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.

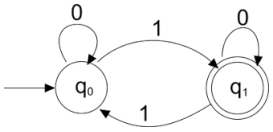


Figura 1: Semplice automa

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".
$\{0^n1^n|n \geq 1\}$ 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

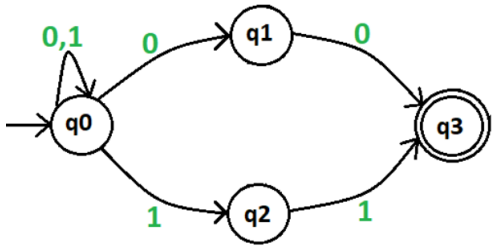


Figura 2: Diagramma di transizione

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

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

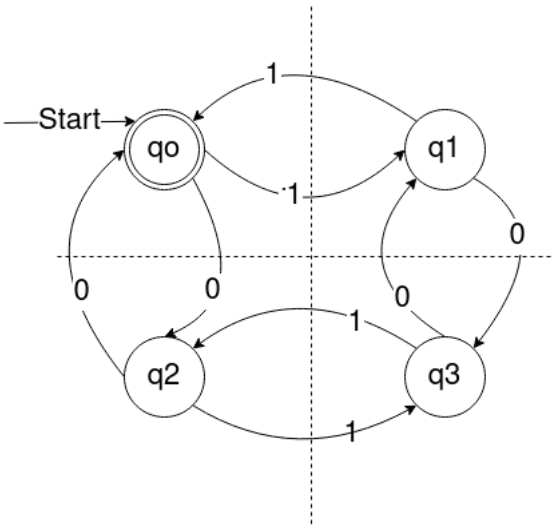


Figura 3: Diagramma

	0	1
$\rightarrow * \mathrm{q}_0$	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:

- $\hat{\delta}(q_0, \epsilon) = \mathrm{q}_0$
- $\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \epsilon), 1) = \delta(q_0, 1) = \mathrm{q}_1$
- $\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_1, 1) = \mathrm{q}_0$
- $\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_1, 11), 0) = \delta(q_0, 1) = \mathrm{q}_2$
- $\hat{\delta}(q_0, 1101) = \delta(\hat{\delta}(q_0, 110), 1) = \delta(q_2, 1) = \mathrm{q}_3$
- $\hat{\delta}(q_0, 11010) = \delta(\hat{\delta}(q_0, 1101), 0) = \delta(q_3, 0) = \mathrm{q}_1$
- $\hat{\delta}(q_0, 110101) = \delta(\hat{\delta}(q_0, 11010), 1) = \delta(q_1, 1) = \mathrm{q}_0$

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

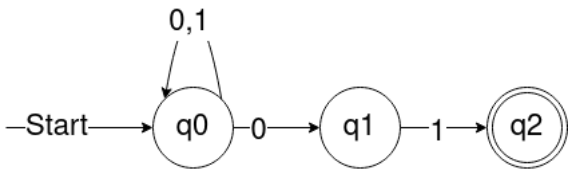


Figura 4: NFA che accetta stringa che finisce con 01

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).

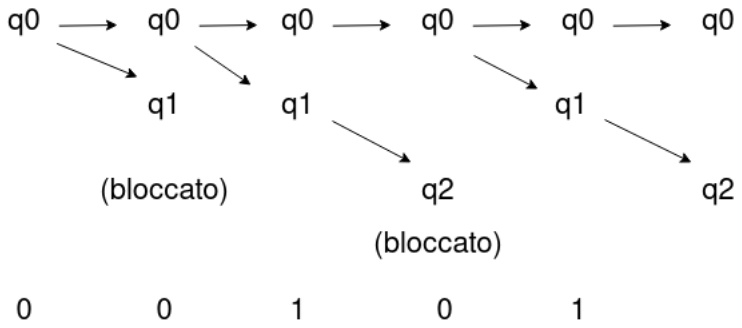


Figura 5: Gli stati del NFA

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.

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.
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:

	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

	0	1
A	A	A
$\rightarrow$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

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.

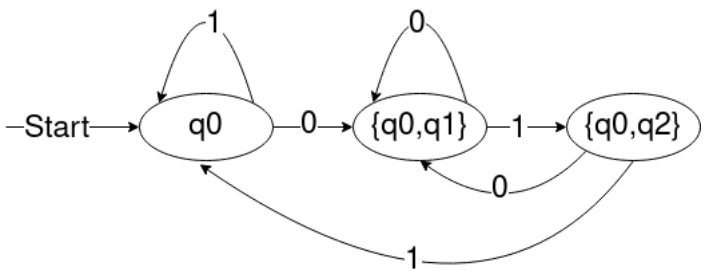


Figura 6: Grafico DFA convertito da NFA

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.

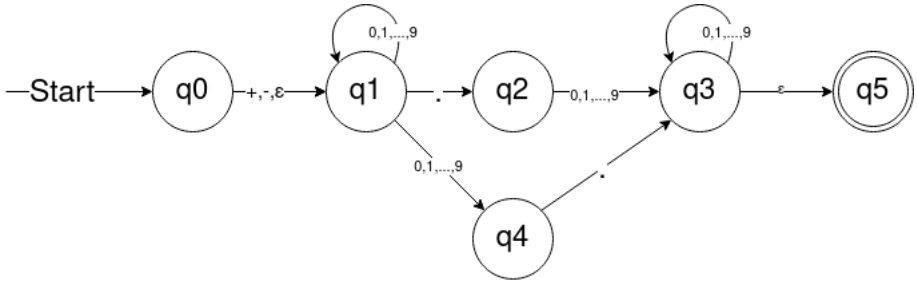


Figura 7: epsilon-NFA che accetta numeri decimali

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·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

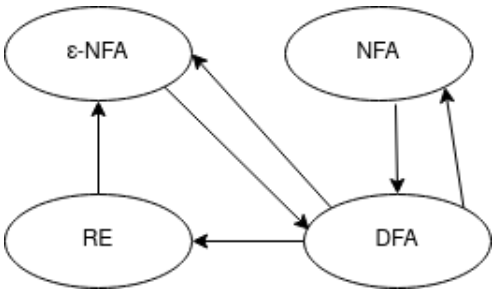


Figura 8: Conversioni

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*.

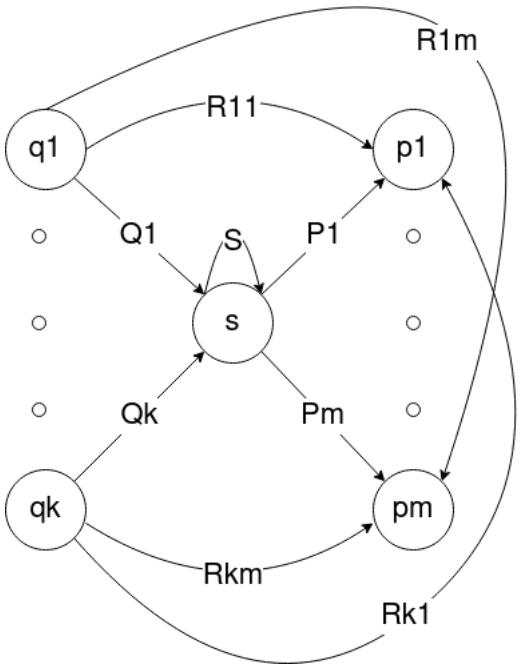


Figura 9: Eliminazione per stati

- 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.

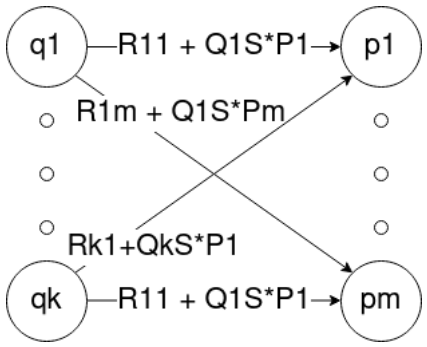


Figura 10: Eliminazione di s

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.

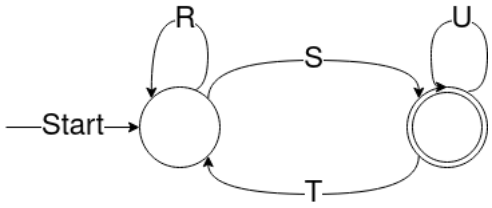


Figura 11: Automa generico 2 stati

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

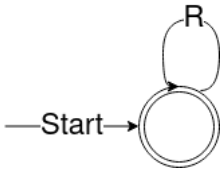


Figura 12: Automa generico 1 stato

Dato un NFA come segue.

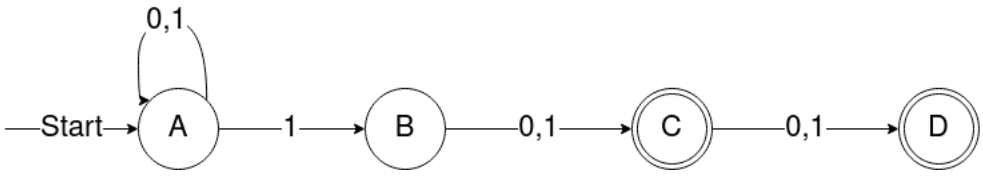


Figura 13: NFA esempio

Esprimiamo le sue funzioni di transizioni come regex

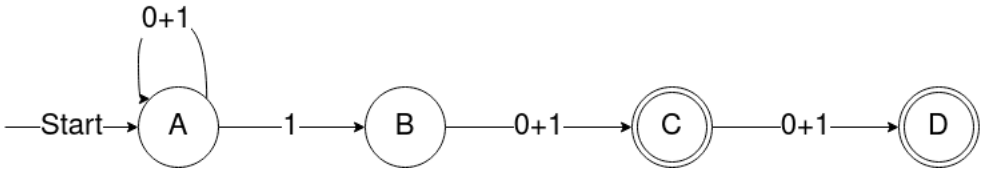


Figura 14: NFA con archi regex

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

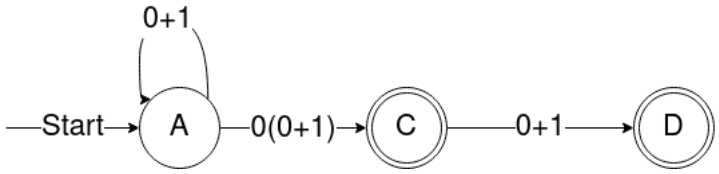


Figura 15: B rimosso

Eliminiamo C

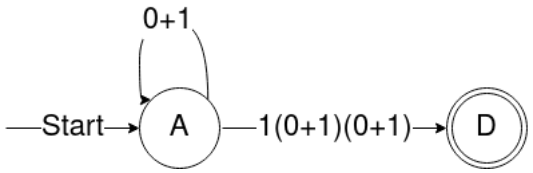


Figura 16: C rimosso

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.

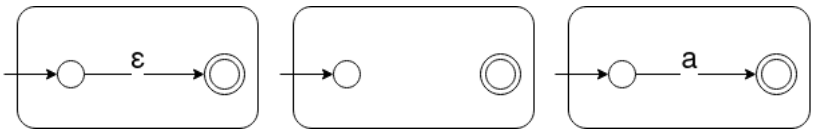


Figura 17: Stati base

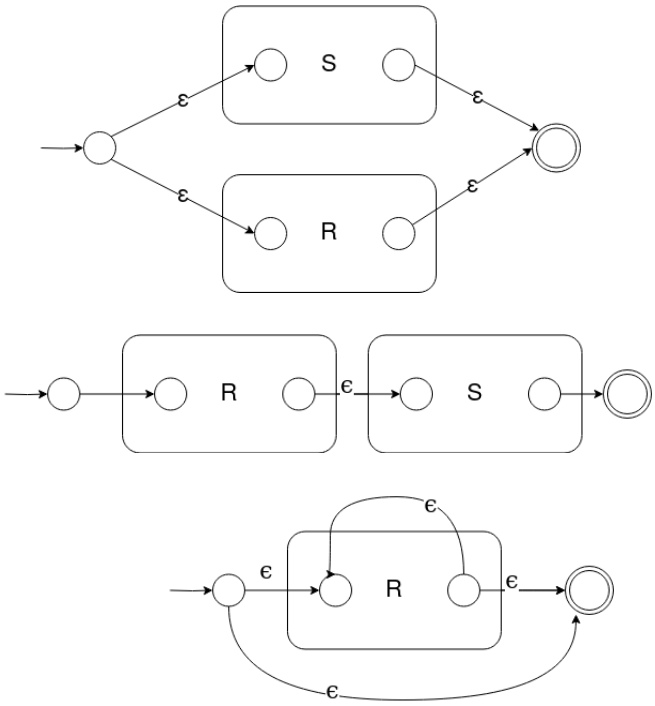


Figura 18: R+S, RS e R*

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)

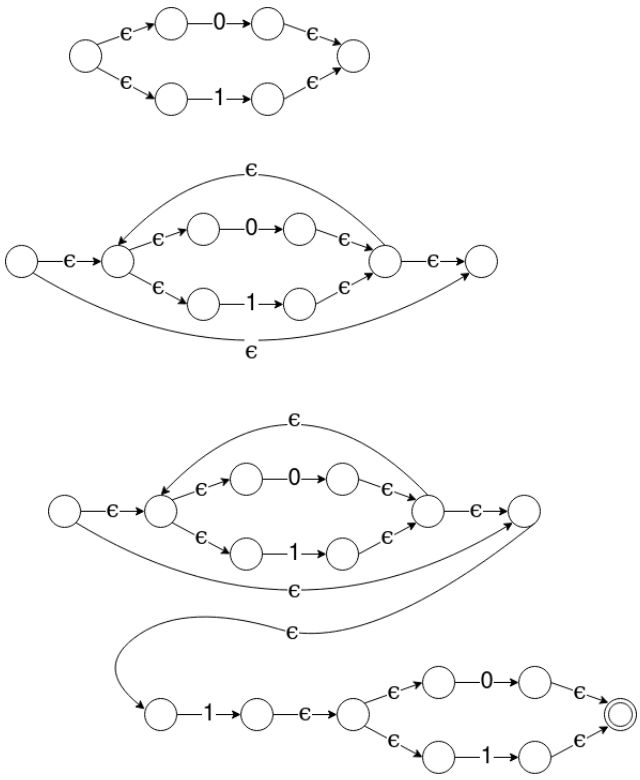


Figura 19: R+S, RS e R*

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. $x = a_{j+1}, a_{j+2}, ..., a_m$

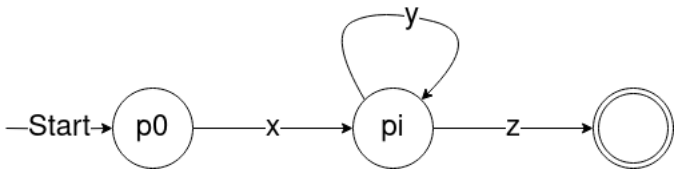


Figura 20: A un certo punto i stati si ripetono

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*: $L \backslash M$
- *Inversione*: $LR = \{wR : w \in L\}$
- *Chiusura*: L^*
- *Concatenazione*: $L \cdot M$

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).

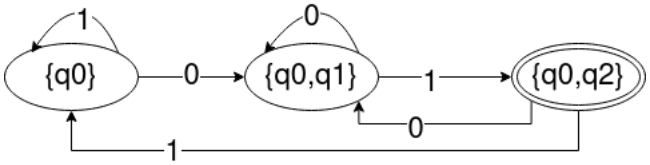


Figura 21: Diagramma di A

Il diagramma di B risulta opposto

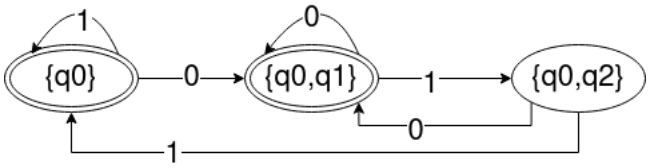


Figura 22: Diagramma di B

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

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

Dimostrazione alternativa Sia L il linguaggio $L = (Q_L, \Sigma, \delta_L, q_0, F_L)$ e M il linguaggio $M = (Q_M, \Sigma, \delta_M, q_0, F_M)$ assumiamo per semplicità che siano dei DFA Se A_L passa da p a s e A_M passa da q a t (sono tutti stati) allora $A_{L \cap M}$ passera da $(p, s)a(q, t)$ quando legge una stringa a.
Formalmente il linguaggio risultante dall'intersezione diventa

$$A = (Q_M \times Q_L, \Sigma, \delta_{M \cap L}, (q_M, q_L), F_L \times F_M)$$

Si può dimostrare che $\hat{\delta}((q_M, q_L), w) = (\hat{\delta}(q_M, w), \hat{\delta}(q_L, w))$, questo perché A accetta solo quando entrambi gli stati sono accettanti, quindi accetta per forza anche l'intersezione.

8.2.1 Chiusura rispetto alla differenza

Teorema Sia L e M dei linguaggi regolari allora anche $L - M$ è un linguaggio regolare

Dimostrazione $L - M = L \cap \overline{M}$, ma sappiamo che $\overline{M}$ è regolare e l'intersezione di 2 linguaggi è regolae, quindi $L - M$ è anche esso regolare.

8.2.2 Inversione

L'inversione di una strina $a_1, a_2, ..., a_n$ è la stringa $a_n, ..., a_2, a_1$ questa stringa la denotiamo come w^R e notiamo che $\epsilon^R = \epsilon$.
Un linguaggio L^R inverso di L presenta tutte le stringe al suo interno inverse. Se L è un linguaggio regolare allora lo è anche L^R.

8.3 Proprietà di decisione

Questi non scontanti che vanno affrontate:

- Un linguaggio descritto è vuoto?
- Una stringa appartiene al linguaggio?
- Due linguaggi equivalenti?

8.3.1 Verificare se un linguaggio è vuoto

Se il linguaggio A è rappresentato da un automa finito posso attraversare tutti i nodi, se trovo uno stato accettante allora il linguaggio **non** è vuoto. Quest'operazione richiede O(n^2) perché è un semplice attraversamento di grafo.
Se iniziamo da un espressione regolare, possiamo trasformarla in un ϵ-NFA e poi effettuare i cammini a costo O(n).
È possibile anche determinare se il linguaggio è vuoto in base alla regex direttamente. Se il linguaggio non ha $\emptyset$ sicuramente non può essere vuoto, altrimenti posso determinarlo ricorsivamente seguendo le regole algebriche delle regex.
Di seguito elenco i casi:

- $R = \emptyset$. $L(\emptyset)$ è vuoto
- $R = \epsilon$. $L(\epsilon)$ **non** è vuoto
- $R = a$ (qualsiasi stringa a) L(a) non è vuoto
- $R = R_1 + R_2$. $L(R)$ è vuoto se sia $L(R_1)$ che $L(R_2)$ siano vuoti
- $R = R_1R_2$. $L(R)$ è vuoto se $L(R_1)$ o $L(R_2)$ è vuoto
- $R = R_1*$. $L(R)$ non è mai vuoto, al massimo è ϵ
- $R = R_1$. $L(R)$ è vuoto solo se $L(R_1)$ è vuoto, sono lo stesso linguaggio

8.3.2 Appartenenza a un linguaggio

Per controllare se una qualsiasi stringa $w \in L(A)$ per un DFA è sufficiente simulare w su A, se $|w| = n$ il tempo risulta O(n).

Se A è un NFA e ha s stati, allora O(ns^2), vale lo stesso epr ϵ-NFA.

Se L=L(R) è una regex, la converto in ϵ-NFA ovvero O(ns^2)

8.3.3 Equivalenza e minimizzazione di automi

Dobbiamo esplorare la possibilità di dire che 2 DFA sono equivalenti, un modo per farlo è minimizzarli, se sono equivalenti basterà cambiare etichette finché non coincidono.
Iniziamo definendo cosa rende equivalenti 2 stati p e q. Data una stringa w, p e q sono equivalenti se $\hat{\delta}(q, w)$ e $\hat{\delta}(p, w)$ sono entrambi accentanti oppure non accentanti.
Nel caso in cui uno sia accentante e l'altro no, allora si dicono distinti. NB. 2 stati equivalenti non ci dicono niente sulla stringa w e non ci dice se i due stati sono lo stesso.
Possiamo raggruppare le nostre distinzioni degli stati in una tabella, tramite l'algoritmo $riempit - tabella$.

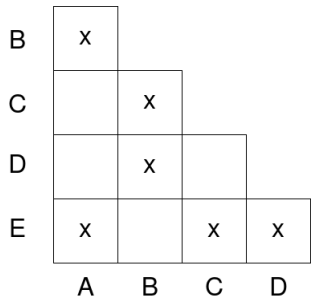


Figura 23: Algoritmo riempi tabella

Ogni x è uno stato distinguibile, un quadrato vuoto significo stati equivalenti.
Per testare se 2 linguaggi L e M sono equivalenti dobbiamo:

- Convertire L e M in DFA

- Costruire il DFA unione dei 2 linguaggi
- Se l'algoritmo dice che i 2 stati iniziali sono equivalenti allora L=M, altrimenti L≠M

Partendo da 2 DFA costruisco un DFA B che ha lo stato iniziale che contiene quello di A e lo stato accentante che contiene quello di A. Ogni altra funzione di un blocco deve essere equivalente.
L'algoritmo non può essere applicato a un NFA.

9 Grammatiche libere da contesto

Esempio informale

Definiamo un linguaggio delle palindrome. Una stringa è palindroma se si legge allo stesso modo in entrambi i versi, come *otto* oppure *madamimadam* (madame I'm Adam). Perciò w è palindroma se $w = w^R$.
Si può facilmente dimostrare che questo linguaggio non è regolare usando il pumping lemma. Scegliamo $w = 0^n 10^n$, scomponiamo in $w = xyz$ tale che y sia fatto di vari 0 e scegliamo k=0, xz dovrebbe adesso appartenere a L_{pal} ma non è così, perché ho meno 0 a sinistra rispetto che a destra.
Posso definire le stringhe che appartengono a L_{pal} in maniera ricorsiva.

Base ϵ, 0 e 1 sono palindrome
Induzione Se w è palindroma allora $0w0$ e $1w1$ sono palindrome

Una **grammatica libera** è una notazione formale per esprimere linguaggi ricorsivamente. Una grammatica consiste in una o più serie di variabili che rappresentano classi di stringhe, ovvero linguaggi.
Ogni classe definisce come costruire le stringhe in ogni classe. Definiamo le classi del linguaggio palindroma:

1. $P \to \epsilon$
2. $P \to 0$
3. $P \to 1$
4. $P \to 0P0$
5. $P \to 1P1$

0 e 1 sono terminali, P è una variabile, P è anche la categoria iniziale, 1-5 sono produzioni.

9.1 Definzione formale di CFG

Un CFG è formato da 4 elementi:

1. Un insieme di simboli detti *terminali*
2. Un insieme di variabili, detti *non termali* oppure *categorie sintattiche*
3. Una variabile detto *simbolo iniziale*
4. Un insieme finito di *produzioni* o *regole* che definiscono il linguaggio ricorsivamente. Ogni produzione consiste in 3 parti
   (a) Una variabile che è definita parzialmente dalla produzione, *testa*
   (b) Il simbolo di produzione $\rightarrow$
   (c) Il *corpo* della produzione, ovvero la stringa o il terminale che la forma. Le stringhe vengono formate sostituendo le variabili.

In maniera contratta, CFG=(V,T,P,S) rispettivamente variabile, terminale, produzioni e simbolo iniziale.
Il linguaggio palindromo descritto come CFG è $G_{pal} = (\{P\}, \{0,1\}, A, P)$ A è l'insieme delle produzioni del linguaggio.

9.2 Derivazione in CFG

Possiamo definire le stringhe tramite concatenazione delle produzione, *inferenza ricorsiva*
Il secondo modo per *derivazione*, ovvero uso le produzioni fino a quanto ho solo simboli terminali. Noi studieremo la derivazione.
Sia $G = (V, T, P, S)$ una grammatica libera e sia αAB una stringa mista di terminali e variabili, dove A è variabile e $\alpha B \in (V \cup T)$ allora se G risulta chiara nel contesto, posso scrivere:

$$\alpha AB \underset{G}{\Longrightarrow} \alpha\gamma B$$

A è una derivazione di G, $A \Rightarrow \gamma$. Posso usare il simbolo * per denotare "zero o più passi"(chiusura transitiva).
Base $\alpha \overset{*}{\underset{G}{\Longrightarrow}} \alpha$, vale per ogni stringa terminale o variabile, ovvero ogni stringa deriva se stessa
Induzione Se $\alpha \overset{*}{\underset{G}{\Longrightarrow}} \beta$ e $\beta \underset{G}{\Longrightarrow} \gamma$ significa che $\alpha \overset{*}{\underset{G}{\Longrightarrow}} \gamma$
Se la grammatica è chiara posso scrivere $\overset{*}{\Rightarrow}$

9.3 Derivazione a sinistra e a destra

È possibile arrivare a diverse conclusioni a seconda della scelta di quali produzioni fare, quindi per evitare di avere incertezze si può scegliere un verso di come derivare ogni volta.
Derivazione a destra: $\underset{rm}{\Longrightarrow}$
Derivazione a sinistra: $\underset{lm}{\Longrightarrow}$

9.4 Linguaggio di una grammatica

Se $G(V, T, P, S)$ è una CFG allora il suo linguaggio è:

$$L(G) = \{w \in T^* : S \overset{*}{\underset{G}{\Longrightarrow}} w\}$$

Ovvero l'insieme delle stringhe T^* derivabili da w

9.5 Alberi sintattici

Data una grammatica $G = (V, T, S, P)$, gli alberi sintattici di G soddisfano i seguenti requisiti:

1. Ogni nodo interno è una variabile V
2. Ogni foglia è un variabili, terminale o ϵ. Quando è ϵ deve essere l'unico figlio
3. Se un nodo A con i figli $X_1, X_2, ..., X_k$ allora $A \to X_1, X_2, ..., X_k$ è una produzione in P. X può essere ϵ solo nel caso in qui $A \to \epsilon$

Nella grammatica seguente:

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$

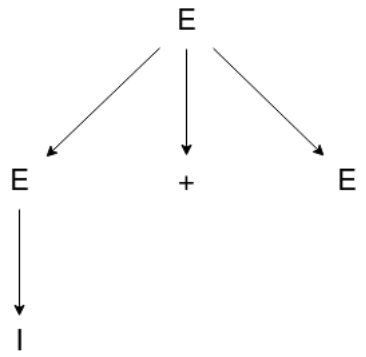


Figura 24: Parsing tree

Questo albero sintattico rappresenta la produzione $E \overset{*}{\Rightarrow} I + E$

9.5.1 Prodotto di un albero sintattico

Se concateniamo le foglie di un albero otteniamo una stringa, ovvero il *prodotto*. Inoltre la radice deve essere il simbolo iniziale e ogni foglia è $\emptyset$ oppure un terminale.

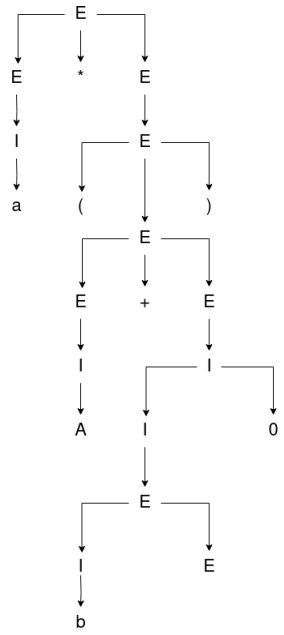


Figura 25: Produzione

Il risultato della produzione è $a * (a + b00)$

9.5.2 Inferenza, derivazione e alberi sintattici

Sia $G = (V, T, P, S)$ una CFG e $A \in V$ allora i seguenti sono equivalenti:

1. $A \overset{*}{\Rightarrow} w$
2. $A \underset{lm}{\overset{*}{\Longrightarrow}} w$
3. $A \underset{rm}{\overset{*}{\Longrightarrow}} w$
4. C'è un albero sintattico G con radice A e prodotto w

Costruiamo il prodotto della fig. 25 per derivazione sinistra:

- $E \underset{lm}{\Longrightarrow} E * E \underset{lm}{\Longrightarrow}$
- $I * E \underset{lm}{\Longrightarrow}$

- $a * E \underset{lm}{\Longrightarrow}$
- $a * (E) \underset{lm}{\Longrightarrow}$
- $a * (E + E) \underset{lm}{\Longrightarrow}$
- $a * (I + E) \underset{lm}{\Longrightarrow}$
- $a * (a + E) \underset{lm}{\Longrightarrow}$
- $a * (a + I) \underset{lm}{\Longrightarrow}$
- $a * (a + I0) \underset{lm}{\Longrightarrow}$
- $a * (a + I00) \underset{lm}{\Longrightarrow}$
- $a * (a + b00) \underset{lm}{\Longrightarrow}$

9.6 Ambiguità in grammatiche e linguaggi

Nella grammatica:

- $E \to I$
- $E \to E + E$
- $E \to E * E$
- $E \to (E)$

L'espressione $E+E*E$ ha due possibili derivazioni: $E \Rightarrow E+E \Rightarrow E+E*E$ oppure $E \Rightarrow E * E \Rightarrow E + E * E$, da qui i due alberi sintattici

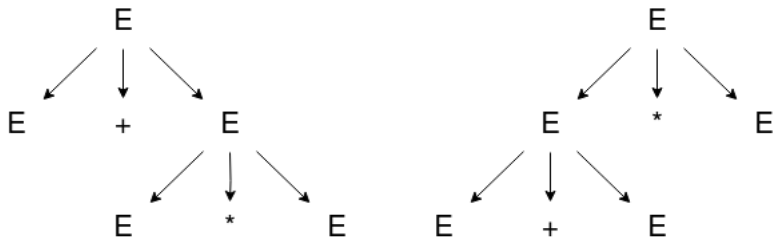


Figura 26: Semplice automa

Queste sono 2 operazioni diverse, usiamo i valori $1+2*3$, dal primo albero troviamo $1 + (2 * 3) = 7$ mentre dal secondo albero troviamo $(1 + 2) * 3 = 9$.
Definizione Sia $G = (V, T, P, S)$ una CFG. Diciamo che G è ambigua se esiste almeno una stringa T* che ha più di un albero sintattico.

9.6.1 Rimuovere ambiguità

È possibile, in alcuni casi, rimuovere l'ambiguità. Non esiste però un modo sistematico e alcuni CFL hanno solo CFG ambigue.
Studiamo la grammatica

$$E \to I \mid E + E \mid E * E \mid (E)$$

$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

Dobbiamo decidere:

1. Chi ha precedenza tra + e *
2. Come raggruppare un operatore

Una soluzione è l'introduzione di una gerarchia di variabili:

1. espressioni E: composizione di uno o più termini T tramite +
2. termini T: composizione di uno o più fattori F tramite *
3. fattori F:
   (a) identificatori I
   (b) espressioni E racchiuse tra parentesi

Ogni più è costretto a essere una T, ogni T può generare E solo chiuse nelle parentesi, questo significa che * ha precedenza rispetto al +.
La seguente grammatica risulta quindi non ambigua:

1. $E \to T|E+T$
2. $E \to F|T*F$
3. $F \to I|(E)$
4. $E \to a|b|Ia|Ib|I0|I1$

9.6.2 Ambiguità inerente

Un CFL è inerentemente ambiguo se tutte le grammatiche per L sono ambigue.

$$L = \{a^n b^n c^m d^m : n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n : n \geq 1, m \geq 1\}$$

Il linguaggio è inerentemente ambiguo.

10 Automi a pila

Un automa a pila (PDA - pushdown automaton) è un ϵ-NFA con una pila che è la sua memoria.
Durante le transizione:

1. Consuma un simbolo di input o esegue una transizione ϵ
2. Va in un nuovo stato o rimane dov'è
3. Rimpiazza la cima della pila con una stringa (lo stack è cambiato), con ϵ (c'è stato un pop) oppure con lo stesso simbolo (nessun cambiamento)

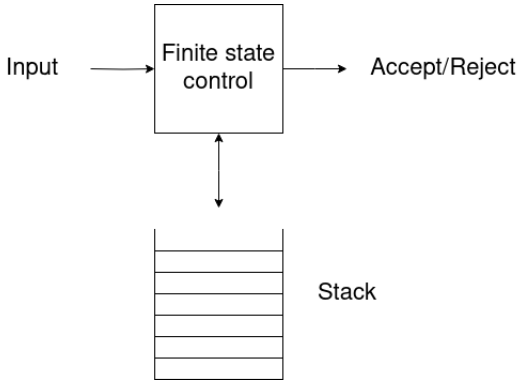


Figura 27: PDA, automa con pila

Esempio: Consideriamo il linguaggio palindromo:

$$L_{wwr} = \{ww^R : w \in \{0,1\}^*\}$$

Una PDA per L_{wwr} ha 3 stati:

1. In q_0 si presume l'input non sia esaurito, quindi leggo 1 alla volta tutti i simboli di input e li accumulo nello stack.
2. Scommettiamo di aver trovato la sequenza corretta, quindi passiamo in q_1 ma continuando a leggere input
3. Confronta la cima della pila con il simbolo in q_1, se sono uguali consumiamo l'elemento, altrimenti il ramo muore
4. Se lo stack è vuoto passiamo in q_2

10.1 Definizione formale PDA

Un PDA è una tupla di 7:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

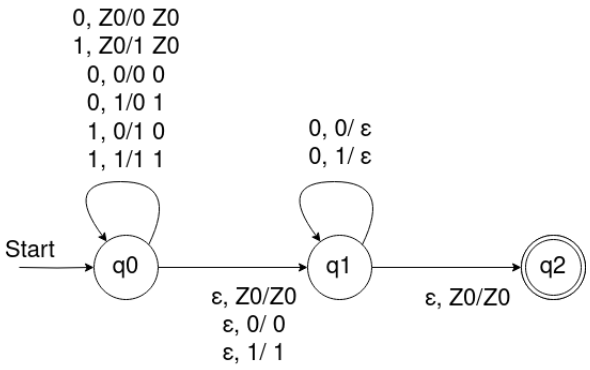
dove

- Q è un insiemi di stati finiti
- Σ è un alfabeto finito di input
- Γ è un alfabeto finito di pila
- (δ è una funzione di transizione da $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ a sottoinsieme di $Q \times \Gamma^*$ - definizione del prof, poco chiara)
  δ è un funzione di transizione che prende 3 input (q, a, X) dove:
  - q è uno stato in Q
  - a è un simbolo in Σ oppure la ϵ, NB ϵ non fa parte dell'input
  - X è un simbolo in Γ, ovvero è un simbolo **sullo stack**

  L'output di σ è una coppia (p, γ), dove p è uno stato e γ è una nuova stringa che rimpiazza X sullo stack. Se $\gamma = \epsilon$ X viene eliminato, se $\gamma = X$ non cambia nulla e se $\gamma = YZ$ allora Y sostituisce X e Z viene aggiunto allo stack
- q_0 stato iniziale
- $Z_0 \in \Gamma$ simbolo iniziale per la pila
- $F \subseteq Q$ è l'insieme degli stati di accettazione

Prendiamo come esempio il PDA di L_{wwr}:

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{Z_0, 0, 1\}, \delta, q_0, Z_0, \{q_2\}$$



dove δ è definita dalle seguenti regole:

1. $\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$ $\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$ una di queste regole è applicata all'inizio, l'input viene inserito lasciando Z_0 come indice del fondo.

2. $\delta(q_0, 0, 0) = \{(q_0, 00)\}$, $\delta(q_0, 0, 1) = \{(q_0, 01)\}$,
   $\delta(q_0, 1, 0) = \{(q_0, 10)\}$, $\delta(q_0, 1, 1) = \{(q_0, 11)\}$
   leggo un input, lo salvo nello stack e continuo a leggere

3. $\delta(q_0, \epsilon, Z_0) = \{(q_1, Z_0)\}$,
   $\delta(q_0, \epsilon, 0) = \{(q_1, 0)\}$
   $\delta(q_0, \epsilon, 1) = \{(q_1, 1)\}$
   passiamo da q_0 a q_1 lasciando intantto lo stack

4. $\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$,
   $\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$
   rimaniamo su q_1 ma eliminiamo un membro dallo stack, se sono uguali

5. $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$ se non ho niente nello stack, passo alla soluzione accettante

10.2 Descrizioni istantanee

Un PDA passa da una configurazione all'altra consumando un simbolo in input oppure la cima della stack.
Possiamo rappresentare una configurazione tramite *descrizioni istantanee* (ID) che sono un tupla (q, w, γ) dove: q è lo stato, w è l'input rimanente e γ è il contenuto della pila.
Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA, allora $\forall w \in \Sigma^*, \beta \in \Gamma^* : (p, \alpha) \in \delta(q, a, X) \Rightarrow (q, aw, X\beta) \vdash (p, w, \alpha, \beta)$

Il simbolo $\vdash$ significa "deduzione logica", mentre definiamo $\overset{*}{\vdash}$ come la chiusura transitiva di $\vdash$

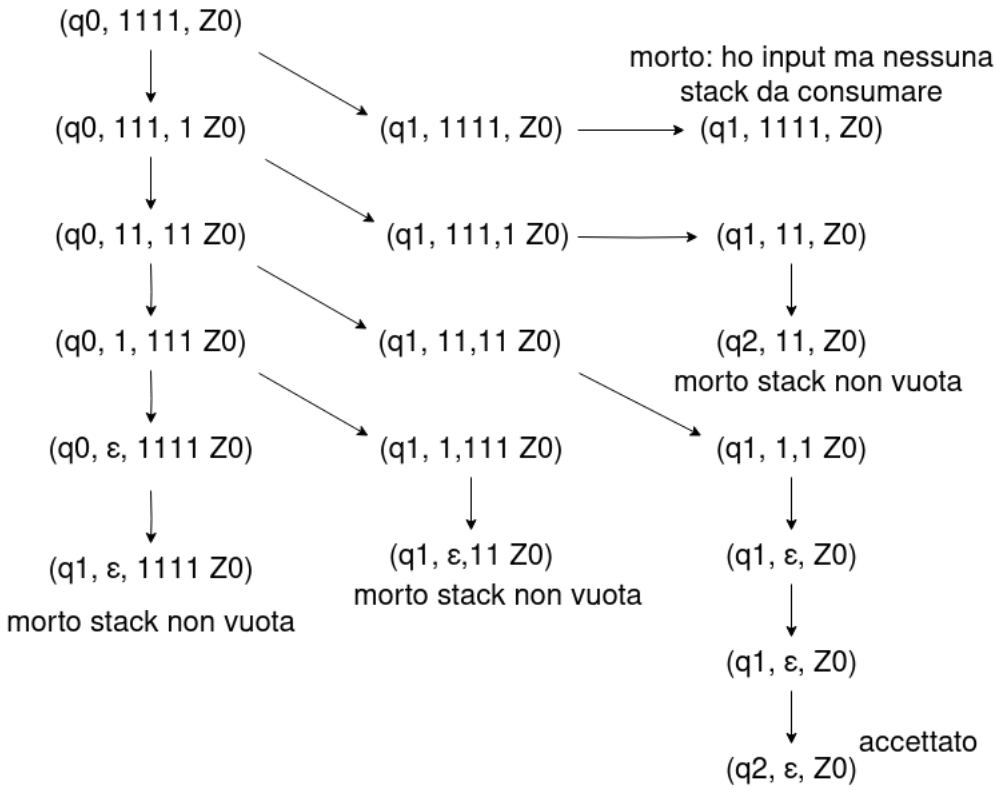


Figura 28: Diagramma PDA

10.3 Accettazione per stato finale

Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA, il *linguaggio accettato* da P per stato finale è:

$$L(P) = \{w : (q_0, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \alpha, q \in F)\}$$

In altre parole una volta che w è in uno stato accettante il contenuto della pila è irrilevante

10.4 Accettazione per pila vuota

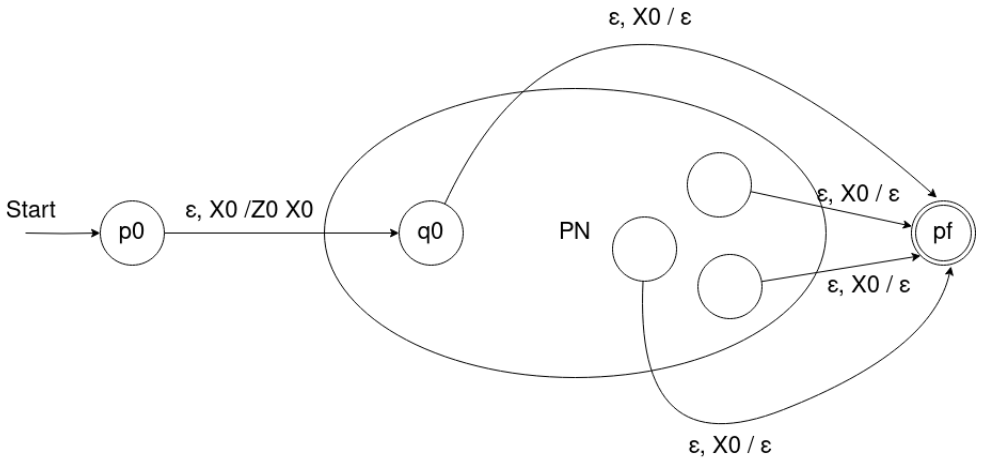
Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA, il *linguaggio accettato* da P per pila vuota è:

$$N(P) = \{w : (q_0, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \epsilon)\}$$

NB q è un generico stato, dunque $N(P)$ è degli input w che svuotano la stack

10.5 Da stack vuota a stato finale

Dimostriamo ora che per pila vuota o per stato finale sono linguaggi equivalenti.
Teorema: Se $L = N(P_N)$ per un PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ allora $\exists$ PDA P_F tale che $L = N(P_F)$.
A parole: L'idea è avere un simbolo nuovo X_0 che specifica quando arrivano allo stato finale sia P_F che P_N. p_0 ci server per inserire il primo simbolo nello stack ed andare in q_0. Andiamo in p_F quando P_N svuota la stack.



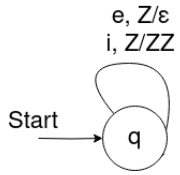
Dimostrazione Sia:

$$P_F = \{Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\}\}$$

dove δ è definita come:

1. $\delta(p_0, \epsilon, X_0) = \{(q_0, Z_0, X_0)\}$ transazione spontanea da P_F a P_N
2. $\forall q \in Q, a \in \Sigma \cup \{\epsilon\}, Y \in \Gamma : \delta_F(q, a, Y) = \delta_N(q, a, Y)$
3. $\delta_F(q, a, Y)$ contiene (p_j, ϵ) per ogni q in Q

Consideriamo un automa *if else* in C, dobbiamo leggere tanti if quanti else quindi usiamo Z per contare la differenza tra i 2 simboli.



Leggendo i inseriamo una Z, leggendo e rimuoviamo una Z.Formalmente:

$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$$

dove δ:

1. $\delta(q, i, Z) = \{(q, ZZ)\}$ leggo i aggiungo Z
2. $\delta(q, e, Z) = \{(q, \epsilon)\}$ leggo e rimuovo Z

A partire da P_N costruiamo P_F:

$$P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\})$$

dove δ:

1. $\delta_F(p, \epsilon, X_0) = \{(q, ZX_0)\}$ simulo lettura primo simbolo, X_0 va in fondo allo stack
2. $\delta_F(q, i, Z) = \{(q, ZZ)\}$ leggo i aggiungo Z
3. $\delta_F(q, e, Z) = \{(q, \epsilon)\}$ leggo e rimuovo z
4. $\delta_F(q, \epsilon, X_0) = \{(r, \epsilon)\}$ P_F accetta quando P_N si svuota

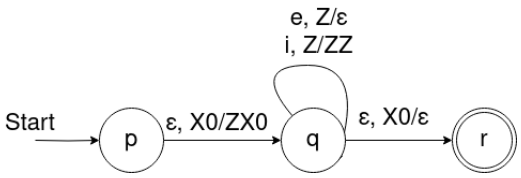


Figura 29: Diagramma P_F

10.6 Da stato finale a pila vuota

Facciamo il percorso inverso $P_F \to P_N$. Si aggiunge un ϵ-transizione da ogni stato accettante di P_F a un nuovo stato p. Quando siamo in p, consumo lo stack senza leggere input.
Per evitare di svuotare lo stack per stringhe non valide uso X_0 come indicatore di fondo. Il nuovo stato p_0 serve solo ad arrivare allo stato iniziale q_0 e mettere X_0 in fondo allo stack.
Teorema: Se $L = N(P_F)$ per un PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$ allora $\exists$ PDA P_N tale che $L = N(P_N)$.
Dimostrazione:

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

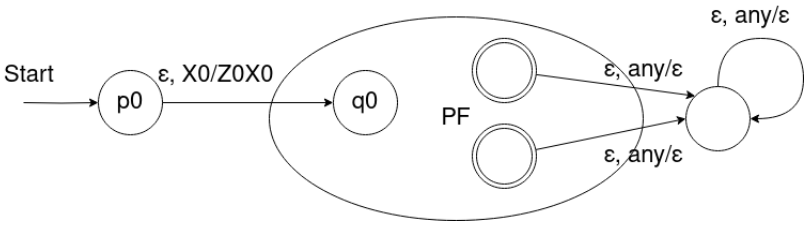
dove δ_N è:

1. $\delta_N(p, \epsilon, X_0) = \{(q, ZX_0)\}$ simulo lettura primo simbolo, X_0 va in fondo allo stack

2. $\forall$ q in Q, a in Σ e ogni Y in Γ: $\delta_N(q, a, Y)$ contiene tutte le coppie in $\delta_F(q, a, Y)$. Ovvero P_N simula P_F.

3. $\forall$ q in F e ogni Y in Γ: $\delta(q, a, Y)$ contiene (p, ϵ). Ogni volta che P_F accetta P_N può svuotare lo stack

4. $\forall$ Y in Γ: $\delta(q, a, Y) = \{p, \epsilon\}$ quando arrivo a p, quindi P_F ha accettato, P_N svuota lo stack senza leggere input



10.7 Equivalenza tra PDA e CFG

Un linguaggio è generato da una CFG se e solo se è accettato da un PDA per pila vuota se e solo se è accettato da un PDA per stato finale

10.8 Da CFG a PDA

Data una CFG costruiamo una PDA che simula la derivazione a sinistra. Ogni espressione non terminale si può scrivere come $xA\alpha$, dove A è la variabile più a sinistra, x sono gli simboli terminali a sinistra di A e α sono le variabili alla destra di A.
Chiamiamo $A\alpha$ la coda della forma/espressione. Una coda di terminali è da considerarsi ϵ. Dobbiamo simulare un PDA, dove accettiamo una stringa terminale w. La coda di $xA\alpha$ compare sullo stack con A in cima, x è quello che consumiamo per aggiungere una stack, $w = xy$.
Quindi data la transazione $(q, y, A\alpha)$, che rappresenta $xA\alpha$, l'automa sceglie di espandere $A \to \beta$. β va in cima allo stack entrando nella ID $(q, y, \beta\alpha)$, il PDA ha un unico stato q.
Non è detto però che β sia terminale, e potrebbe avere dei terminali che lo precedono, quindi dobbiamo eliminare ogni terminale all'inizio di $\beta\alpha$. Confronto i terminali con i successi input per verificare correttezza, altrimenti il processo muore.
Nel caso in cui tutto vada bene, lo stack è vuoto e abbiamo la w corretta.
Formalmente: Sia $G = (V, T, Q, S)$ una CFG, costruiamo il PDA P che accetta $L(G)$ per stack vuoto:

$$P = (\{q\}, T, V \cup T, \delta, q, S)$$

dove δ è definita come segue

$$\delta(q, \epsilon, A) = \{(q, \beta) : A \to \beta \in Q\}$$

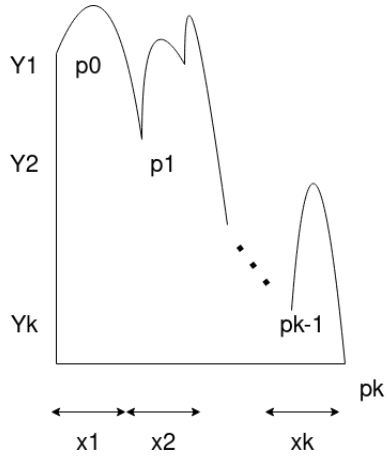
per ogni terminale a, $\delta(q, a, a) = \{(q, \epsilon)\}$
Esempio:
Considero la grammatica $S \to \epsilon|SS|iS|iSe$. Il PDA corrispondente è:

$$P = (\{q\}, \{i, e\}, \{S, i, e\}, \delta, q, S)$$

dove $\delta(q, \epsilon, S) = \{(q, \epsilon), (q, SS), (q, iS), (q, iSe)\}, \delta(q, i, i) = \{(q, \epsilon)\}$
$\delta(q, e, e) = \{(q, \epsilon)\}$.

10.9 Da PDA a CFG

Per ogni PDA P possiamo costruire un CFG G il cui linguaggio coincide con quello accettato dallo stack vuoto di P. Dobbiamo prendere atto del momento più importante di una PDA, ovvero quando viene fatto il pop di un elemento dello stack, perché è cambiato lo stato. Nel nostro caso server tenere traccia dello stato passato. Ogni stato che cambia "scendo" nello stack.



Nel grafico si può notare che ogni eliminazione $Y_1, Y_2, ..., Y_k$ coincide con la lettura di un input x. Non importa quanti passaggi siano stati fatti, qua viene visto lo step finale, quando Y esce.
Inoltre viene visto anche il passaggio di stato, che anche esso coincide con l'eliminazione.
Per costruire una CFG a partire dal PDA usiamo variabile che rappresentano un "evento" con due parti:

1. l'eliminazione definitiva dallo stack di X
2. il passaggio da p a q, dopo che scambio X con ϵ sullo stack

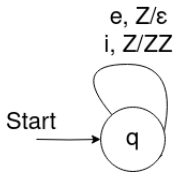
Questa variabile la rappresentiamo come $[pXq]$.
Formalmente: Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ un PDA.
Definiamo $G = (V, \Sigma, R, S)$ con:

- $V = \{[pXq] : \{p, q\} \subseteq Q, X \in \Gamma\} \cup \{S\}$ ovvero V contiene il simbolo iniziale S e tutti i simboli in Q e in Γ

- $R = \{S \to [q_0 Z_0 p] : p \in Q\} \cup$
  $\{qXr_k] \to a[rY_1r_1]...[r_{k-1}Y_kr_k] :$
  $a \in \Sigma \cup \{\epsilon\},$
  $\{r_1, ..., r_k\} \subseteq Q,$
  $(r, Y_1Y_2...Y_k) \in \delta(q, a, X)\}$

  - $[q_0 Z_0 p]$ genera tutte le stringhe w e passa dallo stato q_0 a p. Al termine dell'operazione lo stack P sarà svuotato
  - $\{qXr_k] \to a[rY_1r_1]...[r_{k-1}Y_kr_k]$: è una produzione che indica il passaggio q a r_k e il susseguirsi di passaggi per arrivarci. Al primo passaggio leggo l'input a e itero k volte finché non ho eliminato Y. a può essere ϵ
  - se k=0 allora $Y_1Y_2...Y_k = \epsilon$ e $r_k = r$

Esempio: Convertiamo



$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$$

dove

- $\delta(q, i, Z) = \{(q, ZZ)\}$
- $\delta(q, e, Z) = \{(q, \epsilon)\}$

in una grammatica

$$G = (V, \{i, e\}, R, S)$$

dove

- $V = \{[qZq], S\}$
- $R = \{S \to [qZq], [qZq] \to i[qZq[qZq], [qZq] \to e\}$

Per semplicità $[qZq] = A$ quindi $S \to A$ e $A = iAA|e$

Esempio: Convertiamo

$$P = \{p, q\}, \{0, 1\}, \{X, Z_0\}, \delta, q, Z_0)$$

dove δ è data da:

1. $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$
2. $\delta(q, 1, X) = \{(q, XX)\}$
3. $\delta(q, 0, X) = \{(p, X)\}$
4. $\delta(q, \epsilon, X) = \{(q, \epsilon)\}$
5. $\delta(p, 1, X) = \{(p, \epsilon)\}$
6. $\delta(p, 0, Z_0) = \{(q, Z_0)\}$

in una CFG.
Otteniamo $G = (V, \{0, 1\}, R, S$, dove

$$V = \{[qZ_0q], [pZ_0q], [qZ_0p], [pZ_0p], [qXq], [pXq], [qXp], [pXp], \}$$

e le produzioni in R

$$S \to [qZ_0q][qZ_0p]$$

Dalla transizione (1) $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$ si ha:
$[qZ_0q] \to 1[qXq][qZ_0q]$
$[qZ_0q] \to 1[qXp][pZ_0q]$
$[qZ_0p] \to 1[qXq][qZ_0p]$
$[qZ_0p] \to 1[qXp][pZ_0p]$

Dalla transizione (2) $\delta(q, 1, X) = \{(q, XX)\}$
$[qXq] \to 1[qXq][qXq]$
$[qXq] \to 1[qXp][pXq]$
$[qXp] \to 1[qXq][qXp]$
$[qXp] \to 1[qXp][pXp]$

Dalla transizione (3) $\delta(q, 0, X) = \{(p, X)\}$
$[qXq] \to 0[pXq]$
$[qXp] \to 0[pXp]$

Dalla transizione (4) $\delta(q, \epsilon, X) = \{(q, \epsilon)\}$
$[qXq] \to \epsilon$

Dalla transizione (5) $\delta(p, 1, X) = \{(p, \epsilon)\}$
$[pXp] \to 1$

Dalla transizione (6) $\delta(p, 0, Z_0) = \{(q, Z_0)\}$
$[pZ_0q] \to 0[qZ_0q]$
$[pZ_0p] \to 0[qZ_0p]$

11 PDA deterministici

I PDA deterministici sono quelli usati per i parser, sono un sotto caso utile per noi.
Un PDA deterministico non deve avere mosse alternative, se $\delta(q, a, X)$ ha più di una coppia sicuramente non è deterministico. Questo però non è sufficiente però, perché potresti avere due prodotti diversi e bisogna scegliere.
Definiamo quindi un PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ deterministico se e solo se:

1. $\delta(q, a, X)$ ha al massimo un elemento $\forall q \in Q, a \in \Sigma, X \in \Gamma$
2. Se $\delta(q, a, X)$ non è vuoto per un $a \in \Sigma$ allora $\delta(q, \epsilon, X)$ deve essere vuoto

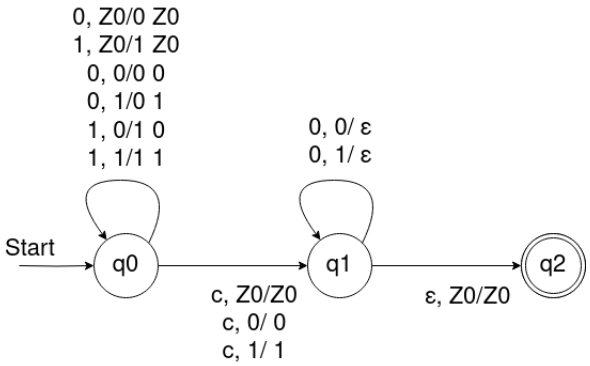


Figura 30: Esempio di PDA deterministico (DPDA)

11.1 DPA che accettano per stato finale

Regex $\subset$ L(DPDA) $\subset$ CFL.
Teorema Se L è regolare allora L=L(P) per un qualsiasi DPDA P.
Prova: Dato che L è regolare $\exists$ DFA A tale che L=L(A)

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

definiamo il DPDA

$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$$

dove

$$\sigma_P(q, a, Z_0) = \{(\delta_A(q, a), Z_0\}$$

$\forall\ p, q \in Q$ e $a \in \Sigma$
applichiamo un induzione su $|w|$

$$(q_0, w, Z_0) \overset{*}{\vdash} (p, \epsilon, Z_0) \iff \hat{\delta}(q_0, w) = p$$

11.2 DPDA che accettano per la pila vuota

Si usa la **proprietà del prefisso**: un linguaggio ha la proprietà del prefisso se NON esistono due stringhe uguali una che è il prefisso dell'altra.
L_{pal} ha la proprietà del prefisso
$\{0\}^*$ non ha la proprietà del prefisso
Teorema L è N(P) per un qualche DPDA P se e solo se L ha la proprietà del prefisso e L è $L(P')$ per qualche DPDA P'

11.3 DPDA e non ambiguità

L(DPDA) non per forza coincide con CFL non ambigue, viceversa invece è vero.

Theorem 1 *Se $L = N(P)$ per qualche DPDA P, allora L ha una grammatica non ambigua*

Dimostrazione stessa dimostrazione che fai da PDA a CFG, solo che parti con un DPDA

L'enunciato può essere rafforzato

Theorem 2 *Se $L = L(P)$ per qualche DPDA P, allora L ha una grammatica non ambigua*

Dimostrazione
Sia \$ un simbolo non presente in L, e sia $L' = L\$$. In altri termini L' sono le stringhe L seguite da \$. Dal Teorema. 1 L' ha la proprietà di prefisso. Dal Teorema. 2 esiste una grammatica G' che genera un linguaggio $N(P')$ che è L'.
Costruiamo quindi una grammatica G tale che $L(G) = L$, dobbiamo solo togliere \$ dalla fine delle stringhe, così che G' coincida con G. Introduciamo \$ in G

$$\$ \rightarrow \epsilon$$

Visto che $L(G') = L'$ consegue che $L(G) = L$. G non è ambigua.

12 Proprietà di CFG

Elenco delle proprietà:

- *Semplificazione* di una CFG. Una CFL ha una grammatica in forma speciale
- *Pumping Lemma* per CFG, simile alle regex
- *Proprietà di chiusura.*
- *Proprietà di decisione.* Verifichiamo l'appartenenza e l'essere vuoto

12.1 Forma normale di Chomsky

Ogni CFL (senza ϵ) è generato da una CFG dove tutte le forme sono

$$A \to BC, A \to a$$

dove A, B, C sono variabili, mentre a è un terminale. Questa è detta forma di Chomsky e per ottenerla dobbiamo pulire la grammatica:

- Eliminare i *simboli inutili*, ovvero ogni simbolo che non appartiene a $S \overset{*}{\Rightarrow} w$
- Eliminare le *produzioni* ϵ, dalla forma $A \to \epsilon$
- Eliminare le *produzioni unità* ovvero le produzioni $A \to B$

12.2 Eliminazione di simboli inutili

Un simbolo X è *utile* per una grammatica $G = (V, T, P, S)$ se esiste una derivazione

$$S \underset{G}{\overset{*}{\Rightarrow}} \alpha X \beta \underset{G}{\overset{*}{\Rightarrow}} w$$

per una stringa $w \in T^*$. Possiamo eliminare i simboli inutili senza cambiare il significato della grammatica. Un simbolo utile deve avere 2 caratteristiche:

1. X è un *generatore* se esiste una stringa w tale che $X \overset{*}{\Rightarrow} w$. Ogni terminale è un generatore di se stesso in 0 passi

2. X è *raggiungibile* se esiste una derivazione $S \stackrel{*}{\Rightarrow} \alpha X \beta$ per qualche $\{\alpha, \beta\} \subseteq (V \cup T)^*$

Un simbolo deve essere sia raggiungibile che generatore, eliminando prima i non generatori e poi i non raggiungibili abbiamo una grammatica solo di simboli utili.
Esempio: Consideriamo la grammatica

$$S \to AB|a$$

$$A \to b$$

A genera b, S genera A, B non è un simbolo generatore. Eliminando B:

$$S \to a$$

$$A \to b$$

Adesso solo a è raggiungibile quindi

$$S \to a$$

che è la nostra grammatica iniziale
NB se elimino prima i simboli non raggiungibili non cambia nulla, tutto è raggiungibile

$$S \to AB|a$$

$$A \to b$$

ora elimino B

$$S \to a$$

$$A \to b$$

e ho una grammatica con simboli inutili

12.3 Eliminazione delle produzioni epsilon

Se L è un CFL, allora L $\{\epsilon\}$ ha una grammatica priva di produzioni ϵ.
La variabile A è annullabile se $A \stackrel{*}{\Rightarrow} \epsilon$

Sia a annullabile, rimpiazzo

$$B \to \alpha A \beta$$

con

$$B \to \alpha\beta$$

indico con n(G) l'insieme dei simboli annullabili della grammatica $G = (V, T, P, S)$.
Esempio Sia la grammatica

$$S \to AB, A \to aAA|\epsilon, B \to bBB|\epsilon$$

Abbiamo $n(G) = \{A, B, S\}$, la prima regola diventa

$$S \to AB|A|B$$

la seconda regola diventa

$$A \to aAA|aA|aA|a$$

la terza regola diventa

$$B \to bBB|bB|bB|b$$

La nuova grammatica sarà:

$$S \to AB|A|BA \to aAA|aA|aB \to bBB|bB|b$$

12.4 Eliminazione produzioni unità

$$A \to B$$

è una produzione unità nel caso in cui A e B siano variabili. Le produzioni variabili sono eliminabili.
Data la grammatica
$E \to T|E + T$
$T \to F|T * F$
$F \to I|(E)$
$I \to a|b|Ia|Ib|I0|I1$
le produzioni unità sono $E \to T, T \to F, F \to I$
Considero la produzione $E \to T$ e la espando

$$E \to F, E \to T * F$$

poi espando $E \to F$

$$E \to I, (E), T * F$$

Infine espando $E \to I$

$$E \to a|b|Ia|Ib|I0|I1|(E)|T * F$$

lo stesso applico lo stesso procedimento con $T \to F$ e $F \to I$:

$$T \to a|b|Ia|Ib|I0|I1|(E)$$

$$F \to a|b|Ia|Ib|I0|I1$$

la nuova grammatica sarà
$E \to a|b|Ia|Ib|I0|I1|(E)|T * F|E + T$
$T \to a|b|Ia|Ib|I0|I1|(E)|T * F$
$F \to a|b|Ia|Ib|I0|I1|(E)$
$I \to a|b|Ia|Ib|I0|I1$
Questo procedimento non funziona nel caso in cui ci siano dei cicli. Se trovo un unità già espansa la rimuovo.

Esempio Si consideri la grammatica
$A \to B|a$
$B \to C|b$
$C \to A|c$
Espando $A \to B$
Da $A \to B$ ottengo

$$A \to C|b$$

Da $A \to C$ ottengo

$$A \to A|c|b$$

Da $A \to A$ ottengo

$$A \to B\ a|c|b$$

Qui mi fermo perché tornei a fare le stesse produzioni, mi rimane:

$$A \to a|c|b$$

eseguo per le altre 3 e la nuova grammatica risulta

$$A \to a|b|c$$

$$B \to a|b|c$$

$$C \to a|b|c$$

12.5 Sommario

Per pulire una grammatica bisogna:

1. Eliminare le produzioni ϵ
2. Eliminare le produzioni unità
3. Eliminare simboli inutili

in quest'ordine

12.6 Forma normale di Chomsky, CNF

Ogni CNF non vuoto, senza ϵ, ha una grammatica G di simboli inutili nella seguente forma $A \to BC$ dove $\{A, B, C\} \subseteq V$ oppure $A \to a$ dove $a \in T$ e $A \in V$.
Per arrivare a questa forma bisogna:

- Pulire la grammatica
- Modificare le produzioni con 2 o più simboli in modo che siano tutte variabili
- Ridurre le produzioni con più di 2 simboli in catene da 2 simboli

Quindi per ogni terminale a più lungo di 1 creo una nuova variabile $A \to a$.
Mentre per ogni regola

$$A \to B_1, B_2, ..., B_k$$

$k \geq 3$ creo nuove variabili $C_1, C_2, ..., C_{k-2}$

$$A \to B_1, C_1$$

$$C_1 \to B_2, C_2$$

$$...$$

$$C_{k-3} \to B_{k-2}, C_{k-2}$$

$$C_{k-2} \to B_{k-1}, B_k$$

Esempio Usiamo la grammatica
$E \to |E + F|T * F|(E)|b|Ia|Ib|I0|I1$

$T \to (E)|T * F|a|b|Ia|Ib|I0|I1$
$F \to (E)|a|b|Ia|Ib|I0|I1$
$I \to a|b|Ia|Ib|I0|I1$

applichiamo il passo 2
$A \to a, B \to b, Z \to 0, O \to 1$
$P \to +, M \to *, L \to (, R \to)$

otteniamo
$E \to EPF|TMF|LER|b|IA|IB|IZ|IO$
$T \to LER|TMF|a|b|IA|IB|IB|IO$
$F \to LER|a|b|IA|IB|IZ|IO$
$I \to a|b|IA|IB|IZ|IO$
$A \to a, B \to b, Z \to 0, O \to 1$
$P \to +, M \to *, L \to (, R \to)$

Applichiamo il passo 3
$E \to EPT$ diventa $E \to EC_1, C_1 \to PT$

$E \to TMF, T \to TMF$ diventa $E \to TC_2, T \to TC_2, C_2 \to MF$

$E \to LER, T \to LER, F \to LER$ diventa $E \to LC_3, T \to TC_3, F \to LC_3, C_3 \to ER$

La grammatica finale diventa
$E \to EC_1|TC_2|LC_3|b|IA|IB|IZ|IO$
$T \to LC_3|TC_2|a|b|IA|IB|IB|IO$
$F \to LC_3|a|b|IA|IB|IZ|IO$
$I \to a|b|IA|IB|IZ|IO$
$C_1 \to PT$, $C_2 \to MF$, $C_3 \to ER$
$A \to a, B \to b, Z \to 0, O \to 1$
$P \to +, M \to *, L \to (, R \to)$

12.7 Pumping lemma per CFL

Pumping lemma per i linguaggi regolari: per una stringa sufficiente lunga è possibile causare un ciclo e creare un infinità di stringhe che appartengono al linguaggio
Pumping lemma per CFL: per una stringa sufficiente lunga è possibile

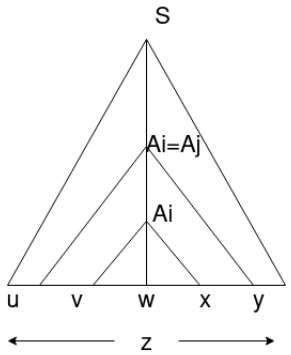
trovare due sottostringhe vicine che si possono iterare "in tandem". Posso iterare i volte per trovare nuove stringhe appartenenti al linguaggio.
Formalmente:
Sia L un CFL. Allora $\exists n \geq 1$ che soddisfa: ogni $z \in L : |z| \geq n$ è composto da 5 stringhe $z = uvwxz$ tali che:

1. $|vwx| \leq n$
2. $|vx| > 0$
3. $\forall i \geq 0, uv^iwx^iy \in L$

Dimostrazione:

- Si consideri una grammatica per $L\backslash\{\epsilon\}$ in CNF
- Assumiamo che la grammatica abbia m variabili, con $n = 2^m$
- Sia $z \in L$ una qualsiasi stringa tale che $|z| \geq 2^m$ allora ogni albero sintattico di z ha un cammino $\geq m + 1$
- Se tutti i cammini dell'albero hanno lunghezza $\leq m$ allora la stringa generata ha lunghezza $\leq 2^{m-1}$

Prendendo un cammino sufficientemente lungo, a partire da A fino a A_0 fino a A_k a un certo punto troverò due sotto alberi A_i, A_j con $i \neq j$ che sono uguali. Questo succede perché $k \geq m$, quindi avendo solo m variabili distinte è normale che si ripetano. Ora posso radicare l'albero A_j sotto A_i e la stringa finale non cambia, perché rispetta $z = uv^iwx^iy$ Notiamo che:



- l'albero in A_i ha altezza $\leq m+1$ quindi la stringa corrispondente ha lunghezza $\leq 2^m = n$ e quindi ($|vwx| \leq n$)
- v e x non possono essere vuote, perché la grammatica genera variabili non terminali perciò $|vx| > 0$
- posso ripetere l'albero A_i n volte e creo sempre una stringa valida ($uv^iwx^iy \in L$)

12.8 Applicazione Pumping lemma

Possiamo usarlo per dimostrare che un linguaggio non è libero.
Esempio: Si consideri $L = 0^m10^m10^m : m \geq 1$, dimostra che L non è CFL.
Dimostrazione: Assumiamo, per assurdo, che L sia CFL. Sia n la costante del Pumping lemma. Si consideri la strina $z = 0^n10^n10^n$. Si ha che $z \in L$ e $|z| \geq n$. Allora per il Pumping lemma, $z = uvwxy$ con $|uwx| \leq n$, $|vx| > 0$ e $uv^iwx^iy \in L$ per ogni $i \geq 0$. Consideriamo i seguenti casi:

- vx contiene almeno un 1
- vx contiene almeno un 0

In entrambi i casi $uwy \notin L$

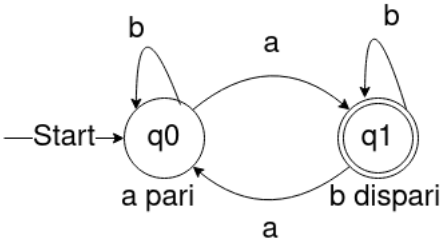
13 Esercitazioni

13.1 Esercitazione 09/21/23

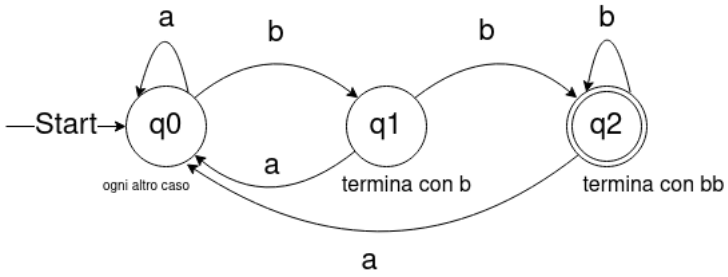
13.1.1 Costruzione DFA

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

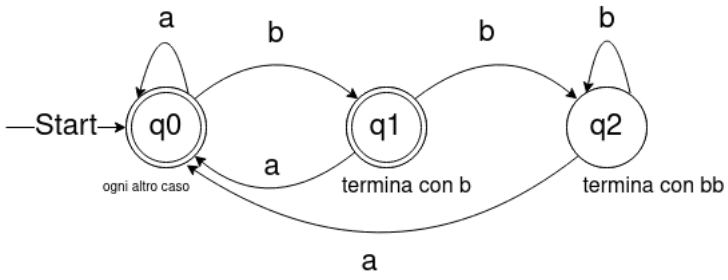
1. stringhe con un numero dispari di a
   SI ab,aaa,bba,aaba
   NO ϵ,aa



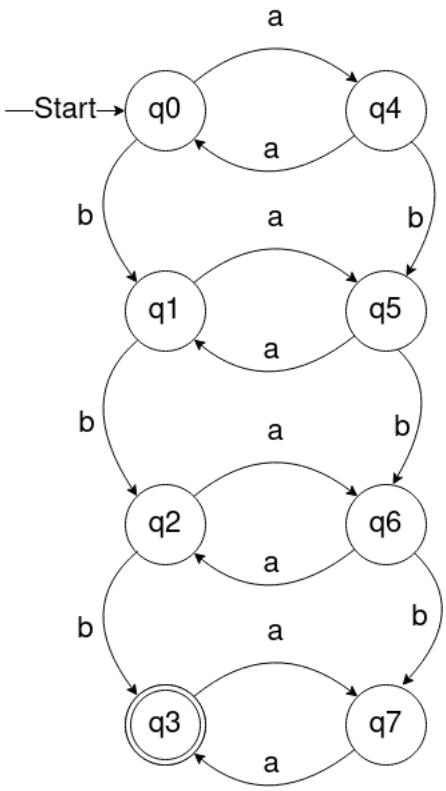
2. stringhe che terminano con bb
   SI bb,babb
   NO ϵ,ba,a,aba



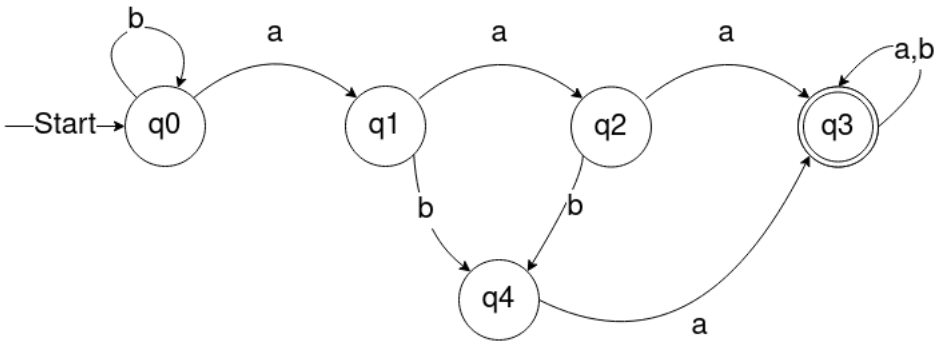
3. stringhe che non terminano con bb
   SI ϵ,ba,a,aba
   NO bb,babb



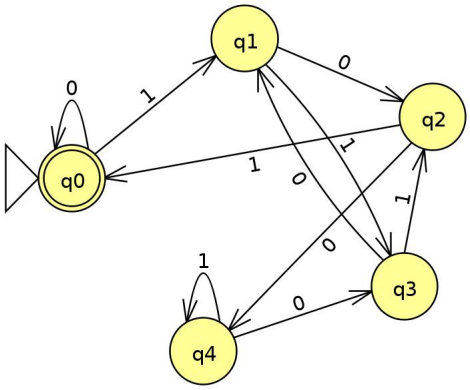
4. stringhe con un numero pari di a ed almeno 3 b
   SI bbb,bababb
   NO ϵ,bbaa,bababa



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

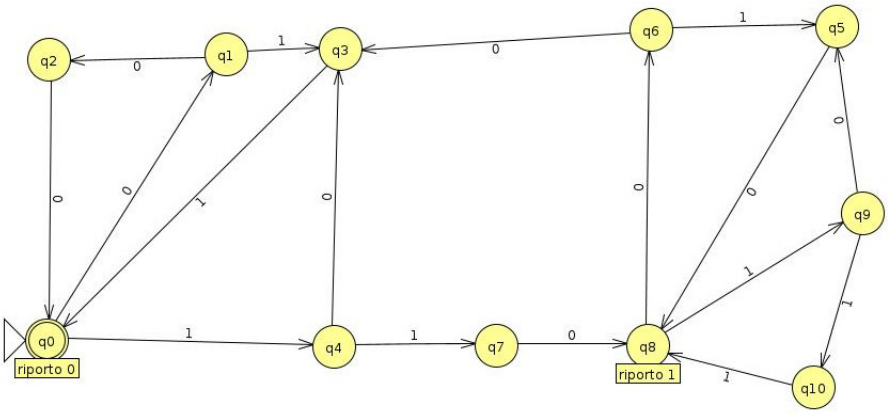


6. Realizzare un DFA che riconosca il seguente linguaggio su alfabeto {0,1}: stringhe che interpretate come numero binario risultano un multiplo di 5
   SI 101,1010,1111,0
   NO 111,1,10



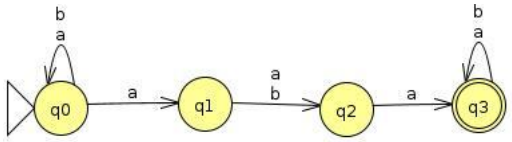
NB: Devi considerare tutti i numeri fino a 5!

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)

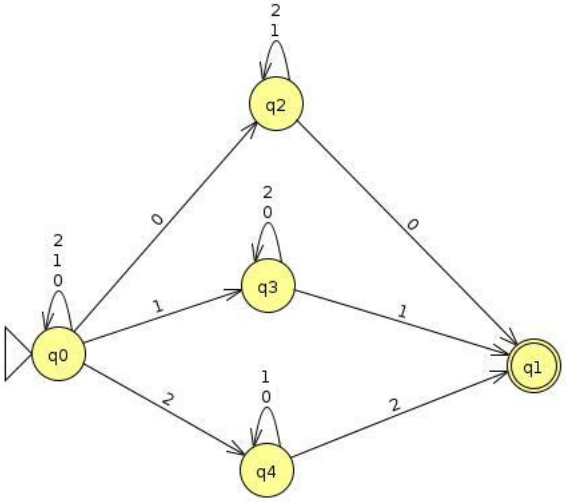


13.1.2 NFA

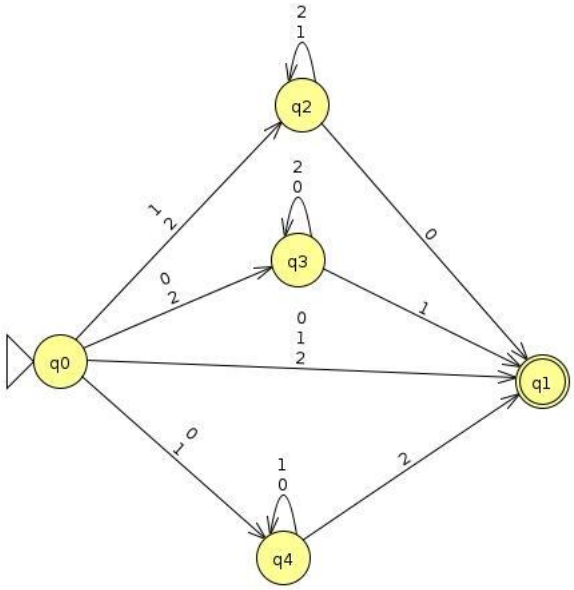
1. Dato l'alfabeto {a,b} realizzare un NFA che riconosce le stringhe che contengono aaa oppure aba
   SI babab,aaaa,aaaba
   NO ϵ,abba,a,b,ab



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.