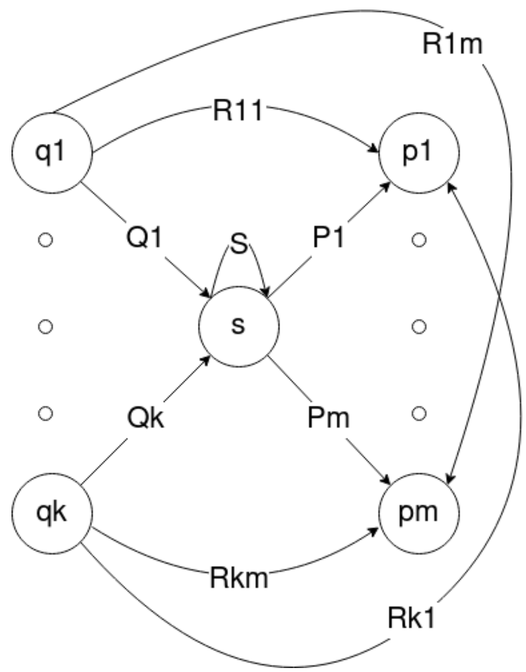


Figura 9: Eliminazione per stati

- s è lo stato generico che sta per essere eliminato
- $q_1, q_2, ..., q_k$ sono i k stati precedenti a s
- Q_i sono tutte le transizioni precedenti
- $p_1, p_2, ..., p_k$ sono i k stati successi a s
- P_i sono tutte le transizioni successive
- R_{ij} sono tutte le transizioni tramite regex, bisogna definirne una per ogni direzione ij ma se non dovesse esistere basterà scrivere $\emptyset$

A questo punto possiamo iniziare a costruire l'espressione regolare a partire dall'automa.

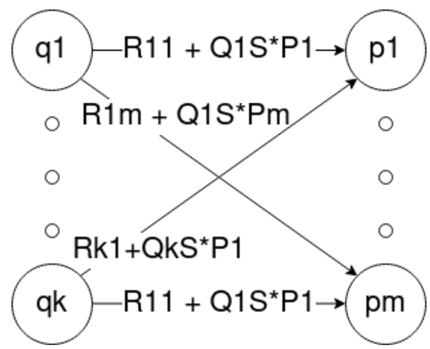


Figura 10: Eliminazione di s

1. Bisogna eliminare tutti gli stati intermedi ad eccezione di q_0.
2. Se $q_0 \neq q_1$ allora questo stato può essere espresso come E_q = (R + SU*T)*SU*, un cammino generico illustrato in figura.

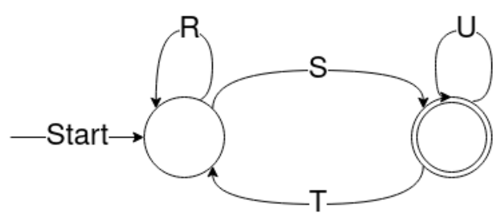


Figura 11: Automa generico 2 stati

3. Se q_0 è accettante allora la regex è data da R*

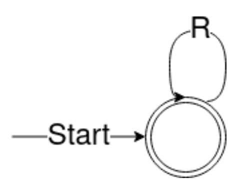


Figura 12: Automa generico 1 stato

Dato un NFA come segue.

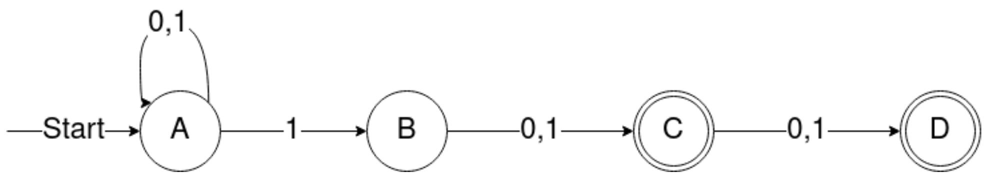


Figura 13: NFA esempio

Esprimiamo le sue funzioni di transizioni come regex

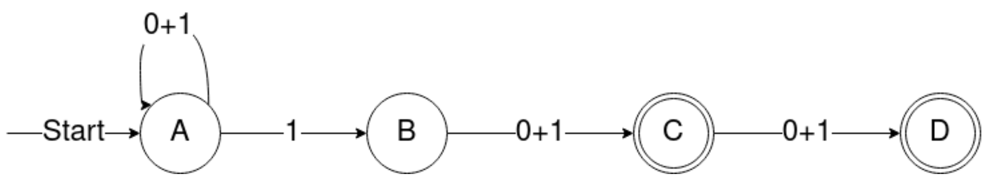


Figura 14: NFA con archi regex

Il primo stato che rimuoviamo è B, applicando la fromula. $R_{11} + Q_1 S^* P_1$, in questo caso risulta $\emptyset + 1\emptyset$*(0+1), che si può ridurre in 1(0+1).
NB $\emptyset$* equivale a ϵ, non annulla le regex, mentre $\emptyset$ sì

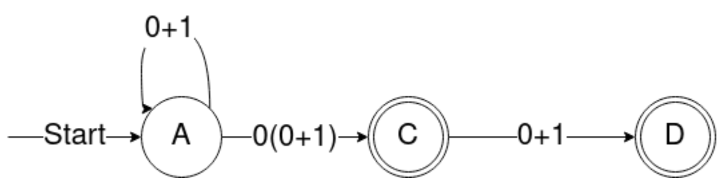


Figura 15: B rimosso

Eliminiamo C

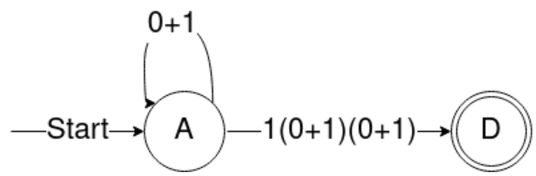


Figura 16: C rimosso

Ora possiamo applicare (R + SU*T)*SU*, quindi

- R=(0+1)
- S=1(0+1)(0+1)
- T=$\emptyset$
- U=$\emptyset$.

$$((0+1)+1(0+1)(0+1)\emptyset^{*}\emptyset)^{*}1(0+1)(0+1)\emptyset$$

Possiamo semplificare U* perché è equivalente a ϵ e possiamo eliminare SU*T perché è T è $\emptyset$.

$$(0+1)^{*}1(0+1)(0+1)$$

Questo è lo stato accettante D, è necessario calcolare lo stato accettante C. Ripartendo dalla fig.15 applichiamo di nuovo E_Q ottenendo (0+1)*1(0+1). L'espressione finale è data dalla **somma** delle 2 espressioni.

$$(0+1)*1(0+1)+(0+1)^{*}1(0+1)(0+1)$$

7.2 Da regex a automi

Teorema Per ogni rex R possiamo costruire un ϵ-NFA A tale che L(R)=L(A). Questo si dimostra per induzione strutturale, prendendo come base gli automi ϵ, $\emptyset$ e a.

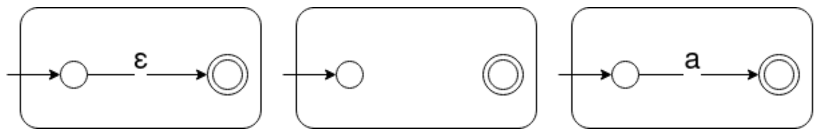


Figura 17: Stati base

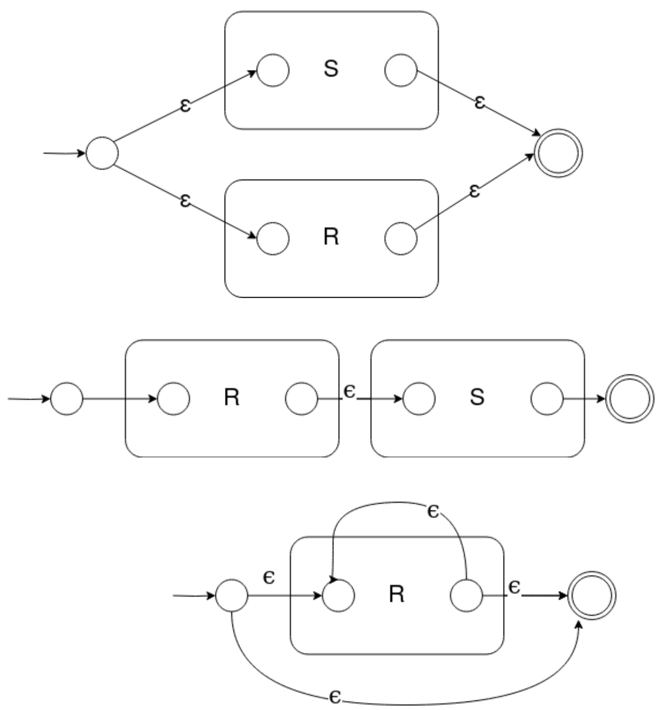


Figura 18: R+S, RS e R*

R+S significa che viene percorso 1 dei 2 espressioni. RS significa che una volta percorso R, quello diventa lo stato iniziale di S. R* va in loop su se stesso.

Usando i blocchi precedenti convertiamo (0+1)*1(0+1)

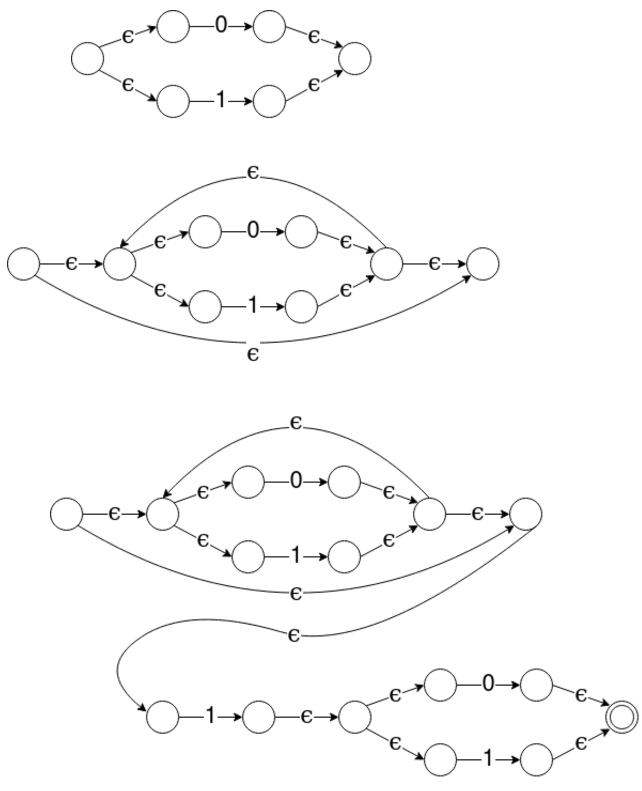


Figura 19: R+S, RS e R*

8 Proprietà dei linguaggi regolari

- *Pumping Lemma*: Ogni linguaggio regolare soddisfa il pumping di lemma
- *Proprietà di chiusura*: Possibilità di costruire un nuovo automa a partire da altri automi, seguendo specifiche operazioni
- *Proprietà di decisione*: Analisi di automi, come l'equivalenza
- *Tecniche di minimizzazione*: Possiamo ridurre un automa

8.1 Pumping Lemma

Un linguaggio non è detto che sia regolare.
Immaginiamo di avere un linguaggio $L_{01} = \{0^n\ 1^n | n \geq 1\}$. Questo è un linguaggio che accetta una stringa con tanti 1 quanti 0. Perché questo linguaggio possa essere un DFA deve avere un numero finito di stati, diciamo k. Quindi dopo $k+1$ simboli, $\epsilon, 0, 00, ..., 0^k$ ci troviamo in un qualche stato. Poiché gli stati sono limitati esistono 2 strade diverse per cui ci troviamo nello stesso stato, chiamiamoli 0^j e 0^i.
Ora immaginiamo dallo stato j di iniziare a leggere 1, l'automa deve fermarsi quando ha letto j quantità di 1, ma non può farlo perché non ricorda lo stato, potrebbe finire dopo i quantità di 1, L_{01} non è regolare.
Teorema Sia L un linguaggio regolare, allora esiste una costante n tale che, per ogni stringa w in L dove $|w| \geq n$ possiamo scomporre w in 3 stringhe $w = xyz$ tale che:

1. $y \neq \epsilon$
2. $|x, y| \leq n$
3. per ogni $k \geq 0$ anche xy^kz è in L

Ovvero c'è una stringa non vuota replicabile da qualche parte, senza uscire dal linguaggio.
Dimostrazione Supponiamo che L sia regolare. Allora L=L(A) e supponiamo che A abbia n stati. Ora consideriamo una stringa w dove $w = a_1, a_2, ..., a_m$ $m \geq n$ e ogni a_i è un simbolo di input. Definiamo la sua funzione $\delta(a_1, a_2, ..., a_n)$ che descrivere tutte le p_i transizioni, e $q_0 = p_0$.

Per il principio della piccionata tutti gli stati non possono essere distinti, quindi esistono due stati p_i e p_j dove $0 \leq i \leq j \leq n$ tale che $p_i = p_j$. Possiamo scomporre w in w=xyz:

1. $x = a_1, a_2, ..., a_i$
2. $y = a_{i+1}, a_{i+2}, ..., a_j$
3. $x = a_{j+1}, a_{j+2}, ..., a_m$

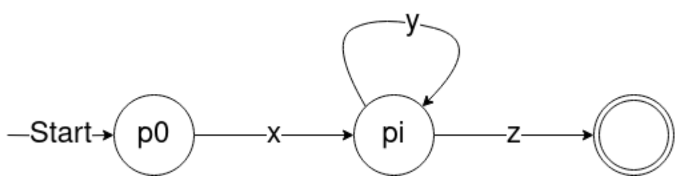


Figura 20: A un certo punto i stati si ripetono

Se k=0 siamo allo stato accettante, se $k \geq 0$ allora dobbiamo necessariamente fare dei loop, perché l'input è xy^kz.

8.2 Chiusura dei linguaggi regolari

Sia L e M due linguaggi regolari allora i seguenti sono a loro volta linguaggi regolari.

- *Unione*: $L \cup M$
- *Intersezione*: $L \cap M$
- *Complemento*: N
- *Differenza*: $L \quad M$
- *Inversione*: $LR = \{wR : w \in L\}$
- *Chiusura*: L^*
- *Concatenazione*: $L.M$

Teorema Sia L e M linguaggi regolari allora anche L ∪ M è un linguaggio regolare.
Dimostrazione L ed M sono linguaggi descritti dalle espressioni regolari S ed R, quindi L=L(S) e M=L(R) quindi L∪M=L(R+S).

Teorema Se L è un linguaggio regolare sull'alfabeto Σ allora anche $\overline{L} = \Sigma^* - L$.
Dimostrazione Sia L=L(A) per un DFA A=$(Q, \Sigma, \delta, q_0, F)$, allora $\overline{L}$=L(B) dove B è il DFA $(Q, \Sigma, \delta, q_0, Q - F)$, quindi B ha gli stati accentanti opposti a quelli di A. In questo caso l'unico modo per cui w è in L(B) se e solo se $\delta(q_0, w)$ è in $Q - F$, ovvero **non** è in L(A).

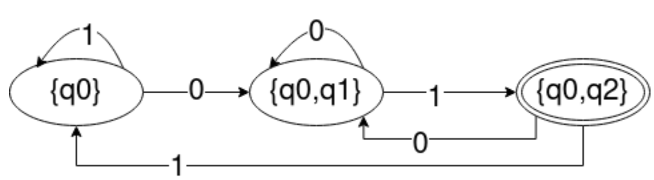


Figura 21: Diagramma di A

Il diagramma di B risulta opposto

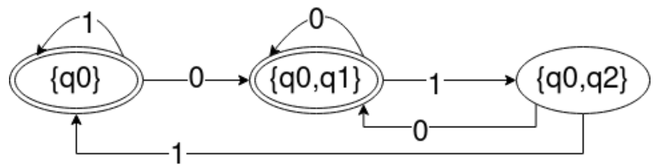


Figura 22: Diagramma di B

Teorema 4.8. Se L e M sono regolari, allora anche $L \cap M$ è regolare. **Dimostrazione** Le 3 operazioni booleane sono interdipendenti, quindi possiamo usare le due precedenti per dimostrare l'Intersezione

$$L \cap M = \overline{\overline{L} \cup \overline{M}}$$

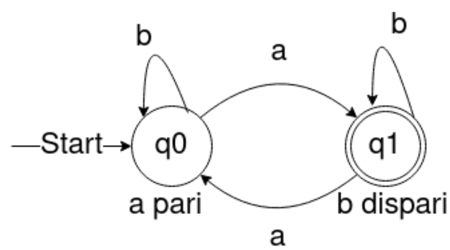
9 Esercitazioni

9.1 Esercitazione 09/21/23

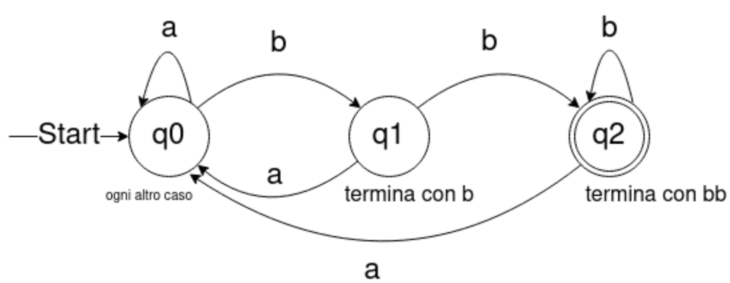
9.1.1 Costruzione DFA

Consideriamo l'alfabeto {a,b}. Realizzare dei DFA che riconoscono i seguenti linguaggi:

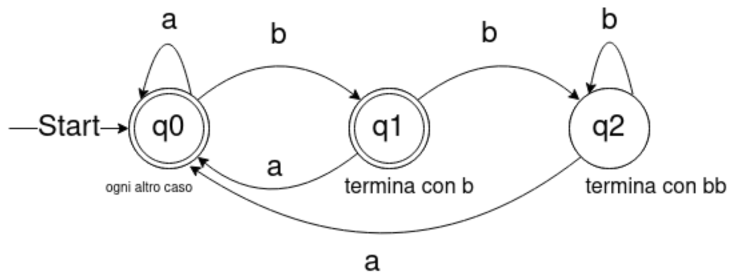
1. stringhe con un numero dispari di a
   SI ab,aaa,bba,aaba
   NO ϵ,aa



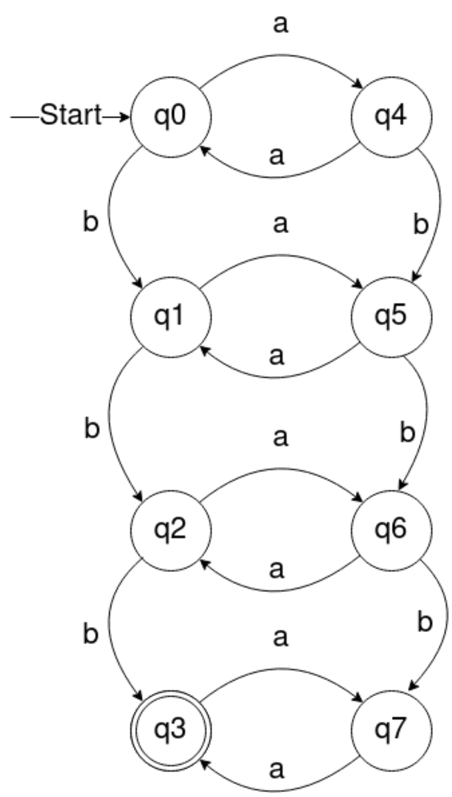
2. stringhe che terminano con bb
   SI bb,babb
   NO ϵ,ba,a,aba



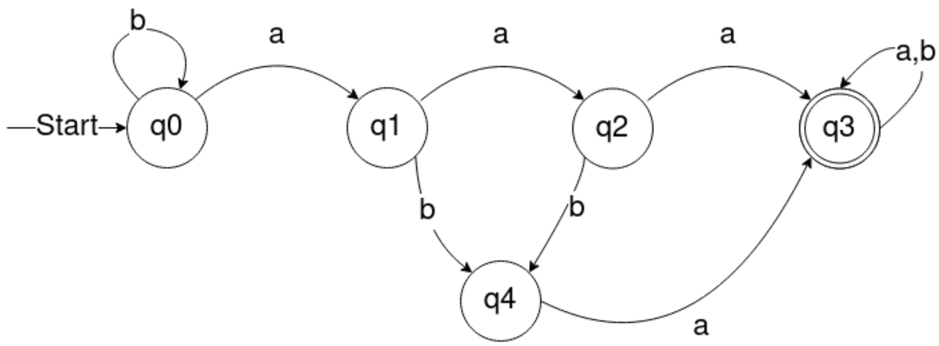
3. stringhe che non terminano con bb
   SI ϵ,ba,a,aba
   NO bb,babb



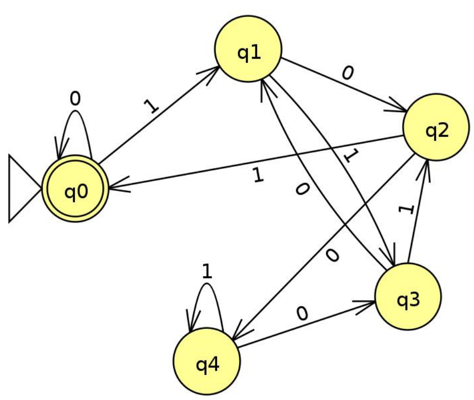
4. stringhe con un numero pari di a ed almeno 3 b
   SI bbb,bababb
   NO ϵ,bbaa,bababa



5. stringhe che contengono la sottostringa aaa o la sottostringa aba (contengono almeno una delle due)
   SI babab,aaaa,aaaba
   NO ϵ,abba,a,b,ab

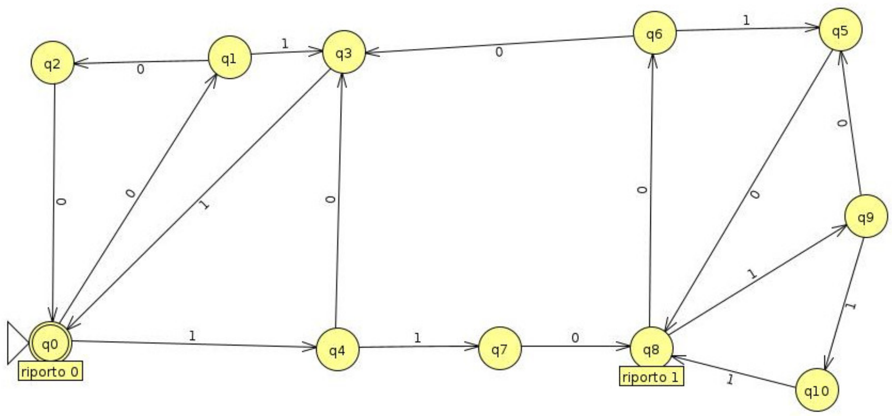


6. Realizzare un DFA che riconosca il seguente linguaggio su alfabeto {0,1}: stringhe che interpretate come numero binario risultano un multiplo di 5
   SI 101,1010,1111,0
   NO 111,1,10



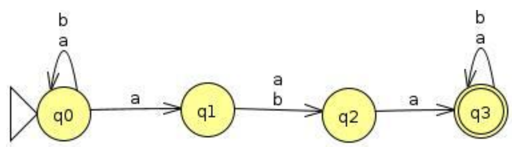
NB: Devi considerare tutti i numeri fino a 5!

7. Sempre considerando alfabeto {0,1}, realizzare un DFA che controlla la correttezza delle somme binarie: data la stringa: $a_0b_0c_0a_1b_1c_1...a_nb_nc_n$ controlla se $a_n...a_1a_0+b_n...b_1b_0 = c_n...c_1c_0$ (cioè a+b=c con a,b,c numeri binari con stessa lunghezza)

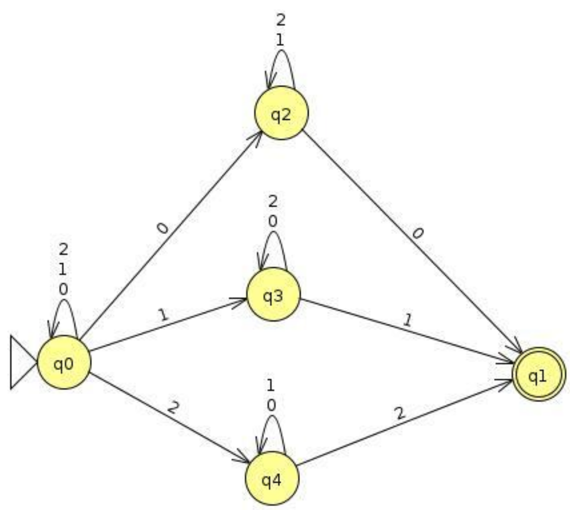


9.1.2 NFA

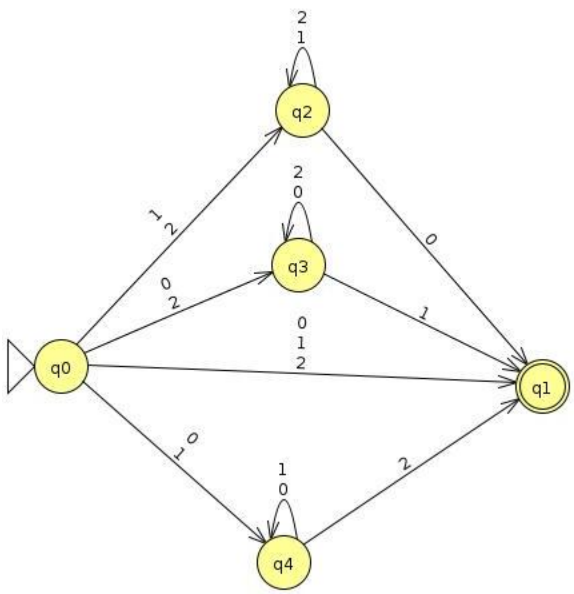
1. Dato l'alfabeto {a,b} realizzare un NFA che riconosce le stringhe che contengono aaa oppure aba
   SI babab,aaaa,aaaba
   NO ϵ,abba,a,b,ab



2. Realizzare un NFA che riconosce le stringhe non vuote sull'alfabeto {0,1,2} in cui l'ultima cifra appare almeno una volta in precedenza
   SI 011,121,22,0120
   NO ϵ,012,20,1



3. Realizzare un NFA che riconosce le stringhe non vuote sull'alfabeto 0,1,2 in cui l'ultima cifra NON appare in precedenza
   SI 012,20,1
   NO ϵ,011,121,22,0120



4. Dato l'alfabeto a,b si consideri l'NFA fatto all'esercizio 1, che riconosce le stringhe che contengono aaa oppure aba. Trasformarlo in DFA.