Appunti del corso Linguaggi e Computabilità

Marco Natali

Il corso di Linguaggi e Computabilità riguarda l'informatica teorica e si occupa di definire la calcolabilità di un problema, di definire le grammatiche e i linguaggi formali con l'ausilio di anche di automi e macchine di Turing.

Incominciamo con la definizione dei componenti basilari attraverso cui svilupperemo poi i concetti del corso

Def. Si definisce come *alfabeto*, indicato con Σ, una sequenza di simboli, attraverso cui possiamo stabilire un alfabeto.
Si definisce invece come stringa, una sequenza finita di simboli appartenenti ad un alfabeto Σ ed esisterà sempre la stringa ϵ, indicante la stringa vuota.

Esempio.

$$\Sigma = \{0, 1\} \text{ e} \Sigma = \{a, b, c\}$$
$$w = 10110 \;\; z = abccabbcc$$

Def. È possibile fornire una definizione induttiva di stringa, partendo dalla stringa ϵ:

CASO BASE : ϵ è una stringa vuota

CASO PASSO : se w è una stringa, allora anche $a \circ w$ è una stringa

Dopo aver fornito le definizioni per le stringhe, definiamo le seguenti operazioni definite su le stringhe:

- insieme di stringhe: definiamo come Σ^k l'insieme di stringhe su Σ con k caratteri come segue:

  $$\Sigma^0 = \{\epsilon\}$$
  $$\Sigma^1 \neq \Sigma \text{ma sono l'insieme delle stringhe di un carattere}$$
  $$\Sigma^2 = \text{insieme di stringhe di due caratteri}$$
  $$\ldots\ldots$$
  $$\Sigma^k = \text{insieme di stringhe di k caratteri}$$

  Le due più importanti insiemi di stringhe, usate per rappresentare l'insieme di stringhe di qualsiasi lunghezza, sono:

  $$\Sigma^* = \cup_{i=0} \Sigma^i$$
  $$\Sigma^+ = \Sigma^* - \{\epsilon\}$$

- $|w| : \Sigma^* \to \mathbb{N}$: rappresenta la lunghezza di una stringa, ossiail numero di caratteri presenti in una stringa, e la definizione di lunghezza avviene induttivamente come segue:

  BASE : la lunghezza di $|\epsilon| = 0$

  PASSO : se $|w| = n$ con $n \in \mathbb{N}$ e sia $a \in \Sigma$ allora $|a \circ w| = 1 + |w| = n + 1$.

  Esempio. $w = abcdec \quad |w| = 6$

- $\circ : \sum^* \times \sum^* \to \sum^*$: rappresenta la concatenazione, ossial'aggiunta dei caratteri della seconda stringa al termine della prima stringa ma vediamo ora una definizione più formale:

  Def. Date due stringhe $x = a_1 a_2 \ldots a_n$ e $y = b_1 b_2 \ldots b_n$ si definisce $x \circ y = a_1 a_2 \ldots a_n b_1 b_2 \ldots b_n$ con $|x \circ y| = |x| + |y| = n + m$.

  La concatenazione possiede le seguenti proprietà:

  – è associativa: $\forall x, y, z \in \sum^* x \circ (y \circ z) = (x \circ y) \circ z$.

- non è commutativa infatti presi due stringhe x, y diverse risulta $x \circ y \neq y \circ x$.
- possiede l'elemento neutro ϵ per cui $x \circ \epsilon = x = \epsilon \circ x$.

Esempio.

$$w = 1011100 \quad z = 1011110$$
$$w \circ z = 10111001011110$$
$$z \circ w = 10111101011100$$

Attraverso le seguenti operazioni si può stabilire che $(\Sigma^*, \circ, \epsilon)$ è un monoide libero su Σ.

Dopo aver considerato e definito l'alfabeto e le stringhe, bisogna definire i linguaggi, concetto su cui si basa tutto il nostro corso e l'informatica teorica:

Def. Definiamo come *Linguaggio* un'insieme di stringhe scelte su Σ^* scelte per far parte del linguaggio ossia $L \subseteq \Sigma^*$. Le componenti di un linguaggio sono:

ALFABETO : insieme di simboli su cui definiamo poi il lessico e la sintassi

LESSICO : definisce il vocabolario del linguaggio e viene definito tramite una grammatica di tipo 3

SINTASSI : definisce come le varie frasi del linguaggio devono essere disposte nel linguaggio e ciò viene definito da una grammatica di tipo 2.

SEMANTICA : il significato attribuito alle frasi del linguaggio però nei linguaggi formali deriva dalla sintassi anche se in questo corso la semantica non verrà affrontata.

I linguaggi possono essere riconosciuti attraverso degli automi oppure generati medianti delle grammatiche.
Un'altra cosa importante è che si hanno sottoinsiemi particolari di linguaggi, come l'insieme vuoto, che resta comunqe un linguaggio, il **linguaggio vuoto** e $\emptyset \in \Sigma^k$, $|\emptyset| = 0$ che è diverso dal linguaggio che contiene la stringa vuota $|\epsilon| = 1$, inoltre $\Sigma^* \subseteq \Sigma^*$ che ha lunghezza infinita.
Vediamo qualche esempio di linguaggio:

- il linguaggio di tutte le stringhe che consistono in n 0 seguiti da n 1:

$$\{\epsilon, 01, 0011, 000111, ...\}$$

- l'insieme delle stringhe con un uguale numero di 0 e di 1:

$$\{\epsilon, 01, 10.0011, 0101.1001, ..\}$$

- l'insieme dei numeri binari il cui valore è un numero primo:

$$\{\epsilon, 10, 11, 101, 111, 1011, ...\}$$

- Σ^* è un linguaggio per ogni alfabeto Σ
- $\emptyset$, il linguaggio vuoto, e $\{\epsilon\}$ sono un linguaggio rispetto a qualunque alfabeto

La prima modalità per definire un linguaggio è attraverso la definizione di una grammatica, per stabilire tutte e sole le stringhe del linguaggio. In questo paragrafo forniremo soltanto una definizione informale e definiamo le diverse tipologie di gerarchie, senza effettuarne una trattazione formale, cosa che avverrà nel prossimo capitolo.

Def. Si definisce come *Grammatica* un'insieme di regole che delineano le stringhe ammissibili del linguaggio e possono essere di due tipologie:

- grammatica generativa : insieme di regole che permettono di generare tutte le stringhe di un lunguaggio partendo dalle stringhe base
- grammatica analitica : analizza le stringhe passate in input e stabilisce l'appartenenza o meno al linguaggio

Nel 1956 il linguista Choumsky introdusse e definì la gerarchia delle grammatiche e linguaggi, che ha avuto una notevole importanza nell'informatica teorica anche se il suo intento era quello di catalogare le varie tipologie di linguaggi naturali:

GRAMMATICHE DI TIPO 0 : non si hanno restrizioni sulle regole di produzione, $\alpha \rightarrow \beta$ e sono linguaggi ricorsivamente numerabili, rappresentati dalle *macchine di Turing*, deterministiche o non deterministiche (la macchina di Turing è un automa).

GRAMMATICHE DI TIPO 1 : il lato destro della produzione ha lunghezza almeno uguale a quello sinistro e si chiamano anche grammatiche dipendenti dal contesto e come automa hanno *la macchina di Turing che lavora in spazio lineare*:

$$\alpha_1 A \alpha_2 \rightarrow \alpha_1 B \alpha_2$$

con α_1 e α_2 detti *contesto* e α_1, α_2, $\beta \in (V \cup T)^*$

GRAMMATICHE DI TIPO 2 : sono quelle libere dal contesto, context free ed usano come riconoscitore gli automi a pila. Come regola ha $A \rightarrow \beta$ con $A \in V$ e $\beta \in V \cup T)^*$.

GRAMMATICHE DI TIPO 3 : sono le grammatiche *regolari*, i cui linguaggi vengono riconosciuti dagli automi a stati finiti.
Come regole ha $A \rightarrow \alpha B$ (o $A \rightarrow B\alpha$) e $A \rightarrow \alpha$ con $A, B \in V$ e $\alpha \in T$.

Durante questo corso effettueremo una trattazione delle grammatiche di tipo 3 e 2, anche se analizzeremo solo un esempio di grammatica di tipo 1; iniziamo nel prossimo capitolo a considerare le grammatiche di tipo 2, chiamate grammatiche context-free.

1 GRAMMATICHE DI TIPO 2

Iniziamo ora a considerare le grammatiche di tipo 2 partendo da un esempio e poi fornendo una definizione formale.

Esempio. Dato il linguaggio delle stringhe palindrome $L_{pal} = \{w \in \Sigma^* : w = w^R\}$ dove w^R rappresenta la stringa reversa per cui ad esempio $'OTTO' \in L_{pal}$ mentre $'PAPA'$ non appartiene al linguaggio.

Definiamo in maniera più formale e meno ambigua i componenti di L_{pal}:

CASO BASE : $\epsilon, 0, 1 \in L_{pal}$

CASO INDUTTIVO : se $w \in L_{pal}$ allora $0w0$ e $1w1$ appartengono a L_{pal} e nient'altro appartiene al linguaggio.

Le regole di derivazione per L_{pal} sono le seguenti, e derivano dalla definizione dei componenti:

- $P \to \epsilon$
- $P \to 0$
- $P \to 1$
- $P \to 0P0$
- $P \to 1P1$

Una versione più concisa delle regole di derivazione è la seguente:

$$P_g\{P \to \epsilon|0|1|0P0|1P1\}$$

Dopo aver dato le regole di derivazione dobbiamo capire se le seguente regole definiscono tutte e sole le stringhe del linguaggio per cui ci chiediamo per esempio se $010 \in L_{pal}$ e $10001 \in L_{pal}$?

$$P \Rightarrow 0P0 \Rightarrow 010 \quad \text{la stringa appartiene correttamente al linguaggio} \tag{1}$$

$$P \Rightarrow 1P1 \Rightarrow 10P01 \Rightarrow 10001 \quad \text{la stringa appartiene correttamente al linguaggio} \tag{2}$$

(3)

Attraverso l'applicazione di una serie di regole di derivazione otteniamo una stringa $w \in \Sigma^*$ se e solo $w \in L$ della grammatica definita.

Def. Si definisce una grammatica context free come $G = (V, T, P_g, S)$, i cui componenti sono:

- V rappresenta l'insieme delle variabili usate per rappresentare un linguaggio
- T rappresenta l'insieme dei simboli terminali, ossia l'insieme dei simboli attraverso cui sono definite le stringhe del linguaggio per questo di solito coincide con l'alfabeto del linguaggio.
- S rappresenta la variabile di inizio della grammatica, ossia la variabile attraverso cui si definisce la grammatica per cui le altre variabili sono classi ausiliari di stringhe che aiutano a definire le stringhe del linguaggio.

- P_g indica l'insieme di regole, che rappresentano la definizione ricorsiva del linguaggio, della seguente forma:

$$P_g = \{X \to \beta | \beta \in (V \cup T)^* \text{ e } X \in V\}$$

  la variabile X rappresenta la testa della produzione mentre β è il corpo

Non si possono applicare delle regole in parallelo, ma soltanto una alla volta.

Nell'esempio del linguaggio palidromo la grammatica che lo genera è

$$G = (\{P\}, \{0,1\}, \{P \to \epsilon, P \to 0, P \to 1, P \to 0P0, P \to 1P1\}, S\}$$

Dopo aver definito in maniera formale di grammatica introduciamo il concetto di derivazione, per stabilire se una stringa appartiene o meno al linguaggio.

Def. Sia $G = (V, T, P_g, S)$ una grammatica context-free, sia $\alpha A \beta$ una stringa di terminali e variabili con $A \in V$ e sia infine $A \to \gamma$ una regola di derivazione allora $\alpha A \beta \Rightarrow_g \alpha \gamma \beta$.
Si indica $\Rightarrow_g^*$, il simbolo di applicazione di zero,uno o più step di derivazione, definito nel seguente modo:

CASO BASE : per ogni stringa α di terminali e variabili, si ha $\alpha \Rightarrow_g^* \alpha$

CASO PASSO : se $\alpha \Rightarrow_g^* \beta$ e $\beta \Rightarrow_g \gamma$ allora $\alpha \Rightarrow_g^* \gamma$

Si può anche dire che $\alpha \to_G * \beta$ se e solo se esiste una sequenza di stringhe $\gamma_1, ..., \gamma_n$ con $n \geq 1$ tale che $\alpha = \gamma_1$, $\beta = \gamma_n$ e $\forall i$, $1 < i < n-1$ si ha che $\gamma_1 \to \gamma_{i+1}$ per cui la derivazione in o o più passi è la chiusura transitiva della derivazione.

Le stringhe che otteniamo sono delle forme sentenziali, ossia stringhe appartenenti a $(V \cup T)^*$ e un particolare sottoinsieme, in cui le stringhe sono composte soltanto da terminali, definisce le stringhe del linguaggio.

Sempre considerando l'esempio del linguaggio delle stringhe palindrome la derivazione di 10011001 è la seguente:

$$P \Rightarrow 1P1 \Rightarrow 10P01 \Rightarrow 100P001 \Rightarrow 1001P1001 \Rightarrow 10011001$$

Al fine di ridurre il numero di scelte nella derivazione di una stringa introduciamo ora:

LEFT DERIVATION : sostituiamo la variabile più a sinistra nell'applicazione di una regola di derivazione e ciò viene rappresentato con il simbolo $\Rightarrow_{lm}$.

RIGHT DERIVATION : sostituiamo la variabile più a destra nell'applicazione di una regola di derivazione ed essoviene rappresentato con il simbolo $\Rightarrow_{rm}$.

Def. Data una grammatica context-free $G = (V, T, P_g, S)$ si definisce un linguaggio context-free L come:

$$L_g = \{w \in T^* | S \Rightarrow_g^* w\}$$

Nei linguaggi context-free si può soltanto effettuare la concatenazione e l'annidamento dei sottolinguaggi come vediamo nei seguenti esempi:

Esempio. Mostriamo ora un esempio di linguaggio definito come la concatenazione di due linguaggi

$$L_g = \{w \in \{0,1\}^* | w = 0^m 1^{m+1} 0 1^n 0 0 1^n \ n \geq 0\}$$

Si può vedere che ci 3 blocchi che concatenati generano il linguaggio L_g per cui le regole di derivazione sono

$$P_g = \{S \to X0Y, X \to 1|0X1, Y \to 1001|1Y1\}$$

Tabella 1: Inferenza ricorsiva di $a * (a + b00)$

passo	stringa ricorsiva	var	prod	passo stringa impiegata
1	a	I	5	\
2	b	I	6	\
3	bo	I	9	2
4	boo	I	9	3
5	a	E	1	1
6	boo	E	1	4
7	a+boo	E	2	5,6
8	(a+boo)	E	4	7
9	a*(a+boo)	E	3	5, 8

Esempio. Mostriamo ora un esempio di linguaggio definito come l'annidamento di linguaggi:

$$L = \{w = \{a, b, c, d\}^* | w = a^n b^m c^m d^n \; m > 0, n \geq 0\}$$

Le regole di produzione del seguente linguaggio sono $P_g = \{S \to aSd|Y, Y \to bYc|bc\}$ e come si nota il linguaggio viene definito come una sequenza di a e d intrammezzate da un blocco Y, formato da b e c, e ciò è l'annidamento tra diversi blocchi di una stringa che formano le stringhe del linguaggio.

Un'altra modalità per stabilire l'appartenza di una stringa di un linguaggio è l'*inferenza ricorsiva*, in cui, a differenza delle altre modalità, applica le regole dal corpo alla testa, ossia concateniamo ogni terminale che appare nel corpo e inferiamo che la stringa trovata è nel linguaggio delle variabili, presenti in testa alle regole.

Viene poco utilizzato in quanto è più naturale e chiaro pensare secondo la derivazione per cui ne vediamo soltanto un esempio e lo usiamo in un importante teorema sull'equivalenza delle modalità di derivazione, che verrà presentato prossimamente.

Esempio. Sia $G = (V, T, O, E)$, con $V = \{E, I\}$ e $T = \{a, b, 0, 1, (,), +, *\}$ quindi ho le seguenti regole, è di tipo 3:

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

Voglio ottenere $a * (a + b00)$ e sostituisco sempre a destra (right most derivation):

$$E \to E * E \to E * (E) \to E * (E + E) \to E * (E + I) \to E + (E + I0)$$

$$\to R + (I + b00) \to E * (a + b00) \to I * (a + b00) \to a * (a + b00)$$

usiamo ora *l'inferenza ricorsiva*:

1.1 ALBERI DI DERIVAZIONE

Introduciamo ora un'importante forma grafica per vedere le regole di derivazioni applicate per formare una stringa

Def. Data una grammatica context-free G, un albero di derivazione per G è un albero composto come:

- ogni nodo interno è etichettato con una variabile $X \in V$, con la radice etichettata con S.
- ogni foglia è etichettata con una variabile, un simbolo terminale o ϵ; se una foglia viene etichettata con ϵ allora dev essere l'unico figlio del padre.

- se un nodo è etichettato con A e i suoi figli sono etichettati come $x_1, x_2, \ldots, x_k$, allora $A \to x_1 x_2 \ldots x_k$ è una regola di produzione della grammatica.

Eseguendo la concatenazione delle stringhe foglia otteniamo una stringa w appartenente alla grammatica di cui abbiamo svolto l'albero sintattico

Esempio. Prendendo il linguaggio L definito come segue:$L = \{w \in \{a,b,c,d\}^* | w = a^n c b^m c d^{n+m} n \geq 0, m > 0\}$ stabiliamo una grammatica context-free e fare l'albero sintattico di $aacbbbcddddd \in L$: le regole di produzione di L sono $P_g = \{S \Rightarrow aSd|cY, Y \Rightarrow bcd|bYd\}$ per cui la grammatica è:

$$G = (\{S, Y\}, \{a, b, c, d\}, P_g, S)$$

L'albero sintattico di *aacbbbcddddd* è il seguente:

1.2 EQUIVALENZA TRA LE DERIVAZIONI

In questo paragrafo consideriamo un importante teorema sulle equivalenze tra le varie modalità di derivazione, definito come segue:

Thm: 1.1. Data una grammatica context-free $G = (V, T, P_g, S)$ abbiamo che le seguenti modalità di derivazione sono equivalenti:

1. la procedura di inferenza ricorsiva che determina che una stringa di terminali w appartiene ad un linguaggio di variabili A
2. $A \Rightarrow^* w$
3. $A \Rightarrow^*_{lm} w$
4. $A \Rightarrow^*_{rm} w$
5. esiste un albero sintattico con radice A e come prodotto di foglie la stringa w.

Esempio. Usiamo l'esempio delle stringhe palindrome:

$$P \to 0P0|\, 1P1|\epsilon$$

sia il seguente albero sintatico:

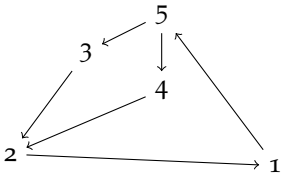
Esempio. Si ha:

$$E \to I|\, E + E|\, E * E|\, (E)$$

$$I \to a|\, b|\, Ia|\, Ib|\, I0|\, I1$$

un albero sintattico per $a * (a + b00)$ può essere:

queste 5 proposizioni si implicano l'uni l'altra:



Le prime due sono banali dato che una derivazione sinistra/destra sono anche delle derivazioni quindi risulta verificato. Vediamo le altre dimostrazioni di implicazioni:

da 1 a 5. si procede per induzione:

- **caso base:** ho un livello solo (una sola riga), $\exists A \to w$:

$$\begin{matrix} A \\ \triangle \\ w \end{matrix}$$

- **caso passo:** suppongo vero per un numero di righe $\leq n$, lo dimsotro per $n+1$ righe:

$$A \to X_1, X_2, ..., X_k$$

$$w = w_1, w_2, ..., w_k$$

ovvero, in meno di $n+1$ livelli:

□

da 5 a 3. procedo per induzione:

- **caso base (n=1):** $\exists A \to w$ quindi $A \to_{lm} w$, come prima si ha un solo livello:

$$\begin{matrix} A \\ \triangle \\ w \end{matrix}$$

- **caso passo:** suppongo che la proprierà valga per ogni albero di profondità minore uguale a n, dimostro che valga per gli alberi profondi $n+1$:

$$A \to X_1, X_2, ..., X_k$$

$$w = w_1, w_2, ..., w_k$$

ovvero, in meno di $n+1$ livelli:

$$A \to_{lm} X_1, X_2, ..., X_k$$

$x_1 \to^*_{lm} w_1$ per ipotesi induttiva si ha un albero al più di n livelli

quindi:

$$A \to_{lm} X_1, ..., X_k \to^*_{lm} w_1, X_2, ..., X_k \to^*_{lm} ... \to^*_{lm} w_1, ..., w_k = w$$

Esempio.

$$E \to I \to Ib \to ab$$

$$\alpha E \beta \to \alpha I \beta \to \alpha I b \beta \to \alpha a b \beta, \quad \alpha, \beta \in (V \cup T)^*$$

□

Esempio. Mostro l'esistenza di una derivazione sinistra dell'albero sintattico di $a * (a + b00)$:

$$E \to^*_{lm} E * E \to^*_{lm} I * E \to^*_{lm} a * E \to^*_{lm} a * (E) \to^*_{lm} a * (E + E) \to^*_{lm}$$

$$a * (I + E) \to^*_{lm} a * (a + E) \to^*_{lm} a * (a + I) \to^*_{lm} a + (a + I0) \to^*_{lm} a * (a + I00) \to^*_{lm} a * (a + b00)$$

1.3 GRAMMATICHE E LINGUAGGI AMBIGUI

Fino ad ora abbiamo considerato delle grammatiche uniche, ossia in grado di generare in maniera univoca un linguaggio ma non è sempre così, infatti in questo paragrafo consideramo il problema dell'ambiguità nelle grammatiche e nelle grammatiche.

Diamo ora una definizione formale di grammatica ambigua:

Def. Si dice che una grammatica è ambigua se e solo se esiste una stringa $w \in L$ tale per cui w ammette due derivazioni lm e/o rm diverse oppure ammette due alberi sintattici definiti sulla stessa grammatica G

L'obiettivo è quello di avere grammatiche non ambigue perchè esse permettono di definire in maniera univoca, per cui automamilizzabile con una procedura, l'insieme delle stringhe del linguaggio e fortunatamente nella maggior parte dei casi, quando il linguaggio non è ambiguo, data una grammatica ambigua è possibile definire una grammatica non ambigua G' tale che $L(G) = L(G')$.

Esempio. vediamo un esempio:

La grammatica data delle espressioni algebriche ha due cause di ambiguità:

1. la precedenza degli operatori non viene rispettata in quanto andrebbe raggruppato prima il silmbolo di $*$ rispetto a $+$
2. una sequenza di uguali operatori può essere reggruppata sia da sinistra che da destra e si stabilisce per convenzione che si raggruppa da sinistra a destra.

Considerate queste cause di ambiguità, per eliminarla si introducono altre variabili definite come:

1. un fattore è un espressione che non può essere spezzata da nessun operatore, sia il $*$ che il $+$, per cui gli unici fattori nel nostro linguaggio delle espressioni sono gli identificatori e ogni espressioni dentro le parentesi.
2. un termine è un espressione che non può spezzata dall'operatore $+$.
3. un espressione può riferirsi a qualsiasi possibile espressione, incluse quelle che possono essere spezzate da un adiacente $*$ o $+$

La grammatica non ambigua del linguaggio delle espressioni è la seguente:

$$P_{exp} = \{E \to T|E + T, T \to F|T * F, F \to I|(E), I \to a|b|aI|bI|0I|1I\}$$

Possono esserci più derivazioni di una stringa ma l'importante è che non ci siano alberi sintattici diversi e capire se una CFG è ambigua è un problema indecidibile, per cui molto complesso ed oneroso.

Esempio. vediamo un esempio:

$$S \to \epsilon|\, SS|\, iS|\, iSeS$$

con S=statement, i=if e e=else. Considero due derivazioni:

1. $S \to iSeS \to iiSeS \to iie$: Fare albero sintattico!!!!

Si ha quindi una grammatica ambigua

Per risolvere codesto problema nei linguaggi di programmazione i compilatori assumono la consecutine di associare l'else all'ultimo if

Def. Un linguaggio $L \subseteq \sum^*$ è detto *ambiguo* se per ogni grammatica G, tale per cui $L = L_G$, risulta che G è ambigua.

Esempio. Sia $L = \{a^n b^n c^m d^m | \, n, m \geq 1\} \cup \{a^n bmnc^m d^n | \, n, m \geq 1\}$
si ha quindi un CFL formato dall'unione di due CFL. L è inerentemente ambiguo e generato dalla seguente grammatica:

- $S \to AB | \, C$
- $A \to aAb | \, ab$
- $B \to cBd | \, cd$
- $C \to aCd | \, aDd$
- $D \to bDc | \, bc$

si possono avere due derivazioni:

1. $S \to_{lm} AB \to_{lm} aAbB \to_{lm} aabbB \to_{lm} aabbcBd \to_{lm} aabbccdd$
2. $S \to_{lm} C \to_{lm} aCd \to_{lm} aaBdd \to_{lm} aabBcdd \to_{lm} aabbccdd$

a generare problemi sono le stringhe con n=m perché possono essere prodotte in due modi diversi da entrambi i sottolinguaggi. Dato che l'intersezione tra i due sottolinguaggi non è vuota si ha che L è ambiguo

2 LINGUAGGI E GRAMMATICHE DIPENDENTI DAL CONTESTO

vediamo un esempio di grammatica dipendente dal contesto:

$L = \{a^n b^n c^n | \, n \geq 1\}$la cui grammatiche che lo genera è la seguente:

$$G = \{V, T, P, S\} = \{(S, B, C, X)\} = \{(a, b, c), P, S\}$$

.
Ecco le regole di produzione, e le grammatiche di tipo 1 posso scambiare variabili a differenza delle context free:

1. $S \to aSBC$
2. $S \to aBC$
3. $CB \to XB$
4. $XB \to XC$
5. $XC \to BC$
6. $aB \to ab$
7. $bB \to bb$
8. $bC \to bc$
9. $cC \to cc$

Vediamo un esempio di derivazione: per $n = 1$ ho abc ovvero $S \to aBC \to abC \to abc$.
con $n = 2$ ho $aabbcc$:

$S \to aSBC \to aaBCBC \to aaBXBC \to aaBXCC \to aaBBCC \to aabBCC \to aabbCC \to aabbcC \to aabbcc$

Vedere dimostrazione pag 14 di Lorenzo Soligo Vediamo un esempio di grammatica dipendente dal contesto:

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

Si ha:

$$G = (\{S, X, C, D, Z\}, \{a, b, c, d\}, P, S)$$

con le seguenti regole di produzione:

- $S \to aSc | \, aXc$
- $X \to bXD | \, bD$
- $DC \to CD$
- $DC \to DZ$
- $DZ \to CZ$
- $XZ \to CD$
- $bC \to bc$
- $cC \to cc$

- $cD \to cd$
- $dD \to dd$

Provo a derivare $aabbbccddd$ quindi con $n = 2$, $m = 3$:

$$S \to aSC \to aaXCC \to aabXDCC \to aabbXDDCC \to$$

$$aabbbDDDCC \to aabbbCCDDD \to aabbbccddd$$

3 LINGUAGGI REGOLARI

Per definire le stringhe appartenenti ai linguaggi regolari, di tipo 3, vi può utilizzare le *grammatiche regolari*, in cui vengono definite delle regole per stabilire se e quando una stringa appartiene al linguaggio, oppure le *espressioni regolari*.

Incominciamo a considerare le grammatiche regolari, sottoinsieme delle grammatiche di tipo 2 secondo la gerarchia di Choumsky, utilizzate per generare i linguaggi regolari.

Si ha la solita grammatica $G = (V, T, P, S)$ con però vincoli su P:

- ϵ si può ottenere solo con $S \to \epsilon$
- le produzioni sono tutte lineari a destra ($A \to aA$ o $A \to a$) o a sinistra ($A \to Ba$ o $A \to a$)

Esempio. $I \to a|\, b|\, Ia|\, Ib|\, I0|\, I1$ è una grammatica con le produzioni lineari a sinistra.
Potremmo pensarlo a destra $I \to a|\, b|\, aI|\, bI|\, 0I|\, 1I$.
Vediamo esempi di produzione con queste grammatiche:

- con $I \to a|\, b|\, Ia|\, Ib|\, I0|\, I1$ possiamo derivare $ab01b0$:

$$I \to I0 \to Ib0 \to I1b0 \to I01b0 \to Ib01b0 \to ab01b0$$

- con $I \to a|\, b|\, aI|\, bI|\, 0I|\, 1I$ invece non riusciamo a generare nulla:

$$I \to 0I \to 0a$$

Definisco quindi un'altra grammatica (con una nuova categoria sintattica):

$$\begin{aligned} I &\to aJ|\, bJ \\ J &\to a|\, b|\, aJ|\, bJ|\, 0J|\, 1J \end{aligned}$$

che però non mi permette di terminare le stringhe con 0 e 1, la modifico ancora otterdendo:

$$\begin{aligned} I &\to aJ|\, bJ \\ J &\to a|\, b|\, aJ|\, bJ|\, 0J|\, 1J|\, 0|\, 1 \end{aligned}$$

e questo è il modo corretto per passare da lineare sinistra a lineare destra

Esempio. Sia $G = (\{S\}, \{0, 1\}, P, S)$ con $S \to \epsilon|\, 0S|\, 1S$ e si ha quindi:

$$L(G) = \{0, 1\}^*$$

Si hanno comunque solo produzioni lineari a destra mentre usando le produzioni lineari a sinistra ottengo:

$$S \to \epsilon|\, S0|\, S1$$

Esempio. Trovo una grammatica lineare destra e una sinistra per $L = \{a^n b^m |\, n, m \geq 0\}$:

- **lineare a destra:** si ha $G = (\{S, B\}, \{a, b\}, P, S)$ e quindi:

$$S \to \epsilon|\, aS|\, bB$$

$$B \to bB|\, b$$

ma non si possono generare stringhe di sole b, infatti:

$$S \Rightarrow aS \Rightarrow abB \Rightarrow abbB \Rightarrow abbb$$

ma aggiungere ε a B **non è lecito**. posso però produrre la stessa stringa da due derivazioni diverse:

$$S \to \varepsilon | \, aS | \, bB | \, b$$

$$B \to bB | \, b$$

che risulta quindi la nostra lineare a destra

- **lineare a sinistra:** si ha $G = (\{S, A\}, \{a, b\}, P, S)$ e quindi:

$$S \to \varepsilon | \, Sb | \, Ab | \, a$$

$$A \to Aa | \, a$$

3.1 ESPRESSIONI REGOLARI

Le espressioni regolari permettono di definire, utilizzando una notazione algebrica, un linguaggio regolare e vengono utilizzate per estrarre parole da un testo ed altre notevole applicazioni, che verranno analizzate nel corso dei paragrafi.

Per riuscire a definire in maniera formale le espressioni regolari dobbiamo definire le seguenti operazioni sui linguaggi regolari:

- Unione: dati due linguaggi $L, M \subseteq \Sigma*$ si definisce $L \cup M$ come:

$$L \cup M = \{w \in \Sigma * | w \in L \vee w \in M\}$$

  Esempio: $L = \{001, 10, 111\}$ e $M = \{\epsilon, 001\}$ risulta $L \cup M = \{\epsilon, 10, 001, 111\}$. Risulta verificata la seguente equivalenza $L \cup M \equiv M \cup L$.

- Concatenazione: dati due linguaggi $L, M \subseteq \Sigma*$ si ha $L \circ M = LM$, ossia il linguaggio formato da tutte le strighe ottenute concatenando le stringhe in L con le stringhe in M. Esempio: $L = \{001, 10\}$ e $M = \{\epsilon, 111\}$ risulta $L \circ M = \{001, 10, 001111, 10111\}$.

- Chiusura di Kleene: dato un linguaggio L si ha $L*$ definito induttivamente come:

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^1 &= L \\ L^2 &= L \circ L \\ &\dots \\ L^i &= L^{i-1} \circ L \\ L^* &= L^0 \cup L^1 \cup L^2 \cup \dots \\ L^+ &= L^* - \{epsilon\} \end{aligned}$$

  Esempio: dato $L = \{0, 1\}$ abbiamo:

$$\begin{aligned} L^0 &= \{epsilon\} \\ L^1 &= \{0, 1\} \\ L^2 &= \{00, 01, 10, 11\} \\ L^3 &= \{000, 001, 010, 011, 100, 101, 110, 111\} \end{aligned}$$

  Il linguaggio $L*$ è generalmente un linguaggio infinito in quanto è l'unione di un numero infinito di linguaggi finiti ma esistono due linguaggi la cui chiusura è finita, che analizziamo ora:

– il linguaggio $L = \{0\}$ la sua chiusura di kleene è finita dato che si ha:

$$\begin{aligned} L^0 &= \epsilon \\ L^1 &= L \\ L^2 = L \circ L &= L \\ L^3 = L^2 \circ L &= L \\ L^i = L^{i-1} \circ L &= L \\ L^* = L^0 \cup L^1 \cup L^2 \cup \cdots = L \cup epsilon &= L \end{aligned}$$

– il linguaggio $L = \emptyset$ la sua chiusera di kleene è finita in quanto:

$$\begin{aligned} L^0 &= \epsilon \\ L^1 &= \emptyset \\ L^2 = \emptyset \circ \emptyset &= \emptyset \\ L^i = \emptyset^i \circ \emptyset &= \emptyset \\ L^* = \emptyset \cup \{\epsilon\} &= \{epsilon\} \end{aligned}$$

Dopo aver definito le operazioni sui linguaggi regolari, definiamo ora le espressioni regolari, molto utilizzate per rappresentare in maniera algebrica la grammatica dei linguaggi regolari:

Def. Si definisce *espressione regolare* induttivamente come segue, considerando anche il linguaggio che generano:

CASO BASE : la base consiste in 3 parti:

1. ϵ e $\emptyset$ sono Regex e generano $L(\epsilon) = \{epsilon\}$, $L(\emptyset) = \emptyset$
2. se a è un simbolo allora a è una Regex e questa espressione genera $L(a) = \{a\}$
3. una variabile, rappresentanti linguaggi regolari, sono Regex, e generano $L(L) = L$

CASO INDUTTIVO : la parte induttiva delle espressioni regolari sono composte da 4 tipologie:

1. se E e F sono delle Regex allora $E + F$ è una Regex e rappresentano $L(E + F) = L(E) \cup L(F)$
2. se E e F sono delle Regex allora $E \circ F = EF$ è una Regex per cui rappresentano $L(EF) = L(E)L(F)$
3. se E è una Regex allora $E*$ è una RegEx, che denota la chiusura di $L(E)$ infatti $L(E^*) = (L(E))^*$
4. se E è una Regex allora (E) è una Regex in cui $L((E)) = L(E)$.

Esempio: Data l'espressione regolare $Regex = 01$ si ha allora $L(01) = L(0) \circ L(1) = 0 \circ 1 = 01$.

Al fine di ridurre la lunghezza delle espressioni regolari per migliorarne la leggibilità e la comprensione, si introducono delle proprietà algebriche iniziando prima dalla definizione di espressioni equivalenti:

Def. Due espressioni regolari sono equivalenti se denotano lo stesso linguaggio. Due espressioni regolari con variabili sono equivalenti se e solo se sono equivalenti per ogni assignamento alle variabili.

Per sapere quali operazioni in una Regex viene eseguita, si introduce la precedenza degli operatori, eseguiti da sinistra a destra:

- * e si applica alla sequenza più piccola a sinistra che sia anche un'espressione regolare
- $\circ$ applicato da sinistra a destra
- $+$ viene valutato da sinistra a destra
- la parentesi $()$ permette di isolare il contenuto dentro e stabilire l'ordine di applicazione

Gli operatori delle espressioni regolari possiedono le seguenti proprietà:

- Unione: l'unione si può vedere come l'addizione nell'aritmetica dato che possiede le stesse proprietà infatti:
  - **Commutatività**: dati due linguaggi L e M risulta $L + M = M + L$
  - **Associavità**: dati tre linguaggi L, M e N risulta $(L + M) + N = L + (M + N)$
  - **Identità**: l'identità per l'unione è l'insieme $\emptyset$ infatti risulta verificato $L + \emptyset = L = \emptyset + L$
  - **Idempotenza**: dato un linguaggio L risulta $L + L = L$.
- la Concatenazione presenta delle analogie con la moltiplicazione infatti possiede le seguenti proprietà:
  - **Associavità**: dati tre linguaggi L, M e N risulta $(LM)N = L(MN)$
  - **Identità**: l'identità per la concatenazione è l'ϵ attraverso cui risulta $\epsilon L = L = L\epsilon$
  - **Annichilatore**: l'annichilatore per la concatenazione è $\emptyset$ in quanto risulta $\emptyset L = \emptyset = L\emptyset$. L'annichilatore risulta molto utile per effettuare delle utili ed importanti semplificazioni.

  L'unione e la concatenazione possiedono le proprietà distributive della concatenazione rispetto all'unione:
  - $L(M + N) = LM + LN$ (legge distribuitiva sinistra della concatenazione rispetto all'unione)
  - $(M + N)L = ML + NL$ (legge distribuitiva destra della concatenazione rispetto all'unione)
- la chiusura possiede le seguenti proprietà:
  - $(L^*)^* = L^*$
  - $\emptyset^* = \epsilon$
  - $\epsilon^* = \epsilon$
  - $L^+ = LL^* = L^*L$
  - $L^* = L^+ + \epsilon$
  - $L? = \epsilon + L$

Esempio:Scrivere la regex del seguente linguaggio:

$$L = \{w \in \{a, b, c, d\}^* | w = a^n bcd^m \text{ con} n > 0, m \geq 0\}$$

Dato che si ha $n > 0$ ci sarà per forza una a per cui per generare a^n si ha aa^* mentre essendo $m \geq 0$ per definire d^m si usa $d*$.
Fatte codeste considerazioni si ottiene Regex = aa^*bcd*.

Scrivere la regex del seguente linguaggio $L = \{w \in 0, 1\}^* | w =$ stringhe in cui compare 10. La regex per generare il linguaggio è $(0 + 1)^*10(0 + 1)^*$ in quanto $(0 + 1)^*$ mi genera una qualsiasi stringa di 0 e 1 mentre 10 mi assicura di avere tutte e sole le strighe in cui compare 10.

4 AUTOMI A STATI FINITI

Un automa a stati finiti è un tipo di automa che permette di descrivere con precisione e in maniera formale il comportamento di molti sistemi.
Grazie alla sua semplicità e chiarezza questo modello è molto diffuso nell'ingegneria e nelle scienze, soprattutto nel campo dell'informatica e della ricerca operativa e può essere utilizzato sia per modellare un sistema esistente che per modellare un nuovo sistema formale in grado di risolvere alcuni nuovi problemi. Un automa a stati finiti permette di definire le stringhe accettabili in un linguaggio regolare e possiede un insieme di stati e un controllo che si muove da stato a stato in risposta a input esterni.
Si ha una distinzione:

AUTOMI DETERMINISTICI dove l'automa non può essere in più di uno stato per volta

AUTOMI NON DETERMINISTICI dove l'automa può trovarsi in più stati contemporaneamente

Iniziamo a parlare degli automi deterministici, facilmente implementabili da un calcolatore e su cui è possibile dimostrare la correttezza, per poi affrontare gli automi non deterministici, facili da disegnare e sviluppare dagli esseri umani.

4.1 AUTOMI DETERMINISTICI

Come già visto, un automa a stati finiti deterministico (*DFA*), è un automa che dopo aver letto una qualunque sequenza di input si trova in un singolo stato.
Il termine *deterministico* deriva dal fatto che per ogni input esiste un solo stato verso il quale l'automa passa dal suo stato corrente e dal punto di vista formale un DFA consiste nelle seguenti parti:

- un insieme finito di stati, spesso indicato con Q
- un insieme finito di simboli di input , spesso indicato con Σ
- una funzione di transizione δ, che prende come argomento uno stato e un simbolo di input e restituisce uno stato.
  Nella rappresentazione grafica informale di automi δ è rappresentata dagli archi tra gli stati e dalle etichette sugli archi. Se q è uno stato e a è un simbolo di input, $\delta(q, a)$ è lo stato p tale che esiste un arco etichettato con a da q a p
- uno stato iniziale q_0, corrispondente ad uno degli stati in Q
- un insieme $F \subseteq Q$ di stati finali o accettanti, in cui si accettano le stringhe arrivate in quello stato

Nel complesso un DFA è rappresentato in maniera concisa con l'enumerazione dei suoi elementi, quindi con la quintupla:

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

Vediamo come decidere se accettare o meno una stringa (sequenza di caratteri) in input mediante un DFA.

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

Ho una sequenza in input $a_1...a_n$. Parto dallo stato iniziale q_0, consultando la funzione di transizione δ, per esempio $\delta(q_0, a_1) = q_1$ e trovo lo stato in cui il DFA entra dopo aver letto a_1.
Poi passo a $\delta(q_1, a_2) = q_2$ e così via, $\delta(q_{i-1}, a_i) = q_i$ fino a ottenere q_n e se q_n è elemento di F allora $a_1...a_n$ viene accettato, altrimenti viene rifiutato.

Vediamo ora un esempio per capire come si rappresenta e come si sviluppa un automa:

Esempio. Specifico un DFA che accetta tutte le strighe binarie in cui compare la sequenza 01:

$$L = \{w \in 0, 1^* | w = x01y \ x, y \in 0, 1^*\} = \{01, 11010, 100011, ...\}$$

Ragioniamo sul fatto che A:

1. se ha "già visto" 01, accetterà qualsiasi input
2. pur non avendo ancora visto 01, l'input più recente è stato 0, cosicché se ora vede un 1 avrà visto 01
3. non ha ancora visto 01, ma l'input più recente è nullo (siamo all'inizio), in tal caso A non accetta finché non vede uno 0 e subito dopo un 1

L'automa DFA per rappresentare il linguaggio è il seguente:

$$A = \{\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}\}$$

con in totale le seguenti transizioni:

$$\delta(q_0, 1) = q_0$$
$$\delta(q_0, 0) = q_2$$
$$\delta(q_2, 0) = q_2$$
$$\delta(q_2, 1) = q_1$$
$$\delta(q_1, 0) = q_1$$
$$\delta(q_1, 1) = q_1$$

Definire l'automa con questa notazione è noioso e non molto semplice da sviluppare per questo si possono utilizzare le seguenti due maniera per rappresentare un automa:

- **tabella di transizione**: rappresentazione tabellare della funzione δ, dove lo stato iniziale viene indicato con $\rightarrow$ e gli stati accettati con $*$ Nel nostro esempio si ha la seguente tabella di transizione:
- **diagramma di transizione**: grafo indicante una rappresentazione grafica dell'automa per capire in maniera semplice ed immediata il comportamento dell'automa.
  Nel nostro esempio di automa il diagramma di transizione è il seguente:

Esempio. Trovo automa per:

$$L = \{w \in \{a, b\}^* | \text{ w che contiene un numero pari di b}\}$$

Figura 1: Automa per rappresentare le stringhe x01y

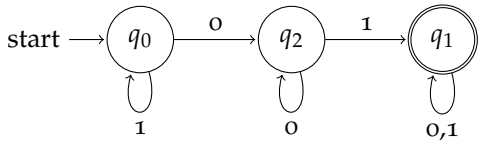
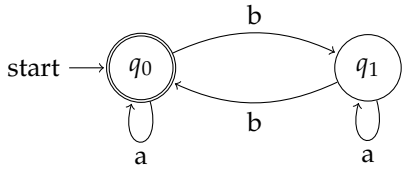


Figura 2: Automa per rappresentare un numero pari di b



Definiamo ora un estenzione della funzione di transizione $\hat{\delta}$, il quale ci descrive come opera un automa fornendo uno stato e una sequenza di input. La definizione può essere di due diversi tipi, ambedue di tipo induttivo:

- versione semplificata, comoda da definire ma difficile per dimostrare la correttezza in cui si considera una stringa $w = ax$ con $x \in \Sigma^*$ per cui la definizione è la seguente:

  CASO BASE $\hat{\delta}(q, \epsilon) = q$ con ovviamente $|w| = 0$

  CASO PASSO Supponiamo che $w = ax$, con $a \in \Sigma$ e $x \in \Sigma^*$ e $|w| \neq 0$ allora $\hat{\delta}(q, w) = \hat{\delta}(q, ax) = \hat{(}\delta)(\delta(q, a), x)$.

- versione fornita dal libro, comoda per la dimostrazione della correttezza del linguaggio in cui si considera una stringa $w = xa$, sempre con $x \in \Sigma^*$ e $a \in \Sigma$. La definizione è la seguente:

  CASO BASE $\hat{\delta}(q, \epsilon) = q$ in caso $|w| = 0$

  CASO PASSO Supponiamo di avere la stringa come $w = xa$, con $|w| \neq 0$, allora $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$.

Dopo aver definito la funzione di transizione estesa possiamo fornire una definizione formale del linguaggio accettato da un'automa infatti:

$$L(a) = \{w | \hat{\delta}(q_0, w) \in F\}$$

. In caso $L = L(a)$ per un DFA A, allora diciamo che L è un linguaggio regolare.

4.2 AUTOMI NON DETERMINISTICI

Un automa a stati finiti non deterministici (*NFA*) è un automa che può trovarsi in diversi stati contemporaneamente e come i DFA accettano linguaggi regolari.
Il motivo per cui sono studiati e definiti è perche spesso sono più semplici da trattare rispetto ai DFA, anche se è più difficile la dimostrazione della correttezza dei linguaggi.

Un NFA è definito come un DFA, ossia con la solita quintupla ma si ha un diverso tipo di transizione δ, che ha sempre come argomenti uno stato e un simbolo di input ma resituisce zero o più stati.

Esempio. Sia $L = \{x01 | x \in \{0, 1\}\}$ ovvero il linguaggio formato da tutte le stringhe binarie che terminano in 01 per cui avremo il seguente automa determinsitico per rappresentare il linguaggio: L'automa NFA corrispondente al linguaggio invece è:

Figura 3: DFA per il linguaggio $w = x01$

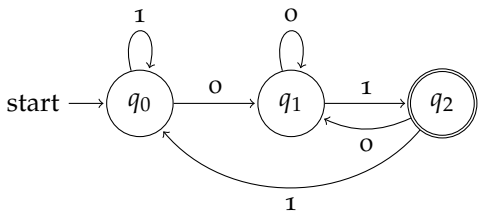
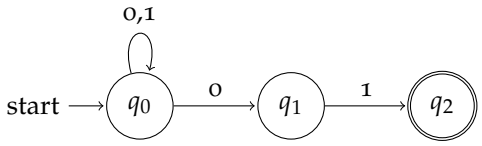


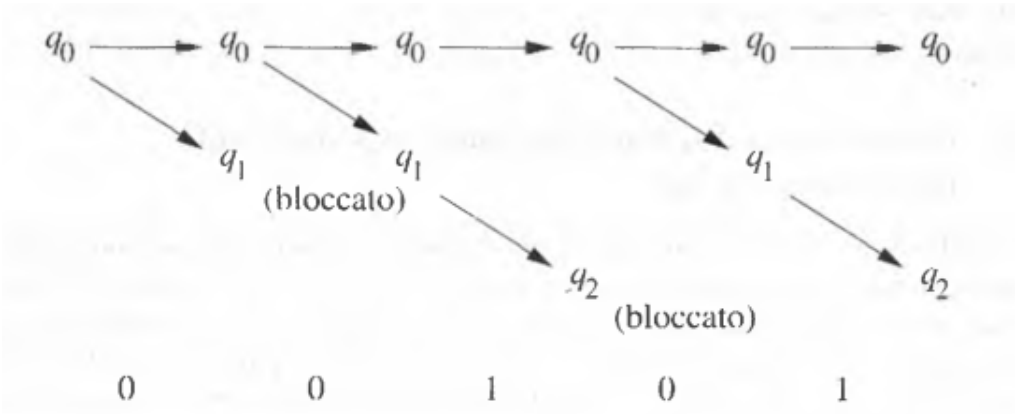
Figura 4: NFA per il linguaggio $w = x01$



quindi con:

$$\delta(q_0, 0) = \{q_0, q_1\}$$
$$\delta(q_0, 1) = \{q_0\}$$
$$\delta(q_1, 0) = \emptyset$$
$$\delta(q_1, 1) = \{q_2\}$$
$$\delta(q_2, 0) = \emptyset$$
$$\delta(q_2, 1) = \emptyset$$

in forma tabulare: vediamone la simulazione per la stringa 00101:



Il funzionamento dell'automa, come si nota nella figura, per la stringa 00101 è la seguente: si parte dallo stato inziale q_0, quando viene letto 0 si passa a q_0 e q_1, poi viene letto il secondo 0 quindi q_0 va nuovamente verso q_0 e q_1 mentre il primo q_1 muore non avendo transizioni su 0.
Arriva poi l'1 quindi q_0 va solo verso q_0 e q_1 verso q_2 e sarebbe accettante ma l'input non è finito.
Ora arriva 0 e q_2 si blocca mentre q_0 va sia in q_0 che in q_1 ed arriva infine un 1 che

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

manda q_0 in q_0 e q_1 in q_2 che è accettante e non avendo altri input si è dimostrata l'appartenenza della stringa al linguaggio.

Definisco quindi un NFA come una quintupla:

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

con, a differenza dei DFA:

$$\delta : Q \times F \to 2^Q$$

Possiamo ora definire induttivamente $\hat{\delta}$, delta cappuccio che prende in ingresso uno stato e l'intera stringa w e come già visto per i DFA ci sono ben due definizioni:

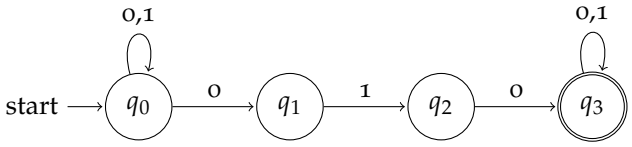
- versione semplificata fornita a lezione in cui si considera una stringa come $w = ax$, con $a \in Sigma$ e $x \in Sigma^*$ per cui la definizione di *delta cappuccio* è:
  - **caso base:** se $w = \epsilon$ si ha $\hat{\delta}(q, \epsilon) = \{q\}$
  - **caso passo:** sia $|w| > 0$, $w = ax$ ed avendo $\delta(q, a) = \{p_1, p_2, \ldots, p_k\}$ allora $\hat{\delta}(q, w) = \bigcup_{i=1}^{k} \hat{\delta}(p_i, x) = \{r_1, r_2, \ldots, r_m\}$
- versione fornita dal libro in cui si considera $w = xa$, con $x \in Sigma^*$ e $a \in \Sigma$ per cui la definizione di $\hat{\delta}$ è la seguente:
  - **caso base:** se $|w| = 0$ si ha $\hat{\delta}(q, \epsilon) = \{q\}$
  - **caso passo:** sia $|w| > 0$, $W = xa$, con $a \in \Sigma$ e $x \in \Sigma^*$.
    Posto $\hat{\delta}(q, x) = \{p_1, ..., p_k\}$ si ha:

$$\hat{\delta}(q, w) = \hat{\delta}(q, xa) = \bigcup_{i=1}^{k} \delta(p_i, a)$$

Il linguaggio accettato dall'automa NFA è il seguente:

$$L(a) = \{w \in \Sigma^* | \, \hat{\delta}(q_0, q) \cap F \neq \emptyset\}$$

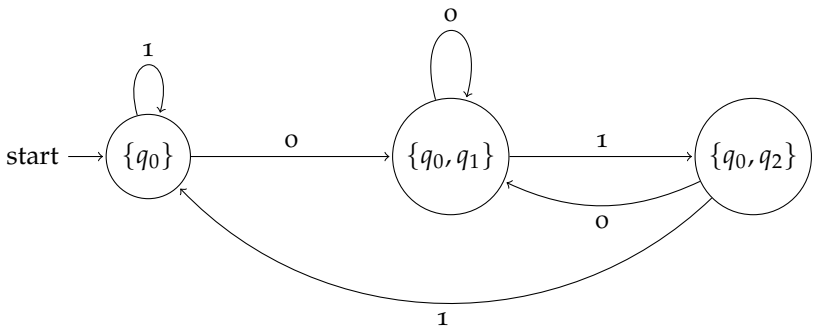
Esempio. Automa per $L = \{x010y | \, x, y \in \{0,1\}^*\}$ ovvero tutte le stringhe con dentro la sequenza 010:



Troviamo ora un algoritmo che trasformi un NFA in un DFA.
Dall'ultimo esempio ricavo:

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

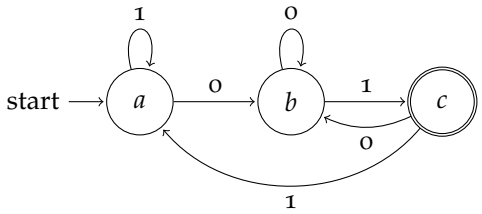
ovvero:



che è il DFA che si era anche prima ottenuto. Si hanno però dei sottoinsiemi mai raggiungibili. Si ha quindi:

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

e definendo $\{q_0\} = a$, $\{q_0, q_1\} = b$ e $\{q_0, q_2\} = c$ si avrà:



Definiamo questo algoritmo che avrà:

- come input un NFA $N = (Q_n, \Sigma, \delta_N, q_0, F_N)$
- come output un DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ tale che $L(D) = L(N)$

con:

- $Q_D = 2^{Q_N}$ (quindi se $Q_N = n$ si ha $|Q_D| = 2^n$)
- $F_D = \{S \subseteq Q_n | \, S \cap F_N \neq \emptyset\}$
- $\forall S \subseteq Q_N$ e $\forall a \in \Sigma$:

$$\delta_D(S, a) = \cup \delta_n(p, a)$$

per esempio:

$$\delta_D(\{q_0, q_1, q_2\}, 0) = \delta_N(q_0, 0) \cup \delta_N(q_1, 0) \cup \delta_N(q_2, 0)$$

4.3 AUTOMI $\epsilon - nfa$

Si definisce l'$\epsilon - NFA$ come l'automa a stati finiti non deterministici con ϵ transizioni, ossia la transizione che permette di saltare i nodi in cui si avanza nel grafo senza consumare caratteri.
Ovviamente questo automa non amplia la classe dei linguaggi accettati dall'automa, ma offre una certa comodità notazionale e come vedremo gli $\epsilon - NFA$ sono strettamente collegati alle espressioni regolari.

Formalmente denotiamo un $\epsilon - NFA$ con $A = (Q, \Sigma, \delta, q_0, F)$, con gli stessi argomenti uguali a quelli per il NFA con la differenza che la funzione δ è definita come $\delta : Q \times \Sigma \cup \{\epsilon\} \rightarrow P \subseteq Q$.

Per stabilire le stringhe e i linguaggi accettati da tali automi, dobbiamo introdurre la funzione $ECLOSE(q)$, per stabilire la chiusura di uno stato rispetto a ϵ, definita induttivamente come:

Figura 5: Automa $\epsilon - NFA$ per generare il linguaggio $a^n b^m c^k$

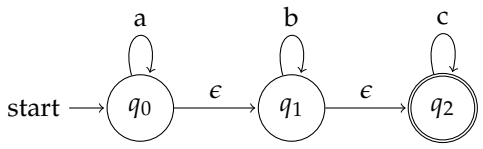


Tabella 2: Tabella di transizione dell'$epsilon - NFA$

	a	b	c
$* \rightarrow \{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\} = A$	$\{q_1, q_2\} = B$	$\{q_2\} = C$
$*\{q_1, q_2\}$	$\emptyset = D$	$\{q_1, q_2\}$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$	$\{q_2\}$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

- **caso base:** lo stato q appartiene a $ECLOSE(q)$
- **caso passo:** se lo stato $p \in ECLOSE(q)$ ed esiste una transizione ϵ da p a r, allora $r \in ECLOSE(q)$.
  Più precisamente, sia δ la funzione di transizione di un $\epsilon - NFA$: se $p \in ECLOSE(q)$, allora $ECLOSE(q)$ contiene anche tutti gli stati in $\delta(p, \epsilon)$.

Tramite questa definizione di $ECLOSE$ possiamo definire ora la funzione $\hat{\delta}$ in maniera induttiva:

- Fare definizione

Esempio. Si ha $ER = a^* b^* c^*$, che genera $L = \{a^n b^m c^k | \, n, m, k \geq 0\}$, l'automa per denotare il linguaggio è il seguente: Si ha poi le seguenti $ECLOSE$:

$$E\,CLOSE(q_0) = \{q_0, q_1, q_2\}$$

$$E\,CLOSE(q_1) = \{q_1, q_2\}$$

$$E\,CLOSE(q_2) = \{q_2\}$$

Mettendo in tabella l'automa $\epsilon - NFA$ si ha: Una rappresentazione formale di come si ottengono le seguenti transizioni nella tabella è la seguente:

$$\begin{aligned}\delta_D(\{q_0, q_1, q_2\}, a) &= E\,CLOSE(\delta_N(q_0, a) \cup \delta_N(q_1, a) \cup \delta_N(q_2, a)) \\ &= E\,CLOSE(\{q_0\} \cup \emptyset \cup \emptyset) = E\,CLOSE(\{q_0\}) \\ &= E\,CLOSE(q_0) = \{q_0, q_1, q_2\}\end{aligned}$$

$$\begin{aligned}\delta_D(\{q_0, q_1, q_2\}, B) &= E\,CLOSE(\delta_N(q_0, b) \cup \delta_N(q_1, b) \cup \delta_N(q_2, b)) \\ &= E\,CLOSE(\emptyset \cup \{q_1\} \cup \emptyset) = E\,CLOSE(\{q_1\}) \\ &= E\,CLOSE(q_1) = \{q_1, q_2\}\end{aligned}$$

$$\begin{aligned}\delta_D(\{q_0, q_1, q_2\}, c) &= E\,CLOSE(\delta_N(q_0, c) \cup \delta_N(q_1, c) \cup \delta_N(q_2, c)) \\ &= E\,CLOSE(\emptyset \cup \emptyset \cup \{q_2\}) = E\,CLOSE(\{q_1\}) \\ &= E\,CLOSE(q_2) = \{q_2\}\end{aligned}$$

Si ottiene quindi il seguente NFA: che diventa il seguente DFA:

Figura 6: NFA corrispondente all'$\epsilon - NFA$

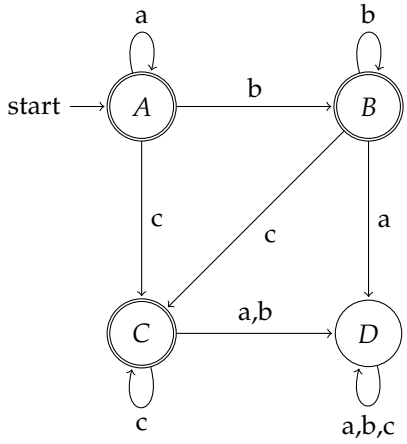


Figura 7: Automa DFA corrispondente all'$\epsilon - NFA$