# Calcolabilità e Linguaggi Formali

**892604 - Donald Gera**

# Indice

# 1 Introduzione

Nello studio della teoria della computabilità, abbiamo scoperto che esistono dei problemi che possiamo informalmente definire "facili" e altri che possiamo definire "difficili".
Più formalmente, quando parliamo di problemi "facili" facciamo riferimento ai problemi appartenenti alla classe computazionale P: tra gli altri, ricordiamo la ricerca di un minimo all'interno di una lista e il problema dell'ordinamento di un array. Al contrario, parlando di problemi "difficili" ci riferiamo ai problemi appartenenti alla classe computazionale NP: tra questi sappiamo esserci il problema SAT e il Problema dello Zaino (*Knapsack Problem*).
Generalizziamo la questione, distinguendo i problemi sulla base della loro risolvibilità: da un lato, troviamo i cosiddetti **problemi risolvibili**: a questa categoria appartengono sia i problemi P che i problemi NP, per i quali sappiamo che, anche se inefficiente, esiste ed è possibile implementare un algoritmo risolutivo; dall'altra, abbiamo i **problemi irrisolvibili**, per cui non è possibile trovare un algoritmo risolutivo.
Discriminare i problemi in base alla loro risolvibilità è compito dell'attore risolutivo, che in gergo tecnico prende il nome di **modello di computazione** o **computazionale**: un modello di computazione è un oggetto matematico che modella il funzionamento di un calcolatore; ne esistono diversi tipi, alcuni più potenti di altri, e la loro potenza è data dal grado di difficoltà dei problemi che riescono a risolvere.
Nello studio di questo corso vedremo il limite espressivo dei calcolatori, e come alcuni dei problemi irrisolvibili possono avere riscontri con problemi reali e applicazioni pratiche; un esempio è la verifica del software, ovvero la dimostrazione che un problema è privo di *bug*: tale problema non si può risolvere attraverso l'algoritmica classica, perché dire che un programma si comporta correttamente implica che si comporti correttamente con tutti i possibili input, che sono infiniti.

# 2 Linguaggi Regolari

**Definizione**

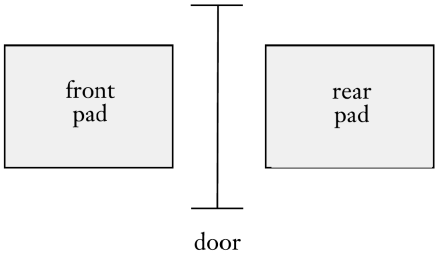
| Un **linguaggio** è un insieme di stringhe. |
|---|

## 2.1 Automi a Stati Finiti

**Definizione**

| Un **automa a stati finiti** è un modello computazionale deterministico. |
|---|

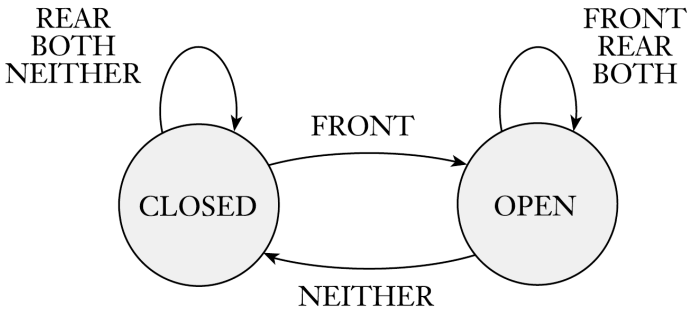
Descriviamo una situazione reale da cui possiamo derivare un automa a stati finiti: supponiamo di trovarci davanti ad una porta automatica, dotata di due sensori che rilevano la presenza di una persona sia da una parte che dall'altra.




La porta si apre quando rileva una persona attraverso il sensore anteriore, e si chiude quando non rileva persone attraverso entrambi i sensori.
Modelliamo la programmazione di questa porta attraverso un automa a stati finiti: per fare ciò, identifichiamo i seguenti tipi di input.

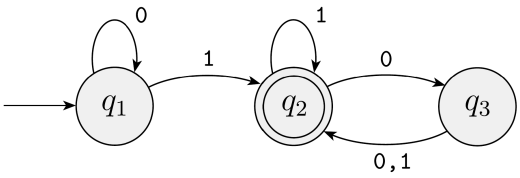
- FRONT: il sensore anteriore rileva la presenza di una persona.
- REAR: il sensore posteriore rileva la presenza di una persona.
- BOTH: entrambi i sensori rilevano la presenza di persone.
- NEITHER: nessun sensore rileva la presenza di persone.

Sulla base di questi input, la porta "rimbalza" tra due possibili stati: OPEN e CLOSED.



Tale meccanismo si può applicare per la codifica di programmi che gestiscono ascensori, macchinette del caffè, *et cetera*. In generale, possiamo usare questo tipo di "macchinetta" per tutti i dispositivi con memoria finita (da cui stati finiti), che aggiornano il proprio stato interno come risultato di una computazione.

Vediamo ora l'esempio di un automa a stati finiti reale.




Tale rappresentazione prende il nome di **diagramma di stato**. Rispetto al diagramma della porta, troviamo alcune differenze cosmetiche, tra cui il numero di stati e il numero di input, ma anche altre differenze sostanziali:

- Una freccia indica che $q_1$ è lo **stato iniziale**.
- Un doppio cerchio indica che $q_2$ è uno stato definito **accettante** o **finale**.

L'obiettivo del diagramma di stato è definire il comportamento di un automa a stati finiti in termini di input ricevuto. Supponiamo che l'automa riceva l'input 1101: partendo dallo stato iniziale $q_1$, l'automa "mangia" il primo 1, quindi si sposta verso lo stato $q_2$. Il prossimo carattere dell'input è un altro 1, quindi restiamo sullo stato $q_2$. Successivamente troviamo uno 0, quindi ci spostiamo su $q_3$. Infine, l'ultimo 1 porta a spostarci sullo stato $q_2$.

L'automa modella un programma che prende stringhe binarie, e dato che con l'input proposto finiamo su uno stato accettante, l'input è accettato.

Supponiamo di avere invece l'input 010: da $q_1$, "mangiando" il primo 0, rimaniamo su $q_1$. Troviamo successivamente un 1, per cui ci spostiamo su $q_2$. L'ultimo 0 ci porta a spostarci sullo stato $q_3$: l'input è terminato, e ci troviamo su uno stato che non è accettante, quindi questo automa non accetta tale input.

Analizzando il comportamento dell'automa in rapporto agli input ricevuti, possiamo dire che esso accetta tutte le stringhe che contengono almeno un 1 e che presentano un numero pari di 0 dopo l'ultimo 1.
Le stringhe binarie che presentano queste caratteristiche, dal momento che sono accettate dall'automa, prendono il nome di **linguaggio dell'automa**.

Osservando questo diagramma, possono sorgerci alcune domande.

1. È richiesto che gli automi a stati finiti abbiano un solo stato accettante?
2. Può un automa a stati finiti avere più stati iniziali?
3. Dato uno stato, ho sempre due frecce: una per il simbolo 0 e una per il simbolo 1. Ciò è richiesto o è un caso?

Occorre quindi dare una definizione formale di automi a stati finiti.

**Definizione**

Un automa a stati finiti deterministico, detto anche **DFA** (*Deterministic Finite Automaton*), è una quintupla $(Q, \Sigma, \delta, q_0, F)$, dove:

- $Q$ è un insieme finito di stati
- $\Sigma$ è un insieme finito di caratteri, detto **alfabeto**
- $\delta : Q \times \Sigma \to Q$ è la funzione di transizione
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati accettanti

Proviamo a descrivere il DFA visto precedentemente tramite la definizione formale appena data.

- $Q = \{q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- $q_0 = q_1$
- $F = \{q_2\}$
- $\delta$ è descritta nel seguente modo:

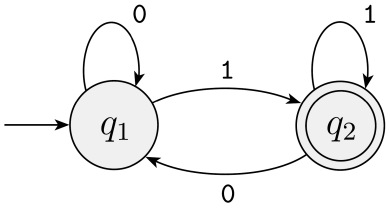
| | 0 | 1 |
|---|---|---|
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_3$ | $q_2$ |
| $q_3$ | $q_2$ | $q_2$, |

La definizione formale appena data ci permette di rispondere alle domande che ci siamo posti: un DFA non può avere più di uno stato iniziale, può avere più stati accettanti o nessuno, e dato uno stato e un simbolo di input bisogna sempre indicare lo stato a cui porta quel dato input.

### 2.1.1 Esempi di Automi a Stati Finiti

Vediamo qualche altro esempio. Per ciascuno di essi, verrà eseguita la conversione in una quintupla data dalla definizione formale, si daranno esempi di stringhe accettate e rifiutate e verrà definito il linguaggio riconosciuto da essi, cioè l'insieme delle stringhe accettate.

**Automa A.**




L'automa descritto da questo diagramma può essere riscritto nel seguente modo:

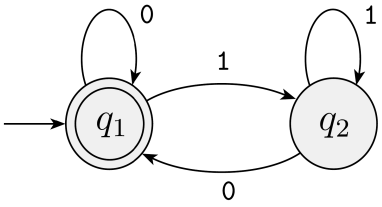
- $Q = \{q_1, q_2\}$
- $\Sigma = \{0, 1\}$
- $q_0 = q_1$
- $F = \{q_2\}$
- $\delta(q_1, 0) = q_1, \quad \delta(q_1, 1) = q_2, \quad \delta(q_2, 0) = q_1, \quad \delta(q_2, 1) = q_2$

Esempi di stringhe accettate sono, per esempio, 1 e 101, mentre esempi di stringhe rifiutate sono 0 e 000. Esiste anche un tipo di input speciale, che prende il nome di **stringa vuota** e viene identificato con $\varepsilon$, che rappresenta un input valido e viene accettato quando lo stato iniziale è uno stato accettante. Possiamo quindi definire il linguaggio riconosciuto dall'automa A:

$$L(A) = \{w \mid w \text{ è una stringa binaria che termina con } 1\}$$

Riconoscere il linguaggio di un automa è un procedimento complicato, che solitamente avviene per induzione sull'universo delle stringhe e avvalendosi di un'ipotesi che determini la proprietà delle stringhe accettate.

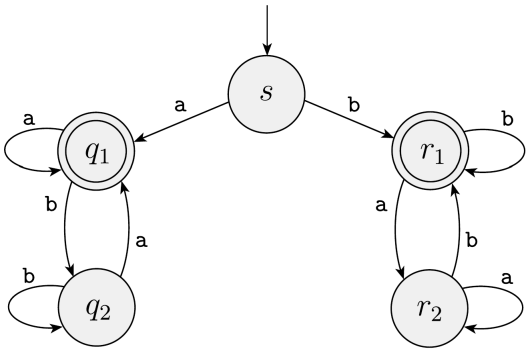
**Automa B.**




Notiamo che questo automa è del tutto uguale all'automa A, se non per lo stato accettante: ora infatti risulta accettante lo stato $q_1$ invece dello stato $q_2$, per cui B accetta tutte le stringhe che sono rifiutate da A e viceversa.
Possiamo definire in modo speculare anche il suo linguaggio:

$$L(B) = \{w \mid w \text{ è una stringa binaria vuota o che termina con } 0\}$$

**Automa C.**




L'automa descritto da questo diagramma può essere riscritto nel seguente modo:

- $Q = \{s,\, q_1,\, q_2,\, r_1,\, r_2\}$
- $\Sigma = \{a,\, b\}$
- $q_0 = s$
- $F = \{q_1,\, r_1\}$
- $\delta(s,a) = q_1, \quad \delta(s,b) = r_1, \quad \delta(q_1,a) = q_1, \quad \delta(q_1,b) = q_2$
  $\delta(q_2,a) = q_1, \quad \delta(q_2,b) = q_2, \quad \delta(r_1,a) = r_2, \quad \delta(r_1,b) = r_1, \; \delta(r_2,a) = r_2, \quad \delta(r_2,b) = r_1$

L'automa accetta le stringhe *bab* e *baab*, ma rifiuta le stringhe *abab* e *baa*. Il linguaggio di questo automa sarà quindi il seguente:

$$L(C) = \{w \mid w \text{ inizia e termina con lo stesso carattere}\}$$

**Definizione formale di Computazione**

Sia $M = (Q, \Sigma, \delta, q_0, F)$ un DFA e sia $w = w_1 w_2 \dots w_n$ una stringa costruita su $\Sigma$.
Diciamo che $M$ accetta $w$ se e solo se esiste una sequenza di stati $r_0$, $r_1$, …, $r_n \in Q$ tali che:

- $r_0 = q_0$
- $\forall i$, $\delta(r_i, w_{i+1}) = r_{i+1}$
- $r_n \in F$

**Definizione formale di Linguaggio**

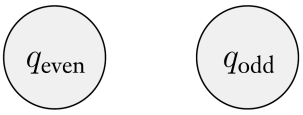
Il linguaggio di un DFA è l'insieme delle stringhe che accetta.

**Definizione formale di Linguaggio Regolare**

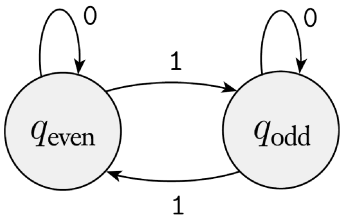
Un linguaggio $A$ è regolare se e solo se esiste un DFA $D$ tale che $L(D) = A$.

### 2.1.2 Progettazione di Automi a Stati Finiti

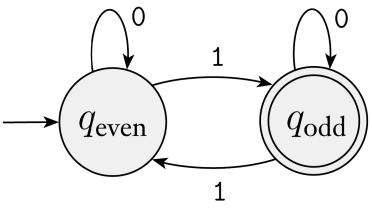
Supponiamo di voler creare un DFA che operi su $\Sigma = \{0, 1\}$ che accetti tutte e sole le stringhe con un numero dispari di 1.
Per prima cosa, identifichiamo i due stati, che ci determinano il numero di 1 contati fino ad ora.




Ora, definiamo i cambiamenti di stato sulla base dell'input ricevuto.



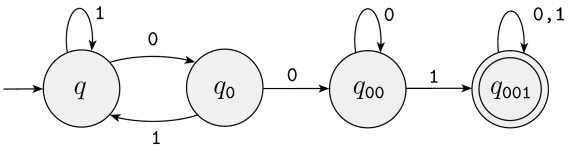

Infine, indichiamo lo stato iniziale e gli stati accettanti. In questo caso, ci sarà un solo stato accettante.



Disegniamo ora un automa che abbia come linguaggio l'insieme delle stringhe binarie che contengono la sottostringa 001.
Vogliamo definire quattro stati che identifichino le seguenti casistiche:

- Non è stato ancora trovato nessun simbolo del pattern.
- È stato trovato il carattere 0.
- Sono stati trovati i caratteri 00.
- Sono stati trovati i caratteri 001.

L'automa risultante sarà il seguente.




Quando ci troviamo nello stato $q_{00}$, ci conviene rimanere su tale stato fintantoché gli input continuano ad essere zeri. Inoltre, una volta che abbiamo trovato il pattern 001 non dobbiamo più uscire dallo stato accettante, in quanto l'automa ha completato il suo obbiettivo, per cui qualsiasi input in uscita da $q_{001}$ riporterà sempre a 001.

## 2.2 Operazioni Regolari

**Definizione**

Siano $A$, $B$ due linguaggi qualsiasi. Definiamo i seguenti operatori:

- Unione: $A \cup B = \{w \mid w \in A \vee w \in B\}$
- Concatenazione: $A \circ B = \{w_1 w_2 \mid w_1 \in A \wedge w_2 \in B\}$
- Star: $A^* = \{w_1 w_2 \dots w_k \mid \forall\, i : w_i \in A \wedge k \geq 0\}$

Ciò che abbiamo appena introdotto è un insieme di operazioni definite regolari, in quanto vengono effettuate sui linguaggi regolari. Dimostreremo che tali operazioni sono chiuse rispetto ai linguaggi regolari: ciò significa che se $A$ e $B$ sono regolari, allora anche il risultato di una qualche operazione regolare su $A$ e $B$ (*i.e.*, $A \cup B$, $A \circ B$, $A^*$ e $B^*$) è regolare.

Riportiamo degli esempi dell'utilizzo delle suddette operazioni regolari.
Se $A = \{a, b\}$ e $B = \{cc, d\}$, allora $A \cup B = \{a, b, cc, d\}$, $A \circ B = \{acc, ad, bcc, bd\}$ e $B^* = \{\varepsilon, cc, d, cccc, ccd, dcc, dd, \dots\}$.

**Teorema**

| Se $A$ e $B$ sono regolari, allora $A \cup B$ è regolare. |
|---|

**Dimostrazione**

Assumiamo che $A$ e $B$ siano regolari. Per definizione, esistono due DFA $M_1$ e $M_2$ tali che $L(M_1) = A$ e $L(M_2) = B$. Costruiamo un nuovo DFA $N$ tale che $L(N) = A \cup B$, lavorando su un caso ristretto: quello in cui $M_1$ e $M_2$ abbiano lo stesso alfabeto ($\Sigma_1 = \Sigma_2 = \Sigma$), e passeremo poi a generalizzare la dimostrazione.

$$M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$$
$$M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$$
$$N = (Q, \Sigma, \delta, q_0, F)$$

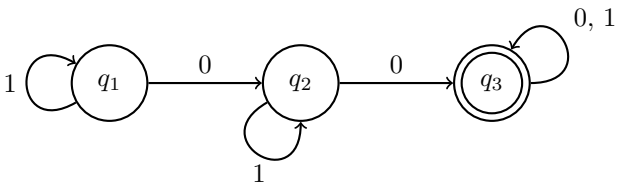
Vogliamo determinare $Q, \delta, q_0, F$:

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

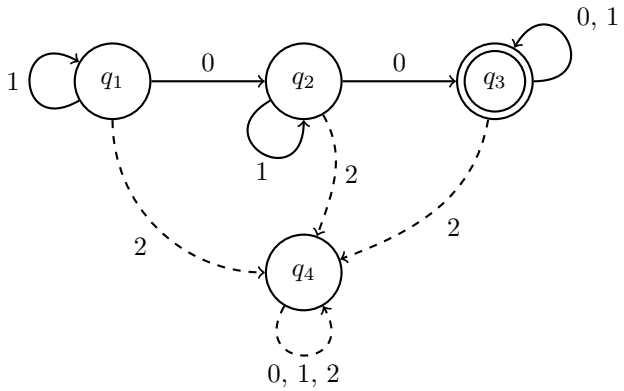
Proviamo a dare una definizione alternativa di $F$: se $F = F_1 \times F_2$, che indica l'insieme di coppie di stati in cui entrambi gli stati sono accettanti, siamo sicuri che $F$ rispetti le condizioni necessarie alla correttezza del nuovo automa, ma potremmo anche dire che la classe dei linguaggi regolari è chiusa anche rispetto all'operazione di intersezione, essendo $F$, secondo la nuova definizione, l'intersezione degli stati accettanti dei due automi di partenza.

**Generalizzazione**

Nel caso in cui $\Sigma_1 \neq \Sigma_2$, vogliamo trasformare $M$ e $N$ in automi equivalenti che operano sullo stesso alfabeto $\Sigma = \Sigma_1 \cup \Sigma_2$. Per fare ciò, possiamo ricorrere ad un trucco che prevede l'aggiunta di stati per ogni carattere non presente nell'alfabeto di partenza dell'automa.
Vediamo un esempio. Il seguente automa ha come alfabeto $\Sigma_1 = \{0, 1\}$




Vogliamo ora ridisegnare il nostro automa, utilizzando però un nuovo alfabeto $\Sigma_2 = \{0, 1, 2\}$. Per fare ciò, possiamo introdurre un nuovo stato che funga da "pozzo", sul quale termina l'esecuzione ogni volta che, da qualsiasi stato, si riceva il carattere 2. Tale stratagemma serve per estendere l'alfabeto garantendo tuttavia il funzionamento originale dell'automa di partenza.




Attuando tale trasformazione in un automa equivalente, possiamo applicare la dimostrazione compiuta precedentemente utilizzando l'ipotesi dell'uguaglianza tra i due alfabeti.

**Teorema**

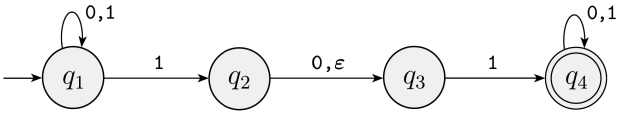
| Se $A$ e $B$ sono regolari, allora $A \circ B$ è regolare. |
|---|

Le conoscenze in nostro possesso non sono sufficienti per dimostrare tale teorema. Supponendo di voler utilizzare lo metodo messo in atto nella dimostrazione precedente, data una stringa come *aaabbaa* dovremmo trovare due automi che, dividendola in due sottostringhe, riconoscano ciascuna. Tuttavia, ci sono tanti possibili tagli che si possono effettuare sulla stringa, e gli automi visti finora non ci permettono di verificare tutte le possibili partizioni di una stringa.

Il motivo per cui tali automi non sono in grado di risolvere questo problema è dovuto alla loro natura, appunto, deterministica: dato uno stato e un input, l'automa si sposterà verso uno ed un solo preciso stato. Per effettuare tale dimostrazione, ma anche per affrontare una vasta gamma di problemi, è necessario introdurre un nuovo concetto, un nuovo modello di automa.

## 2.3 Automi a Stati Finiti Non Deterministici

Prima di dare una definizione formale di automa a stati finiti non deterministico, vediamone un esempio e cerchiamo di capire cosa lo distingue da un automa a stati finiti deterministico.




Notiamo subito delle differenze:

- Possiamo notare la presenza delle cosiddette $\varepsilon$-transizioni: un automa che si trova in uno stato che ha un arco uscente con valore $\varepsilon$ genera una nuova esecuzione, nella quale si sarà automaticamente spostato verso lo stato puntato da tale arco, e da cui continuerà a parsare l'input.

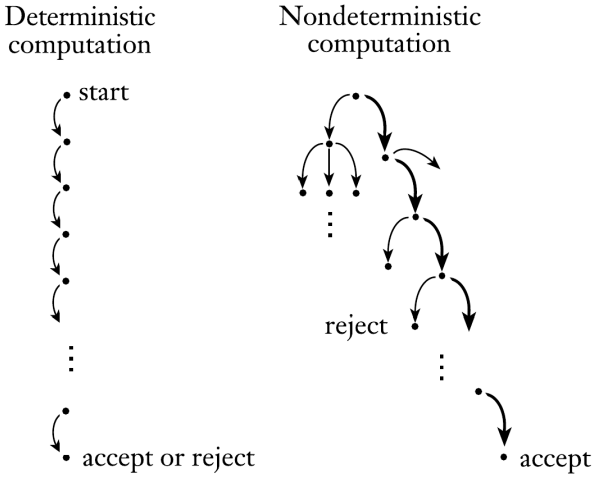
- Mancano alcune transizioni: da $q_2$ non ho transizioni per il simbolo 1; se l'automa si trova nello stato $q_2$ e riceve un 1, la computazione termina.
- Alcune transizioni possono portare in più stati: da $q_1$ ho due archi uscenti per il simbolo 1; se l'automa si trova nello stato $q_1$ e riceve un 1, vengono generate due diverse esecuzioni.

Le suddette osservazioni hanno anticipato molto la spiegazione della natura degli automi a stati finiti non deterministici. A differenza dei DFA, questo tipo di automi non prevede una singola esecuzione per una stringa passata in input, bensì possono esistere diverse esecuzioni di cui, necessariamente, qualcuna potrà culminare in uno stato accettante e qualcun'altra no. L'esistenza di un solo cammino che porti dallo stato iniziale allo stato accettante implica l'accettazione dell'input da parte dell'automa.

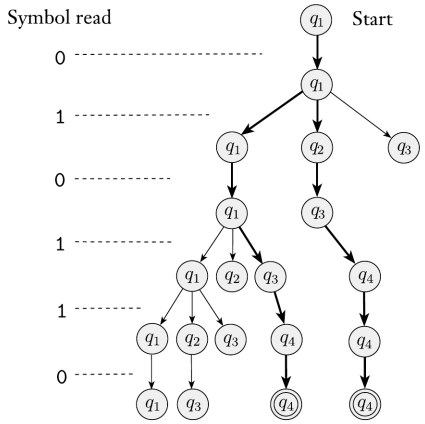
Per riferirci agli automi a stati finiti non deterministici possiamo usare il termine NFA, in maniera analoga al caso deterministico in cui per riferirci a tali automi usiamo il termine DFA.

Vediamo come si comporta l'automa riportato dato l'input 101. Tra le varie esecuzioni che l'automa può intraprendere, notiamo che esiste anche la seguente: $< q_1, q_2, q_3, q_4 >$; tale esecuzione culmina in uno stato accettante, per cui l'input 101 è accettato.
L'input 100 non viene invece accettato: dovendo parsare il primo 1, possiamo decidere o di rimanere in $q_1$ o di spostarci verso $q_2$, tuttavia in nessun caso uno 0 in input ci permette di raggiungere $q_4$; tendenzialmente, dimostrare che un input non viene accettato da questo tipo di automi è più difficile di dimostrare che viene accettato, in quanto per verificare che viene accettato basta trovare un'esecuzione che termina in uno stato accettante, mentre per verificare che non lo accetti occorre esaminare tutte le possibili esecuzioni.
La stringa 11 è una stringa che viene accettata dall'automa: parsando il primo 1 in input, l'automa passa da $q_1$ a $q_2$, successivamente per via della $\varepsilon$-transizione avviene il passaggio in $q_3$ e qui viene parsato l'ultimo 1, per cui si arriva in $q_4$.

Nel caso deterministico, la computazione avviene in modo lineare, e termina con l'accettazione o il rifiuto della stringa. Nel caso non deterministico, dobbiamo vedere la computazione come un albero in cui ciascun cammino radice-foglia rappresenta una diversa esecuzione: ci possono essere esecuzioni che rifiutano l'input e altre che lo accettano, e per affinché possiamo dire che l'automa accetta l'input è necessario che esista almeno un'esecuzione che termini con la sua accettazione.



Visualizziamo la computazione della stringa 010110 per l'automa a stati finiti non deterministico appena analizzato.

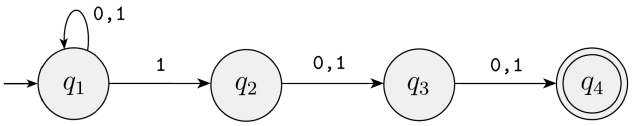



Automi a stati finiti deterministici e non deterministici hanno la stessa espressività: per definizione di automa a stati finiti deterministico, un DFA non è altro che un caso particolare di un NFA, e per questo motivo riconoscono le stesse classi di linguaggi.

### 2.3.1 Progettazione di Automi a Stati Finiti Non Deterministici

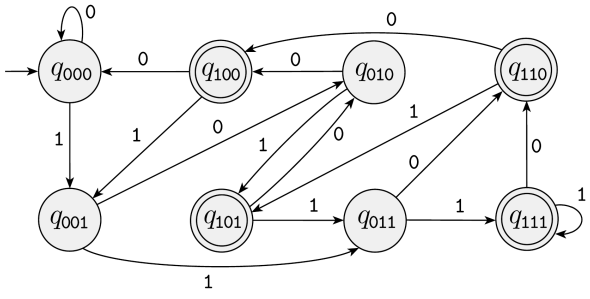
Si consideri il linguaggio delle stringhe binarie che hanno 1 come terzultimo carattere. Costruire un NFA e un DFA che lo riconoscano.

Iniziamo modellando il NFA. Sia DFA che NFA mangiano un carattere alla volta: non sanno quindi quante posizioni mancano alla fine della stringa, ma l'NFA ha il vantaggio di poter scegliere tra due possibilità: quando legge un 1, tale 1 è in terzultima posizione o non lo è. Forti di questa osservazione, possiamo modellare il seguente NFA.

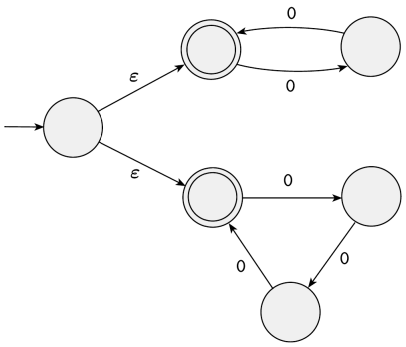



In questa rappresentazione, $q_2$ e $q_3$ sono stati ausiliari che servono per parsare il penultimo e l'ultimo carattere. Se si arriva all'ultimo stato, quello accettante, e non c'è più input da mangiare, allora l'input è accettato. Se invece si arriva allo stato accettante con ancora input da parsare, siccome tale stato non ha archi uscenti, l'esecuzione termina e non accetta l'input.

La conversione a DFA si può applicare cercando di mappare a gruppi gli ultimi input ricevuti. Quello che può capitare, nella conversione da NFA a DFA (vale la pena ricordare che per ogni NFA esiste sempre almeno un DFA che riconosce lo stesso linguaggio), è che il numero di stati del DFA sia di molto superiore al numero di stati nell NFA, e l'automa seguente ne è un esempio.



Vediamo un altro esempio di NFA.




L'alfabeto di questo NFA si dice **unario** per il fatto che è costituito da un solo carattere, lo 0. Ciò non deve indurci a pensare che per questo motivo accetti ogni stringa costituita da quel carattere: l'automa riconosce comunque un certo linguaggio, le cui stringhe sono accomunate da una certa proprietà. Scopriamo quale attraverso qualche esemmpio.

Innanzitutto, notiamo che la stringa 0 non viene accettata, in quanto nessuna delle tre diverse esecuzioni che non lo accetta.
Nella prima, in primo stato tenta di parsare lo 0, ma non esiste un arco uscente per il valore 0, per cui l'esecuzione termina.
Nella seconda, dallo stato iniziale viene effettuata una $\varepsilon$-transizione verso lo stato in alto. Una volta raggiunto, viene parsato lo 0, che porta in uno stato non accettante e l'esecuzione termina per la fine dell'input.
La terza esecuzione è analoga alla seconda, l'unica differenza è che l'$\varepsilon$-transizione viene effettuata verso lo stato in basso.

Al contrario, la stringa 00 viene accettata: effettuando una $\varepsilon$-transizione dallo stato iniziale verso lo stato in alto, ci troviamo nella situazione della seconda esecuzione che rifiuta lo 0, ma prima di terminare l'esecuzione viene parsato l'ultimo 0, che porta ad uno stato accettante.

La stringa 000 viene accettata secondo dinamiche simili a 00, l'unica differenza è che lo stato accettante è quello in basso.

Per gli stessi motivi, con l'alternarsi della parte superiore e inferiore dell'automa, vengono accettate le stringhe 0000, 000000, 00000000 *et cetera*, ovvero tutte le stringhe che hanno un numero di 0 pari o multipli di 3.

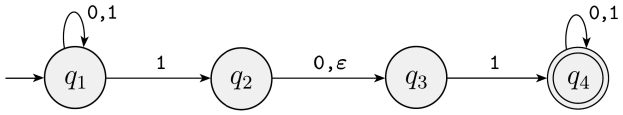
Siamo ora in grado di poter dare una definizione formale di NFA.

**Definizione**

Un automa a stati finiti non deterministico, detto anche **NFA** (*Nondeterministic Finite Automaton*), è una quintupla $(Q, \Sigma, \delta, q_0, F)$, dove:

- $Q$ è un insieme finito di stati
- $\Sigma$ è un insieme finito di caratteri, detto **alfabeto**
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathscr{P}(Q)$ è la funzione di transizione
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati accettanti

Nella definizione di $\delta$, $\mathscr{P}(Q)$ denota il *power set* di $Q$, cioè l'insieme di tutti i sottinsiemi di $Q$ (*i.e.*, se $Q = \{1, 2\}$, allora $\mathscr{P}(Q) = \big\{\emptyset, \{1\}, \{2\}, \{1, 2\}\big\}$).
Per capire il significato della definizione formale di NFA, e per renderci conto che la definzione è completamente analoga a quella del DFA, riscriviamo un NFA a noi già noto attraverso tale definizione.




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

| | 0 | 1 | $\varepsilon$ |
|---|---|---|---|
| $q_1$ | $\{q_1\}$ | $\{q_1, q_2\}$ | $\emptyset$ |
| $q_2$ | $\{q_3\}$ | $\emptyset$ | $\{q_3\}$ |
| $q_3$ | $\emptyset$ | $\{q_4\}$ | $\emptyset$ |
| $q_4$ | $\{q_4\}$ | $\{q_4\}$ | $\emptyset$, |

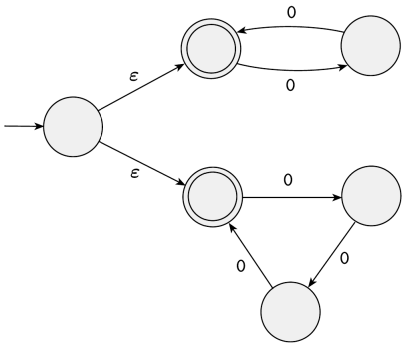
**Definizione formale di Computazione - Caso Non deterministico**

Sia $N = \{Q, \Sigma, \delta, q_0, F\}$ un NFA. Diciamo che $N$ accetta $w = w_0 w_1 \dots w_m$, dove $\forall\, i$, $w_i \in \Sigma_\varepsilon$, se e solo se esiste una sequenza di stati $r_0, r_1, \dots, r_m \in Q$ tali che:

- $r_0 = q_0$
- $r_m \in F$
- $\forall\, i,\ r_{i+1} \in \delta(r_i, w_{i+1})$

Osserviamo che, nel caso deterministico, $r_{i+1} = \delta(r_i, w_{i+1})$, mentre qui $r_{i+1} \in \delta(r_i, w_{i+1})$, in quanto lo stato successivo della sequenza non è il risultato della funzione di transizione ma è uno degli elementi che costituisce l'insieme risultato della funzione transizione.
Inoltre, un'altra differenza rispetto al caso deterministico consiste nel fatto che l'$i$-esimo carattere

di $w$ appartiene a $\Sigma_\varepsilon$, che non è che un altro modo per definire $\Sigma \cup \{\varepsilon\}$: nel caso non deterministico, infatti, ciascuna coppia di caratteri dell'input può essere inframmezzata dalla stringa vuota, e ciò ci aiuta a codificare la mossa $\varepsilon$ come se fosse un carattere del mio alfabeto. Per fare un esempio, stringhe che danno la stessa esecuzione di *abba* sono $a\varepsilon bba$, $ab\varepsilon ba\varepsilon$ e $\varepsilon a\varepsilon b\varepsilon b$.
Riprendiamo un altro NFA che abbiamo visto per aiutarci a capire il significato di questa proprietà.




Per dimostrare che la stringa 0000 è accettata dall'automa, dobbiamo innanzitutto riscrivere la stringa 0000 come $\varepsilon 0000$ e simuliamo la successione degli stati. A questo punto, la mossa $\varepsilon$ si comporta in modo analogo a qualsiasi altra mossa di parsing dell'input.

**Teorema**

| Qualsiasi NFA $N$ può essere convertito in un DFA $M$ tale che $L(M) = L(N)$. |
|---|

**Dimostrazione**

Sia $N = (Q, \Sigma, \delta, q_0, F)$ il NFA di partenza, costruiamo $M = (Q', \Sigma, \delta', q_0', F')$ tale che $L(M) = L(N)$. Per semplicità, assumiamo che $N$ non abbia $\varepsilon$-transizioni.
L'idea alla base di questa dimostrazione si basa sullo sfruttare un trucco che ci è già tornato utile nel dimostrare la chiusura dell'unione rispetto ai linguaggi regolari, cioè cambiare lo spazio degli stati; in questo caso, ci risulta utile considerare come spazio non $Q$, bensì il suo *power set*:

- $Q' = \mathscr{P}(Q)$
- $q_0' = \{q_0\}$, dove $\{q_0\}$ è un *singleton* (cioè un insieme con un solo elemento) per garantire la coerenza di tipi con $Q'$, che è appunto un insieme di insiemi
- $F' = \{R \in Q' \mid R \cap F \neq \emptyset\}$: $R$ è un insieme di stati appartenente a $Q'$, che ricordiamo essere l'insieme dei sottinsieme di $Q$, che contiene uno stato accettante per il primo automa
- $\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$

Il lato negativo di tale algoritmo di traduzione da NFA a DFA è che, siccome $Q'$ viene generato dal *power set* di $Q$, se $Q$ ha $k$ stati allora $Q'$ avrà $2^k$ stati.

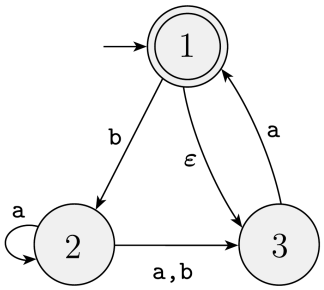
Per generalizzare la dimostrazione in presenza di $\varepsilon$-transizioni, definiamo la seguente funzione:

$$E(R) = \{q \in Q, \mid q \text{ può essere raggiunto da qualche } r \in R \text{ seguendo solo } \varepsilon\text{-transizioni}\}$$

Tale funzione prende il nome di $\varepsilon$-chiusura. Andiamo a modificare le definizioni di $\delta'$ e $q_0'$, servendoci della funzione appena definita.

- $q_0' = E(\{q_0\})$
- $\delta' = \bigcup_{r \in R} E(\delta(r, a))$

**Esempio di conversione** Vogliamo convertire il seguente NFA in un DFA equivalente.




Partiamo definendo lo stato iniziale del DFA. Siccome in questo caso esiste una $\varepsilon$-transizione, che in particolare parte dallo stato iniziale, lo stato iniziale del DFA verrà codificato dalla $\varepsilon$-chiusura dell'insieme degli stati iniziali del NFA, cioè lo stato 1: $E(\{1\}) = \{1, 3\}$. Tale stato è accettante, perché contiene lo stato 1 del NFA, che è accettante.

Se ci troviamo nello stato $\{1, 3\}$ e riceviamo il carattere $b$, siccome nel NFA dallo stato 3 non c'è nessuna transizione per $b$ ma dallo stato 1 sì, definiamo lo stato $\{2\}$. Se invece riceviamo il carattere $a$, si forma un cappio perché da 3 c'è una transizione che porta ad 1, e considerando la sua $\varepsilon$-chiusura riotteniamo lo stato $\{1, 3\}$.

Se ci troviamo nello stato $\{2\}$ e riceviamo il carattere $a$, siccome esistono due transizioni per $a$ che da 2 portano agli stati 2 e 3, codifichiamo il nuovo stato $\{2, 3\}$. Se invece riceviamo il carattere $b$, dobbiamo definire il nuovo stato $\{3\}$.

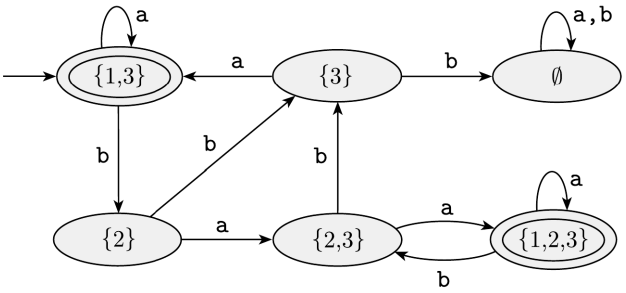
Se ci troviamo nello stato $\{3\}$ e riceviamo il carattere $a$, ritorniamo in $\{1, 3\}$ per lo stesso motivo della transizione da $\{1, 3\}$ a sé stesso. Se invece riceviamo il carattere $b$, andremo verso un nuovo stato codificato dall'insieme vuoto.

Se ci troviamo nello stato $\{2, 3\}$ e riceviamo il carattere $a$, dobbiamo codificare il nuovo stato $\{1, 2, 3\}$ che è accettante perché contiene lo stato $\{1\}$. Se invece riceviamo $b$, andremo verso lo stato $\{3\}$ per via della transizione da 2 a 3.

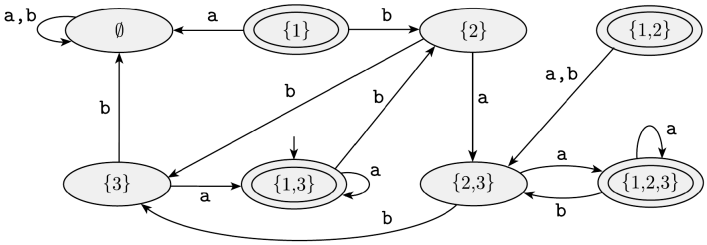
Se ci troviamo nello stato $\{1, 2, 3\}$ e riceviamo il carattere $a$, si forma un cappio. Se invece riceviamo il carattere $b$, andremo verso lo stato $\{2, 3\}$.

Per terminare la conversione, aggiungiamo un cappio per ogni carattere dell'alfabeto sullo stato definito dall'insieme vuoto.

Il risultato finale sarà il seguente.




Notiamo che nell'applicare la conversione abbiamo ottenuto un DFA con sei stati, invece dei $2^3 = 8$ stati che ci aspettavamo dalla definizione dell'algoritmo. Questo è avvenuto perché abbiamo applicato una versione ottimizzata dell'algoritmo, costruendo i nuovi stati man mano che venivano verificate le destinazioni delle transizioni. Gli stati che non abbiamo ottenuto sono degli stati che se fossero presenti sarebbero impossibili da raggiungere, per cui tanto vale non considerarli. Il risultato completo, considerando tali stati, è il seguente.




**Corollario**

| Un linguaggio $A$ è regolare se esiste un NFA $N$ tale che $L(N) = A$. |
|---|

**Dimostrazione**

Se abbiamo un linguaggio $A$ che è regolare, esiste un DFA che lo riconosce. Un DFA è un caso speciale di un NFA, per cui vale la proprietà.

## 2.4 Linguaggi regolari e operazioni regolari

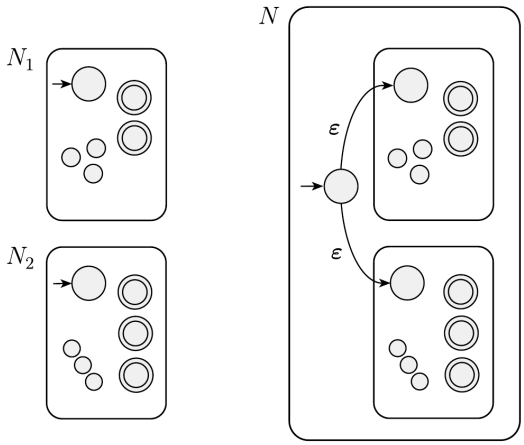
Utilizziamo le conoscenze appena acquisite per dimostrare tramite gli NFA una proprietà che abbiamo già dimostrato attraverso i DFA.

**Teorema**

| La classe dei linguaggi regolari è chiusa rispetto all'unione $\bigcup$. |
|---|

**Dimostrazione**

Siano $A$ e $B$ due linguaggi regolari. Allora, esistono due NFA $N_1$ e $N_2$ tali che $L(N_1) = A$ e $L(N_2) = B$. Costruiamo un nuovo NFA $N$ tale che $L(N) = A \cup B$.




L'automa che abbiamo appena costruito non sa a quale dei due automi di partenza appartiene il linguaggio che riceve, per cui prova in maniera non deterministica a parsare l'input attraverso entrambi tramite l'aggiunta di uno stato iniziale da cui può accedere ai due automi tramite due $\varepsilon$-transizioni.
Gli automi sono definiti nel seguente modo: $N_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1\}$, $N_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2\}$, $N = \{Q, \Sigma, \delta, q_0, F\}$, dove

- $Q = Q_1 \cup Q_2 \cup \{q_0\}$; è necessario specificare l'unione con $q_0$ perché $q_0 \notin Q_1 \cup Q_2$.
- $q_0$ è lo stato che abbiamo creato
- $F = F_1 \cup F_2$
- $\delta(q, a) = \begin{cases} \delta_1(q, a) & \text{se } q \in Q_1 \\ \delta_2(q, a) & \text{se } q \in Q_2 \\ \{q_1, q_2\} & \text{se } q = q_0 \wedge a = \varepsilon \\ \emptyset & \text{se } q = q_0 \wedge a \neq \varepsilon \end{cases}$

Procediamo a dimostrare la chiusura delle restanti operazioni regolari tramite l'utilizzo di NFA.

**Teorema**

| Se $A$ e $B$ sono regolari, allora $A \circ B$ è regolare. |
|---|

**Dimostrazione**

Poiché $A$ e $B$ sono regolari, allora esistono due NFA $N_1$ e $N_2$ tali che $L(N_1) = A$ e $L(N_2) = B$. L'idea alla base di questa dimostrazione è la medesima della precedente: vogliamo creare un nuovo automa $N$, che consista nella concatenazione di $N_1$ e $N_2$, in modo che attraverso una $\varepsilon$-transizione si possa passare dagli stati accettanti di $N_1$ allo stato iniziale di $N_2$.

Bisogna notare che nella creazione di $N$, gli stati che prima erano accettanti in $N_1$ ora non possono esserlo, in quanto l'automa riconoscerebbe anche i linguaggi solamente di $N_1$ mentre noi vogliamo riconoscere la concatenazione di stringhe appartenenti ai due linguaggi. Gli automi sono definiti nel seguente modo: $N_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1\}$, $N_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2\}$, $N = \{Q, \Sigma, \delta, q_0, F\}$, dove

- $Q = Q_1 \cup Q_2$
- $q_0 = q_1$
- $F = F_2$
- $\delta(q, a) = \begin{cases} \delta_1(q, a) & \text{se } q \in Q_1 \backslash F_1 \\ \delta_2(q, a) & \text{se } q \in Q_2 \\ \delta_1(q, a) \cup \{q_2\} & \text{se } q \in F_1 \wedge a = \varepsilon \text{ (potrebbero esserci transizioni da } q \text{ verso altri stati di } Q_1) \\ \delta_1(q, a) & \text{se } q \in F_1 \wedge a \neq \varepsilon \end{cases}$

**Teorema**

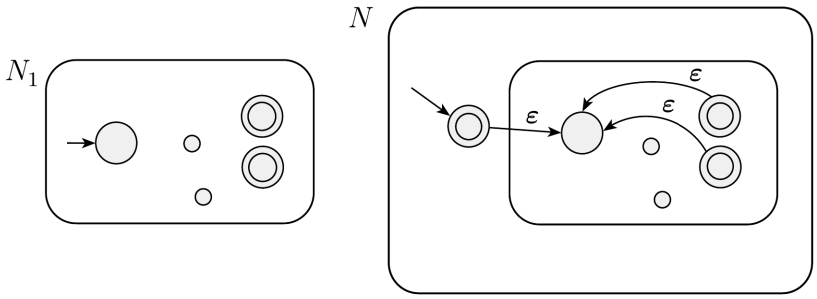
| Se $A$ è regolare, $A^*$ è regolare. |
|---|

Ricordiamo il risultato dell'operatore $*$:

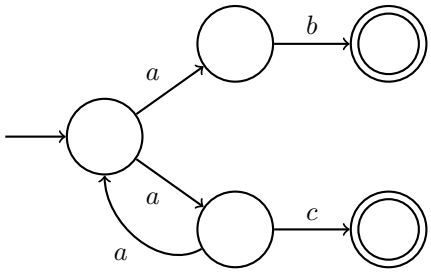
$$A^* = \{w_1 \dots w_n \,|\, n \geq 0 \wedge \forall i,\ w_i \in A\}$$

**Dimostrazione**

Dato che $A$ è regolare, esiste un NFA $N_1$ tale che $L(N_1) = A$. Vogliamo costruire un NFA $N$ tale che $L(N) = A^*$.
Avendo a che fare con un solo automa di partenza, non dobbiamo creare concatenazioni tra automi, ma il nuovo automa che dobbiamo progettare avrà come scopo quello di "reindirizzare" l'input dallo stato accettante, qualora dovesse essere raggiunto, allo stato iniziale tramite una $\varepsilon$-transizione, per generare una nuova esecuzione che verifichi l'accettazione di una nuova porzione di input.

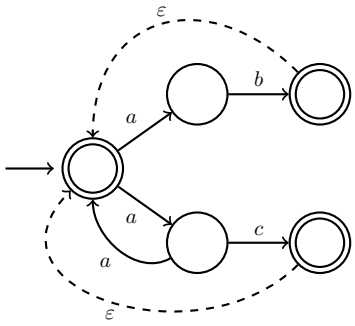



Avremmo potuto prendere $N_1$, rendere lo stato iniziale uno stato accettante, aggiungere delle $\varepsilon$-transizioni dagli stati accettanti verso di esso e terminare la dimostrazione, tuttavia ci dovessero essere delle transizioni verso lo stato iniziale (che noi abbiamo reso accettante), verrebbero accettate delle stringhe che non dovrebbero essere accettate. Vediamo un esempio.

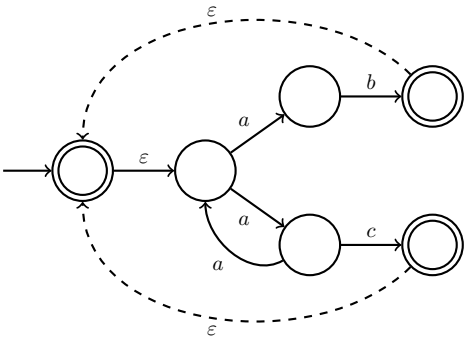



Il linguaggio di tale NFA è costituito dalle stringhe $ab$ e $ac$, precedute da un numero pari di $a$.

Applichiamo la trasformazione supposta:




Questo nuovo automa accetta la stringa $aa$, ma essa non può essere accettata perché l'automa di partenza non la accettava: se chiamiamo $A$ il linguaggio dell'automa, diciamo che $aa$ non fa parte di $A^*$. La trasformazione corretta che dobbiamo fare è la seguente.




Questa è la dimostrazione del professore, che prevede di eseguire delle $\varepsilon$-transizioni verso il nuovo stato iniziale inserito; il testo propone di indirizzare tali transizioni verso lo stato iniziale originale, tuttavia le due situazioni sono equivalenti dal momento che tra i due stati iniziali (quello appena aggiunto e quello originale) vi è una $\varepsilon$-transizione.
Formalizziamo i risultati ottenuti; siano $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ e $N = (Q, \Sigma, \delta, q_0, F)$, dove:

- $Q = Q_1 \cup \{q_0\}$
- $q_0$ è lo stato appena aggiunto
- $F = F_1 \cup \{q_0\}$
- $\delta(q, a) = \begin{cases} \delta_1(q, a) & \text{se } q \in Q_1 \backslash F_1 \\ \delta_1(q, a) \cup \{q_1\} & \text{se } q \in F_1 \ \wedge \ a = \varepsilon \\ \delta_1(q, a) & \text{se } q \in F_1 \ \wedge \ a \neq \varepsilon \\ \{q_1\} & \text{se } q = q_0 \ \wedge \ a = \varepsilon \\ \emptyset & \text{se } q = q_0 \ \wedge \ a \neq \varepsilon \end{cases}$

## 2.5 Espressioni Regolari

La definizione di espressione regolare viene data in maniera ricorsiva.

### Definizione

Una **Espressione Regolare** o ***regexp*** $R$ è definita come:
- $a \in \Sigma$, cioè un carattere dell'alfabeto $\Sigma$ su cui è definita
- $\varepsilon$, cioè la stringa vuota
- $\emptyset$
- $R_1 \cup R_2$ dove $R_1$, $R_2$ sono *regexp*
- $R_1 \circ R_2$ dove $R_1$, $R_2$ sono *regexp*
- $(R_1)^*$ dove $R_1$ è una *regexp*

Un esempio di *regexp* è $(a \circ b)^*$: per dimostrarlo, occorre dimostrare che $a \circ b$ è una *regexp*, ma la concatenazione rientra nei casi della definizione e $a$ e $b$ costituiscono i casi base della definizione, in quanto sono due caratteri appartenenti all'alfabeto $\Sigma$.

### Semantica delle *regexp*

- $L(a) = \{a\}$
- $L(\varepsilon) = \{\varepsilon\}$
- $L(\emptyset) = \emptyset$
- $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$
- $L(R_1 \circ R_2) = L(R_1) \circ L(R_2)$
- $L(R_1^*) = L(R_1)^*$

### Esempi

- $0^*10^*$: $L(0^*10^*) = L(0^*) \circ L(10^*) = L(0^*) \circ L(1) \circ L(0^*) = L(0)^* \circ L(1) \circ L(0)^* = \{0\}^* \circ \{1\} \circ \{0\}^*$
  Tale espressione regolare indica una sequenza arbitraria di 0, seguita da un 1, seguito da una sequenza arbitraria di 0. In altre parole, è l'insieme delle stringhe binarie che comprendono esattamente un 1 (1, 0010, 1000, 01000 *et cetera*).
- $(0 \cup \varepsilon)1^*$: tale espressione regolare accetta una stringa arbitrariamente lunga di 1, preceduta eventualmente da un carattere 0.
- $(0 \cup \varepsilon)(1 \cup \varepsilon)$: tale espressione regolare riconosce le seguenti quattro stringhe: $\{\varepsilon, 0, 1, 01\}$.
- $L(\emptyset^*) = L(\emptyset)^* = \emptyset^* = \{\varepsilon\}$; ricordiamo che l'operazione $*$ comprende 0 o più occorrenze del pattern, per cui la stringa vuota è sempre contenuta.
- $L(1^*\emptyset) = L(1^*) \circ L(\emptyset) = L(1)^* \circ L(\emptyset) = \{1\}^* \circ \emptyset = \emptyset$; riprendiamo la definizione di concatenazione:
  $$A \circ B = \{w_1 w_2 \mid w_1 \in A \wedge w_2 \in B\}$$
  Per cui, se l'insieme $B$ è vuoto, non c'è nessun elemento da concatenare che rispetti la proprietà della concatenazione, dunque il risultato sarà l'insieme vuoto.

## 2.6 Equivalenza espressiva di *regexp* e automi

**Teorema**

| $A$ è regolare se e solo se esiste una espressione regolare $R$ tale che $L(R) = A$. |
|---|

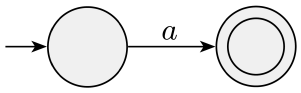
**Dimostrazione**

Dal momento che il teorema sancisce una doppia implicazione, la dimostrazione avviene dimostrando entrambe le implicazioni coinvolte.

**Lemma** Se $A$ è descritto da una espressione regolare $R$, allora $A$ è regolare.

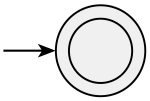
**Dimostrazione** Sia $R$ tale che $L(R) = A$. Costruisco un NFA $N$ tale che $L(N) = A$.
Procediamo per induzione sulla struttura di $R$, cioè sul numero e sulla natura dei simboli di $R$. Distinguiamo quindi i diversi casi:

- $R = a$ per qualche $a \in \Sigma$. Tale NFA è molto facile da progettare:

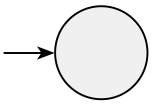



  Notiamo che l'automa in questione è un NFA e non un DFA perché lo stato accettante non ha archi uscenti, tuttavia è possibile trasformarlo in un DFA equivalente in quanto sappiamo che DFA e NFA hanno la stessa potenza espressiva.

- $R = \varepsilon$. Anche il NFA che descrive questa *regexp* è elementare:

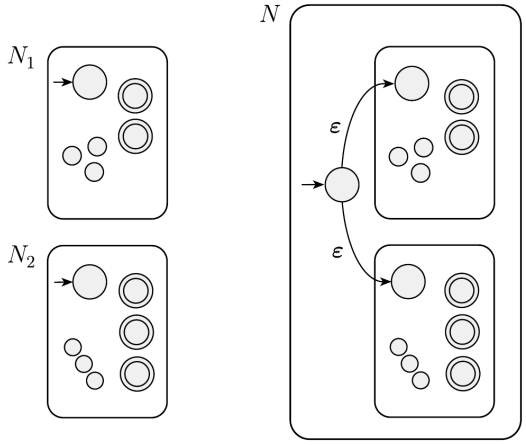


- $R = \emptyset$. L'automa che descrive questo scenario è analogo al precedente, tuttavia non permette l'accettazione del linguaggio vuoto:



- $R = R_1 \cup R_2$ per qualche $R_1$, $R_2$. L'osservazione che possiamo fare è che le strutture di $R_1$ e di $R_2$ sono più piccole di quella di $R$: per ipotesi induttiva, sia $R_1$ che $R_2$ sono regolari, quindi esistono due NFA $N_1$ e $N_2$ tali che $L(N_1) = L(R_1)$ e $L(N_2) = L(R_2)$.
  A questo punto, la dimostrazione procede in modo analogo ad una dimostrazione che abbiamo già visto: quella della chiusura dei linguaggi regolari rispetto all'operazione di unione.

Il nostro obiettivo è dimostrare l'esistenza di un NFA che riconosce come linguaggio l'unione dei linguaggi di $N_1$ e $N_2$:



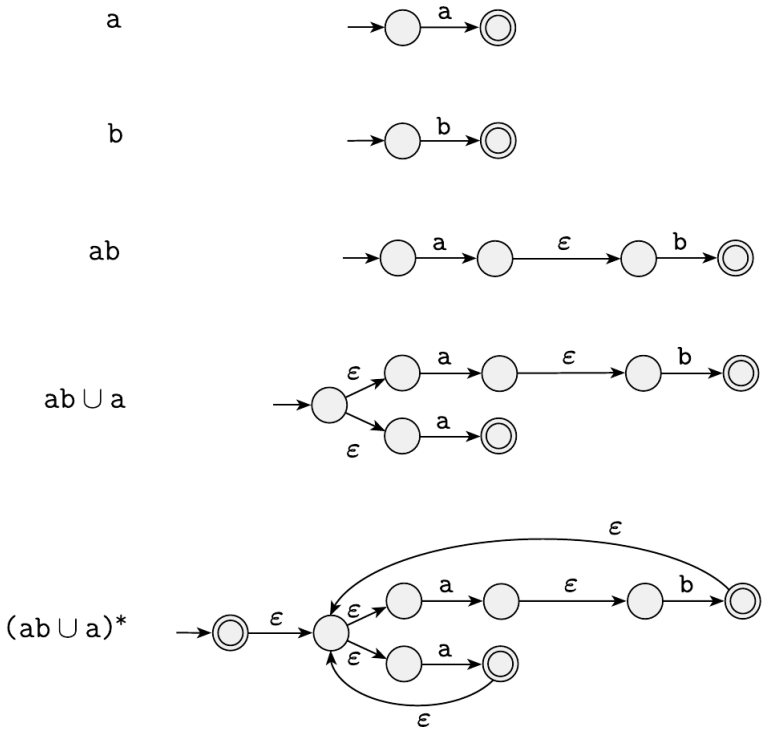

Il nuovo automa costruito, $N$, avrà il seguente linguaggio:

$$\begin{aligned} L(N) &= L(N_1) \cup L(N_2) \\ &= L(R_1) \cup L(R_2) \\ &= L(R_1 \cup R_2) \\ &= L(R) \end{aligned}$$

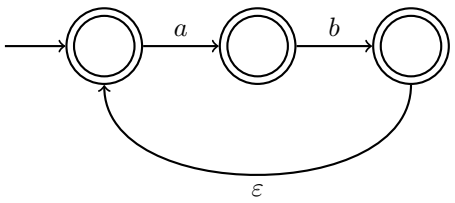
- $R = R_1 \circ R_2$ per qualche $R_1$, $R_2$. Per ipotesi induttiva sia $R_1$ che $R_2$ sono regolari, quindi esistono due NFA $N_1$ e $N_2$ tali che $L(N_1) = L(R_1)$ e $L(N_2) = L(R_2)$. Vogliamo costruire un nuovo NFA $N$ tale che $L(N) = L(R)$.
  In modo analogo a quanto svolto per il punto precedente, avendo verificato che l'insieme dei linguaggi regolari è chiuso rispetto all'operazione di concatenazione, procediamo alla dimostrazione.
  $$\begin{aligned} L(N) &= L(N_1) \circ L(N_2) \\ &= L(R_1) \circ L(R_1) \\ &= L(R_1 \circ R_2) \\ &= L(R) \end{aligned}$$

- $R = R_1^*$ per qualche $R_1$. La dimostrazione di questo caso è analoga a quella dei casi precedenti.

**Conversione da *regexp* a NFA** Supponiamo di voler tradurre l'espressione regolare $(ab \cup a)^*$ in un NFA. Applicando le nozioni apprese nella dimostrazione del lemma di cui sopra, oltre alle conoscenze assorbite nella dimostrazione della chiusura dei linguaggi regolari rispetto alle tre operazioni regolari, possiamo procedere passo dopo passo per codificare in un automa le singole componenti dell'espressione regolare.

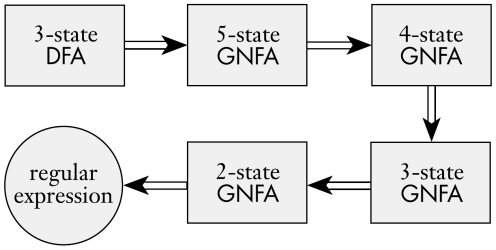



Tale automa non è minimo: sono presenti molti stati superflui, tuttavia sono stati mantenuti nella conversione per esemplificare passo passo lo svolgimento dell'applicazione dell'algoritmo.
Un automa più piccolo che descrive lo stesso linguaggio è il seguente:



**Lemma** Se un linguaggio è regolare, allora esiste una *regexp* che lo descrive.

**Dimostrazione** Sia $A$ regolare. Vogliamo dimostrare che esiste una *regexp* $R$ tale che $L(R) = A$. Poiché $A$ è regolare, esiste un DFA $M$ tale che $L(M) = A$. Occorre convertire $M$ in una *regexp* equivalente.
Nella dimostrazione del lemma precedente abbiamo verificato per induzione sulla struttura della *regexp* che è possibile costruire un NFA equivalente ad una *regexp* data; ora, dobbiamo fare il contrario: dobbiamo trovare un algoritmo che prenda in input una quintupla, costituita dai cinque elementi che abbiamo visto costituire un NFA, e restituisca una *regexp* equivalente. È un'operazione più complicata da compiere, perché la sintassi di una *regexp* è costituita da un totale di sei casi, mentre la sintassi di un automa a stati finiti è più complessa.
Per questo motivo, prima di effettuare la conversione vera e propria in *regexp*, ci conviene effettuare una prima conversione del DFA di partenza in un GNFA (*Generalized Nondeterministic Finite Automaton*, ovvero Automa a Stati Finiti Non Deterministico Generalizzato), e poi tradurre tale GNFA in una *regexp*. Un GNFA è un NFA in cui le transizioni sono etichettate con delle *regexp*.
Tendenzialmente, un DFA avente $k$ stati viene convertito in un GNFA avente $k + 2$ stati; tramite un ulteriore algoritmo viene generato un nuovo GNFA utilizzando il GNFA di partenza, avente però uno stato in meno. Si può applicare questo algoritmo $k - 1$ volte per arrivare ad ottenere un GNFA con due stati, lo stato iniziale e lo stato accettante, e avente come corpo della transizione la nostra *regexp*.

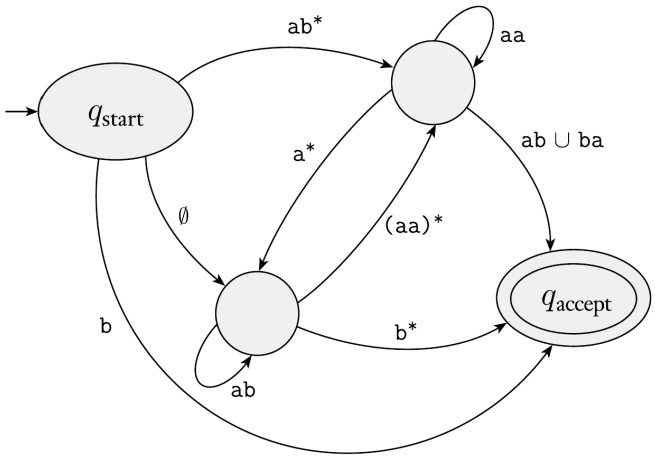



**Definizione**

Un GNFA è un NFA etichettato con *REGEXP*. Per il nostro obiettivo, richiediamo le seguenti condizioni:

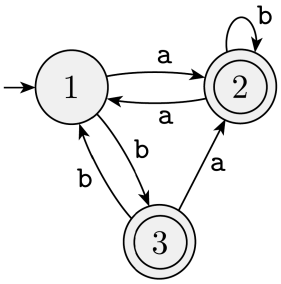
1. Lo stato iniziale ha solo transizioni in uscita verso tutti gli altri stati.
2. Abbiamo un solo stato accettante che è distinto dallo stato iniziale e ha solo transizioni in entrata da tutti gli altri stati.
3. Ogni stato è collegato a tutti gli stati (compreso sé stesso), fatta eccezione per gli stati iniziale e finale.

Vediamo un esempio di GNFA.

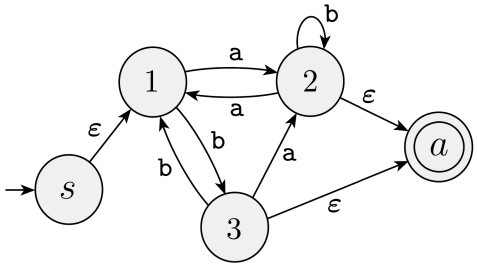



È possibile notare come, a differenza degli automi che abbiamo visto finora, i GNFA parsano l'input prendendo più di un carattere alla volta, identificando pattern di sottostringhe presenti nell'input sulla base delle *regexp* presenti sulle transizioni in uscita.

Dato un DFA, possiamo convertirlo in un GNFA equivalente che rispetti le tre condizioni desiderate. Proviamo a convertire il seguente DFA.

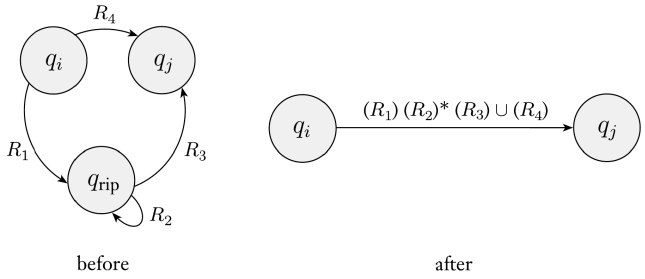



Per cominciare, creiamo un nuovo stato iniziale che presenta una $\varepsilon$-transizione verso lo stato 1 e delle $\emptyset$-transizioni verso tutti gli altri stati (omesse nel disegno), in modo da non avere transizioni verso lo stato iniziale, e un nuovo stato accettante, che è raggiungibile dagli stati 2 e 3 (che non saranno più accettanti) tramite $\varepsilon$-transizioni. L'aggiunta di questi stati è il motivo per cui il GNFA risultato della conversione ha due stati in più.

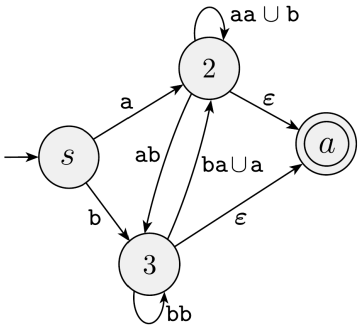


Quello che abbiamo ottenuto è effettivamente un GNFA a 5 stati. Analizzeremo ora un algoritmo che, preso un GNFA con $k > 2$ stati, restituisce un nuovo GNFA con $k - 1$ stati. L'iterazione di questo algoritmo tante volte quante sono gli stati diversi dallo stato iniziale ci procurerà la nostra espressione regolare desiderata; tale algoritmo, ogni volta che viene eseguito, prevede l'eliminazione di uno stato diverso, appunto, da quello iniziale e da quello finale.

Vediamo in che modo viene applicato questo algoritmo. Dato l'automa a sinistra, supponiamo di voler eliminare lo stato $q_{rip}$. Per farlo, dobbiamo considerare tutti i cammini che ci permettono di arrivare da $q_i$ a $q_j$: chiaramente potremmo arrivarci tramite la transizione che prevede il riconoscimento di $R_4$, tuttavia non possiamo ignorare il cammino dato dal riconoscimento di $R_1$ (per arrivare a $q_{rip}$), di un numero arbitrario di riconoscimenti di $R_2$ (il cappio di $q_{rip}$) e dal riconoscimento di $R_3$ (la transizione da $q_{rip}$ a $q_j$). Il risultato della rimozione di $q_{rip}$ prevede la generazione dell'automa a destra.




Forti di ciò, procediamo alla conversione del nostro DFA in una *regexp*. Scegliamo arbitrariamente di rimuovere lo stato 1: dallo stato iniziale si può arrivare agli stati 2 e 3 passando per 1 riconoscendo, rispettivamente, i caratteri $a$ e $b$, per cui lo stato iniziale avrà le stesse transizioni in uscita dello stato 1. Inoltre, dallo stato 2 si può arrivare allo stato 3 passando per lo stato 1, per cui ci sarà una transizione dallo stato 2 allo stato 3 avente come etichetta $a \cup b$; analogamente, siccome dallo stato 3 si può arrivare allo stato 2 o passando per 1 (riconoscendo prima $b$ e poi $a$) o direttamente tramite una transizione etichettata con $a$: da 3 a 2 sarà presente una nuova transizione avente etichetta $ba \cup a$. Inoltre, 2 e 3 sono raggiungibili da sé stessi passando per 1: si formeranno dei cappi su 2 e 3 aventi rispettivamente le etichette $aa \cup b$ (c'era già un cappio etichettato con $b$) e $bb$. Il risultato dopo l'eliminazione di 1 è il seguente.



Gli altri passi della conversione sono analoghi e meccanici. Scegliamo di rimuovere lo stato 2, e otteniamo il seguente GNFA.

L'ultima applicazione dell'algoritmo, per rimuovere lo stato 3, ci dà come etichetta dell'unica transizione la *regexp* desiderata.

La dimostrazione della correttezza dei due lemmi, cioè delle due implicazioni coinvolte nel teorema di equivalenza di DFA ed espressioni regolari, dimostra la correttezza stessa del teorema, come volevasi dimostrare.

Forniamo, per completezza, la definizione formale di GNFA.

**Definizione**

Un **GNFA** (*Generalized Nondeterministic Finite Automaton*) è una quintupla $(Q, \Sigma, \delta, q_{start}, q_{accept})$ tale che:

- $Q$ è un insieme finito di stati
- $\Sigma$ è un alfabeto finito su cui costuiremo le *regexp*
- $\delta : (Q \backslash \{q_{accept}\}) \times (Q \backslash \{q_{start}\}) \to RE_\Sigma$, dove $RE_\Sigma$ indica una *regexp* costruita su $\Sigma$
- $q_{start} \in Q$ è lo stato iniziale
- $q_{accept} \in Q$ è lo stato accettante

Tale GNFA accetta una stringa $w$ costruita su $\Sigma$ se $w = w_1 \dots w_k$ ed esiste una sequenza di stati $q_0 \dots_k$ tale che:

- $q_0$ è lo stato iniziale
- $q_k$ è lo stato finale
- $\forall\, i,\ w_i \in L(\delta(q_{i-1}, q_i))$

# 3 Linguaggi Non Regolari

## 3.1 Definizione e *pumping lemma*

Un **linguaggio non regolare** è un linguaggio che non può essere riconosciuto da un automa a stati finiti. Vediamo subito un esempio.

$$\{0^n 1^n \mid n \geq 0\}$$

Tale linguaggio identifica una sequenza arbitraria di 0, seguita da una sequenza della stessa lunghezza di 1. Tale linguaggio non è regolare, perché un automa a stati finito ha bisogno di memorizzare quanti zeri della sequenza di lunghezza arbitraria ha letto finora, ma non è in grado di farlo; siccome il numero di zeri non è limitato, non è possibile tenere traccia di potenzialmente infiniti zeri.

Per capire se un linguaggio è regolare o meno, si utilizza la tecnica del ***pumping lemma***.

### Definizione

Se $A$ è un linguaggio regolare, allora esiste un numero naturale $p$, detto ***pumping length***, tale che se $s$ è una stringa di $A$ di lunghezza almeno $p$, allora $s$ può essere scritta nella forma $xyz$ tali che:

1. $\forall\, i \geq 0,\ xy^i z \in A$
2. $|y| > 0$
3. $|xy| \leq p$

### Dimostrazione

Sia $A$ regolare. Allora, esiste un DFA $M = (Q, \Sigma, \delta, q_0, F)$ tale che $L(M) = A$. Determino $p = |Q|$, cioè il numero di stati di $M$. Consideriamo ora una stringa $s = s_1 \dots s_n$, con $n \geq p$, tale che $s \in A$. Poiché questa stringa è accettata da $M$, esiste una sequenza di stati $r_1 \dots r_{n+1}$ tali che $r_1 = q_0$, $r_{n+1} \in F$ e $\forall\, i,\ r_{i+1} = \delta(r_i, s_i)$.
Tale sequenza avrà la seguente forma: $< r_1, \dots, r_p, r_{p+1}, \dots, r_{n+1} >$; siccome la sequenza comprende lo stato accettante e poi uno stato per ogni elemento dell'input, che è lungo $n$, vuol dire che avrà lunghezza totale $n + 1$, e poiché $n \geq p$, avrà lunghezza pari almeno a $p + 1$. Poiché $n + 1 \geq p + 1$, ci deve essere in questa sequenza almeno uno stato che si ripete. In particolare,

$$\exists\, j, l : j < l \wedge l \leq p + 1 \wedge r_l = r_j$$

A questo punto, poniamo $x = s_1 \dots s_{j-1}$, $y = s_j \dots s_{l-1}$ e $z = s_l \dots s_n$. Quello che avviene all'interno di un DFA che parsa una stringa che rispetta tali condizioni viene descritto nella seguente figura.

Gli stati presenti nella figura cosituiscono parti della sequenza identificata sopra: $q_1 = r_1$, $q_9 = r_j$ e $q_{13} = r_{n+1}$. Quello che succede è che si forma un cappio su $r_j$, e le stringhe parsate con l'attraversamento di eventuali altri stati di questo cappio fa parte di $A$.

In sintesi, il lemma sostiene che una stringa sufficientemente lunga implica la ripetizione di uno stato e poiché questo stato si ripete possiamo prendere il *loop* attorno tale stato 0 o più volte e rimanere sempre nello stesso linguaggio.

### Applicazione del *pumping lemma*

Il *pumping lemma* può essere usato per dimostrare che un linguaggio non è regolare: supponiamo di voler dimostrare che un certo linguaggio $B$ non sia regolare. La procedura avviene secondo diversi passi:

1. Assumiamo per assurdo che $B$ sia regolare.
2. Poiché è regolare, deve valere il *pumping lemma*.
3. Trovo $s \in B$ tale che $|s| \geq p$ e che non soddisfa le condizioni del *pumping lemma*.
4. Deduco che $B$ non è regolare.

**Esempio B.** Usiamo tale schema per mostrare un esempio della sua applicazione, volendo dimostrare che il linguaggio $B = \{0^n 1^n \mid n \geq 0\}$ non è regolare. Assumiamo per assurdo che sia regolare, e sia $p$ la sua *pumping length*. Costruisco $s$ che viola il *pumping lemma*: $s = 0^p 1^p$. Tale stringa appartiene al linguaggio, e in più è lunga almeno $p$, in particolare $2p$. Spezziamo $s$ in tre sottostringhe: $s = xyz$. Conviene ragionare per casi su $y$:

- $y$ contiene solo 0: $xy^2z$ ha forma $0^{p+k}1^p \notin B$, per $k > 0$ che è lunghezza di $y$
- $y$ contiene solo 1: $xy^2z$ ha forma $0^p 1^{p+k} \notin B$, per $k > 0$ che è lunghezza di $y$
- $y$ contiene sia 0 che 1: in questo caso violo la condizione di $B$ che gli 1 non precedono gli 0.

$$\underbrace{00}_{x}\underbrace{01}_{y}\underbrace{11}_{z} \overset{\text{pumping up}}{\rightarrow} \underbrace{00}_{x}\underbrace{01}_{y}\underbrace{01}_{y}\underbrace{11}_{z}$$

**Esempio C.** Vediamo un altro esempio. Prendiamo il seguente linguaggio:

$$C = \{w \mid w \text{ ha lo stesso numero di 0 e di 1}\}$$

Assumiamo per assurdo che $C$ sia regolare, e sia $p$ la sua *pumping length*. Prendiamo come candidato la stringa $s = 0^p 1^p$: è un buon candidato, perché appartiene al linguaggio ed è lunga almeno $p$.
Consci dei casi visti nell'applicazione precedente, potremmo pensare di fare la seguente divisione: $x = \varepsilon$, $y = (0^p 1^p)$ e $z = \varepsilon$; ad un primo impatto non sembra violare il *pumping lemma*, in quanto ora non ci interessa la posizione degli 1 e degli 0. Tuttavia, tale divisione viola il terzo punto del *pumping lemma*: $|xy| \leq p$.
Poniamo $s = xyz$: osserviamo che, poiché $|xy| \leq p$, $y$ è composta solo da 0. Ma allora $xy^2z$ ha forma $0^{p+k}1^p$ con $k > 0$. Questa stringa non può fare parte di $C$, perché presenta più 0 che 1.

**Esempio D.** Verifichiamo che anche il linguaggio $D = \{1^{n^2} \mid n \geq 0\}$ non è regolare. Assumiamo per assurdo che $D$ sia regolare e sia $p$ la sua *pumping length*. Un candidato ragionevole è $s = 1^{p^2}$: appartiene al linguaggio ed è lunga almeno $p$, in particolare è lunga $p^2$. Dividiamo $s$ in $xyz$ e

ragioniamo sulla sua lunghezza:

$$\begin{aligned}\underbrace{|xy^2z|}_{>0} = |xyyz| &= \underbrace{|xyz|}_{p^2} + \underbrace{|y|}_{\leq p,\ \text{poiché } |xy|\leq p} \\ &\leq p^2 + p \\ &< p^2 + 2p + 1 \\ &= (p+1)^2\end{aligned}$$

Poiché $|xyz| = p^2$ e $|y| > 0$, e siccome $|xy^2z| = |xyz|+|y|$, allora avremo la seguente disuguaglianza:

$$p^2 < |xy^2z| < (p+1)^2$$

Di conseguenza, $|xy^2z|$ non è un quadrato perfetto, per cui non appartiene al linguaggio.

**Esempio E.** Vediamo un ulteriore esempio, dimostrando che il linguaggio $E = \{0^i 1^j \mid i > j\}$ non è regolare. Assumiamo per assurdo che $E$ sia regolare e sia $p$ la sua *pumping length*. Un candidato ragionevole è $s = 0^{p+1}1^p = xyz$. Per la terza condizione del *pumping lemma*, $y$ si compone solo di 0. In questo caso, $xy^2z \in E$: fare *pumping up* in questo caso non aiuta. Tuttavia, la stringa $xy^0z \notin E$: fare *pumping down* in questo caso è l'unica possibilità, perché l'unico modo per violare la proprietà qui è diminuire il numero di 0. Non sappiamo quanti ne abbiamo tolti, ma potenzialmente ne basta anche uno per violare la proprietà.

**Esempio F.** Concludiamo dimostrando che il linguaggio $F = \{ww \mid w \in \{0, 1\}^*\}$ non è regolare. Supponiamo per assurdo che lo sia, e sia $p$ la sua *pumping length*. Poniamo $s = 0^p10^p1$: $s$ fa parte di $F$ ed è più lunga di $p$, per cui è un candidato valido. Il *pumping lemma* garantisce che $s$ possa essere divisa in tre sottostringhe $xyz$ che soddisfano le tre condizioni del lemma.
Anche in questo caso, la terza condizione è fondamentale per dimostrare l'assurdo: affinché $|xy| < p$, $y$ deve essere costituita da soli 0; fare *pumping up* o *down* di $y$ porterebbe quindi alla rottura del pattern definito da $ww$, per cui $xyyz \notin F$. Di conseguenza, $F$ non è regolare.

## 3.2 Esercizi su linguaggi regolari e non

**Esercizio 1** Sia $D$ l'insieme delle stringhe tali che:

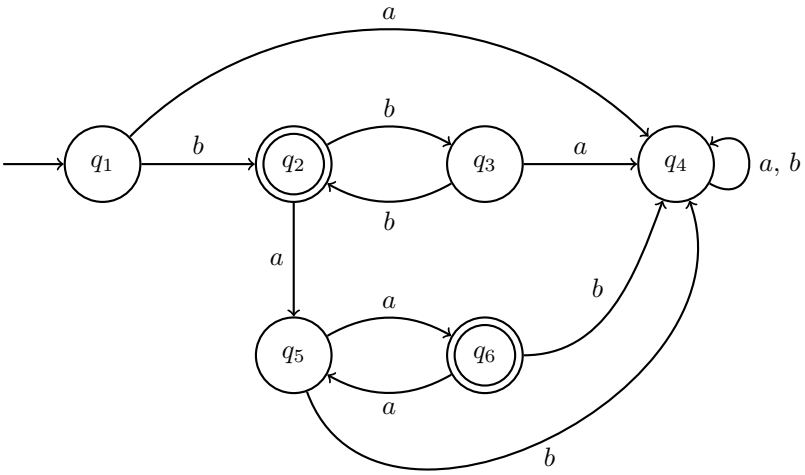
- contengono un numero pari di $a$
- contengono un numero dispari di $b$
- non contengono la sottostringa $ab$

Scrivere una *regexp* e un DFA che riconoscono tale linguaggio.

Dal momento che le stringhe di $D$ non possono contenere la sottostringa $ab$, il carattere $b$ deve necessariamente venire prima di qualsiasi occorrenza del carattere $a$. Per porre la condizione sul numero di $b$ e di $a$ che le stringhe devono avere, formuliamo la seguente *regexp*:

$$b(bb)^*(aa)^*$$

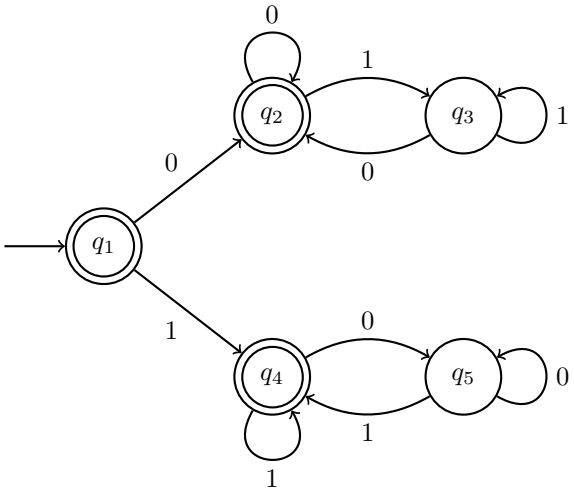
La $b$ iniziale indica che la stringa deve necessariamente iniziare per tale carattere, mentre l'operazione di star sulla sottostringa $(bb)$ permette di avere zero o più occorrenze di tale sottostringa, risultando quindi in un numero dispari di $b$. Per garantire invece la parità del numero di $a$, si applica lo stesso ragionamento senza però anteporre una $a$ iniziale. Tale *regexp* riconosce tutte le stringhe appartenenti a $D$.
Di seguito viene riportato un relativo NFA.




Lo stato $q_4$ funge da pozzo, dove viene indirizzata la computazione qualora una $a$ venga letta prima di una $b$. Gli stati accettanti sono $q_2$, che indica la lettura di un numero dispari di $b$, e $q_6$, che indica la lettura di un numero pari di $b$ dopo una lettura di un numero dispari di $b$. La computazione può terminare e l'automa può riconoscere la stringa anche qualora non vengano lette $a$, dal momento che 0 è considerato pari.

**Esercizio 2** Sia $D$ l'insieme delle stringhe che contengono lo stesso numero di occorrenze delle sottostringhe 01 e 10. Dimostrare che $D$ è regolare.

Per dimostrare che $D$ è regolare, progettiamo un DFA che riconosce tale linguaggio.




Il DFA è corretto, in quanto accetta le stringhe qualora l'input sia costituito dalla stringa vuota, da uno 0 o da un 1 (rispettivamente, gli stati $q_1$, $q_2$ e $q_4$). Il DFA si dirama sulla base del primo carattere letto, e in base a ciò conta le occorrenze delle due sottostringhe alternandosi tra $q_2$ e $q_4$ se il primo carattere letto è 0, tra $q_3$ e $q_5$ altrimenti.

**Esercizio 3** Dimostrare che la classe dei linguaggi regolari è chiusa rispetto all'operazione complemento.

Sia $A$ un linguaggio regolare. Vogliamo dimostrare che $\bar{A} = \{w \mid w \notin A\}$ è regolare.

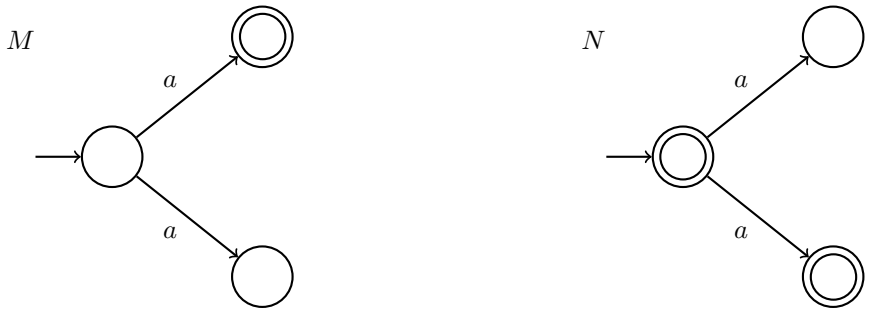
Abbiamo a nostra disposizione tre strumenti per dimostrare la regolarità di un linguaggio: le *regexp*, i DFA e gli NFA. In questo caso, ci conviene escludere le *regexp*, in quanto non ci offrono un operatore per rappresentare il complemento di un insieme.

Poiché $A$ è regolare, allora esiste un DFA $M$ tale che $L(M) = A$. Sia $M = (Q, \Sigma, \delta, q_0, F)$: vogliamo costruire un nuovo DFA $N$ tale che $L(N) = \bar{A}$. Questo nuovo DFA sarà, per forza di cose, definito in questo modo: $N = (Q, \Sigma, \delta, q_0, Q \backslash F)$, cioè avrà stesso insieme degli stati, stesso alfabeto, stessa funzione di transizione e stesso stato iniziale di $M$, tuttavia gli stati accettanti di $N$ sono tutti quelli stati di $M$ che non sono accettanti.

Fondamentalmente, si può dimostrare la seguente doppia implicazione:

$$\forall\, w,\ w \in L(M) \Leftrightarrow w \notin L(N)$$

Per risolvere questo tipo di esercizi bisogna stare attenti ad utilizzare gli strumenti giusti: se avessimo usato un NFA invece di un DFA, la dimostrazione potrebbe non risultare corretta, in quanto si potrebbe verificare una situazione simile:




In questo caso, $a \in L(M) \cap L(N)$, per cui utilizzare un NFA risulta scorretto.

**Esercizio 4** Dimostrare che $A = \{a^{2^n} \mid n \geq 0\}$ non è regolare.

Utilizziamo il *pumping lemma*. Assumiamo per assurdo che $A$ sia regolare; allora, deve esistere una *pumping length* $p$ tale che ogni stringa di $A$ con lunghezza almeno $p$ può essere divisa nella concatenazione di tre sottistringhe $xyz$ dove:

- $\forall\, i,\ xy^iz \in A\ (i \geq 0)$
- $|y| > 0$
- $|xy| \leq p$

Considero la stringa $a^{2^p} = xyz$. Tale stringa è un candidato plausibile, perché è una stringa appartenente al linguaggio in quanto costituita dalla concatenazione di $2^p$ occorrenze del carattere $a$ ed è più lunga di $p$.
Ragioniamo sulla lunghezza di $xy^2z$:

$$2^p < |xy^2z| \leq 2^p + p < 2^p + 2^p = 2^{p+1}$$

Vale che $|xy^2z| > 2^p$ perché $|y| > 0$, e che $|xy^2z| \leq 2^p + p$ perché $|xy \leq p$ e quindi $|y| \leq p$. Inoltre, $2^p + p$ è sicuramente minore di $2^p + 2^p = 2 \cdot (2^p) = 2^{p+1}$, e dal momento che $|xy^2 + z|$ è strettamente compresa tra due potenze di due consecutive, $xy^2z$ non può fare parte di $A$. Di conseguenza, $A$ non è un linguaggio regolare.

**Esercizio 5** Dimostrare che $G = \{ab^nc^n \mid n \geq 0\}$ non è regolare.

Assumiamo per assurdo che $G$ sia regolare e sia $p$ la sua *pumping length*. Sia $s = ab^pc^p$ una stringa appartenente a $G$: è un candidato ragionevole perché appartiene a $G$ ed è lunga almeno $p$, per essere precisi è lunga $2p + 1$. Dividiamola in tre sottostringhe $xyz$ che rispettino le condizioni esposte nell'esercizio precedente.

Poiché $|xy| \leq p$, il *pumping* verrà eseguito sicuramente nella prima metà della stringa; distinguiamo due casi:

- Se $y$ contiene il carattere $a$, allora $xy^2z \notin G$ perché essa contiene più di un'occorrenza di $a$.
- Se $y$ non contiene il carattere $a$, allora essa è composta solo da $b$. Ma allora $xy^2z \notin G$, perché essa contiene più occorrenze del carattere $b$ che $c$: avrà la forma $ab^{p+k}c^p$ con $k > 0$, per cui non appartiene al linguaggio.

**Esercizio 6** Utilizzare il risultato appena dimostrato nel punto precedente per dimostrare la non regolarità di $F = \{a^i b^j c^k \mid i, j, k \geq 0 \wedge (i = 1 \Rightarrow j = k)\}$.

Stringhe come *bbccc*, *aabbccc* e *abc* fanno parte di $F$, mentre stringhe come *bac*, *ab* e *abcc* non fanno parte di $F$: il linguaggio comprende tutte le stringhe costituite dai caratteri $a$, $b$ e $c$ tali che, se è presente una sola occorrenza del carattere $a$, allora le occorrenze dei caratteri $b$ e $c$ devono essere dello stesso numero.

Assumiamo per assurdo che $F$ sia regolare. Considero il linguaggio $F \cap ab^*c^*$: $F$, per ipotesi, è regolare, mentre $ab^*c^*$ è regolare perché è espresso tramite una *regexp*. Poiché sia $F$ che $ab^*c^*$ sono regolari, allora anche la loro intersezione è regolare. La loro intersezione, tuttavia, è proprio il linguaggio $G$ che abbiamo dimostrato non essere regolare nell'esercizio precedente, in quanto comprende tutte le stringhe costituite da una sola occorrenza del carattere $a$ e dallo stesso numero di occorrenze dei caratteri $b$ e $c$. Di conseguenza non è regolare, e siccome $ab^*c^*$ lo è perché espresso tramite una *regexp*, allora per forza $F$ non è regolare.

Per questo linguaggio non si poteva dimostrare la non regolarità tramite il *pumping lemma*: esistono linguaggi non regolari che ne rispettano tutte le condizioni, e che "pompandone" le stringhe i risultati appartengono al linguaggio.

**Esercizio 7** Sia $B$ un linguaggio e sia $B^+ = \{w_1 \dots w_n \mid n \geq 1 \wedge \forall i,\ w_i \in B\}$. Dimostrare che $B = B^+$ se e solo se $BB \subseteq B$. Ricordiamo che $BB = \{w_1 w_2 \mid w_1 \in B \wedge w_1 \in B\}$.

$\Rightarrow$) Sia $B = B^+$, dimostriamo che $BB \subseteq B$. Per definizione, $BB \subseteq B^+$: per formare una stringa appartenente a $BB$ concateniamo due stringhe appartenenti a $B$, mentre per formare una stringa appartenente a $B^+$ concateniamo un numero arbitrario di stringhe appartenenti a $B$. Di conseguenza, siccome per ipotesi $B^+ = B$, abbiamo che $BB \subseteq B$.

$\Leftarrow$) Sia $BB \subseteq B$, dimostriamo che $B = B^+$. Poiché sicuramente $B \subseteq B^+$, mi basta dimostrare che anche $B^+ \subseteq B$. Ci aspettiamo che $B^+$ sia contenuto in $B$ perché, prese due stringhe in $B$, concatenandole rimaniamo in $B$.
Considero una stringa $w \in B^+$: per definizione ha forma $w_1 \dots w_n$ con $n \geq 1 \wedge \forall i,\ w_i \in B$. Procediamo per induzione su $n$:

- Se $n = 1$, $w = w_1 \in B$.
- Se $n > 1$, $w = w_1 \dots w_{n-1}, w_n$; osserviamo che la sottostringa $w_1 \dots w_{n-1}$ è un elemento di $B$ per ipotesi induttiva, ma anche $w_n$ appartiene a $B$ perché per definizione di $B^+$, tutte le $w_i$ che ne compongono le stringhe solo elementi di $B$. Quindi, anche $w \in B$ perché $BB \subseteq B$.

# 4 Linguaggi Context-Free

## 4.1 Grammatiche Context-Free (CFG)

Le **grammatiche context-free** (abbreviate con la sigla **CFG**) sono degli insiemi di regole, e ciascuna di queste viene definita **produzione**. Tali produzioni hanno una forma specifica: associato ad un simbolo presente nella parte sinistra della regola (**simboli non-terminali** o **variabili**, rappresentati tendenzialmente con una lettera maiuscola dell'alfabeto) un insieme di simboli nella parte destra (stringhe di **simboli terminali** e **non-terminali**).
Il seguente è un esempio di grammatica context-free:

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

In questo esempio, 0, 1 e # sono simboli terminali, mentre $A$ e $B$ sono non-terminali. Ogni produzione, a sinistra, ha esattamente un terminale, mentre a destra ha una sequenza di caratteri terminali e/o non-terminali.
Il terminale presente a sinistra della prima produzione prende il nome di ***start symbol***.

Le grammatiche operano mediante un processo di riscrittura non arbitrario, costituito da tre passi fondamentali:

- Parto dallo *start symbol*
- Riscrivo un non-terminale come la parte destra di una produzione
- Continuo fino ad avere una stringa di soli terminali

La sequenza di riscritture e sostituzioni di non-terminali con terminali prende il nome di **derivazione**. La stringa 000#111 appartiene alla grammatica appena presentata, e il suo processo di derivazione è il seguente:

$$A \Rightarrow \underbrace{0A1}_{\text{Prima regola}} \Rightarrow \underbrace{00A11}_{\text{Prima regola}} \Rightarrow \underbrace{000A111}_{\text{Prima regola}} \Rightarrow \underbrace{000B111}_{\text{Seconda regola}} \Rightarrow \underbrace{000\#111}_{\text{Terza regola}}$$

Una derivazione può essere visualizzata graficamente tramite il concetto di ***parse tree***:

Quando abbiamo introdotto il concetto di *pumping lemma*, l'abbiamo usato per dimostrare la non regolarità del linguaggio $\{0^n 1^n \mid n \geq 0\}$. Tale linguaggio è facilmente descrivibile dalla seguente grammatica:

$$A \to 0A1$$
$$A \to \varepsilon$$

La derivazione che permette di generare la stringa 0011 è la seguente:

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 0011$$

Procediamo a dare qualche definizione formale.

**Definizione di CFG**

Una **grammatica context-free** (**CFG**) è una quadrupla $(V, \Sigma, R, S)$ dove:
- $V$ è un insieme finito di non-terminali
- $\Sigma$ è un insieme finito di terminali ($V \cap \Sigma = \emptyset$)
- $R$ è un insieme finito di regole della forma $A \to w$ dove $A \in V$ e $w \in (V \cup \Sigma)^*$
- $S \in V$ è detto *start symbol*

**Definizione di Produzione**

Siano $u, v, w \in (V \cup \Sigma)^*$ e sia $A \to w$ una produzione. Allora, diciamo che $uAv$ **produce** $uwv$ e lo indichiamo con $uAv \Rightarrow uwv$.

**Definizione di Derivazione**

Diciamo che $u$ **deriva** $v$ e lo indichiamo con $u \Rightarrow^* v$ se e solo se $u = v$ oppure esistono delle stringhe $u_1 \dots u_k$ tali che $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$.

**Definizione di Linguaggio di una CFG**

Sia $G = (V, \Sigma, R, S)$. Definiamo il **linguaggio** di $G$ come $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$.

Proviamo a riscrivere la grammatica introdotta a inizio sezione tramite una quadrupla avente le proprietà date dalla definizione.

- $V = \{A, B\}$
- $\Sigma = \{0, 1, \#\}$
- $R = \{A \to 0A1,\ A \to B,\ B \to \#\}$
- $S = A$

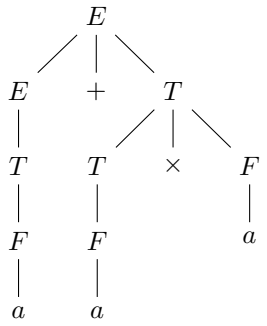
**Esempio** Vediamo un altro esempio di grammatica: $G = S \to aSb \,|\, SS \,|\, \varepsilon$. Il simbolo dato dalla barra verticolare è uno *shortcut* per esprimere più produzioni date dallo stesso simbolo non-terminale.
Elenchiamo alcune stringhe e cerchiamo di capire se appartengono al linguaggio di $G$, riportandone il processo di derivazione in caso affermativo:

- $ab \in L(G) : S \Rightarrow aSb \Rightarrow ab$
- $aabb \in L(G) : S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$
- $abb \notin L(G)$; tendenzialmente, dimostrare la non appartenenza di una stringa ad un linguaggio di una grammatica è più complicato di dimostrarne l'appartenenza; in questo caso, si può dimostrare per induzione che il numero di $a$ e di $b$ nelle stringhe di questo linguaggio è lo stesso.
- $bbaa \notin L(G)$; il numero di $a$ e $b$ non è l'unico aspetto coperto da una proprietà del linguaggio.
- $abbaab \notin L(G)$;
- $abab \in L(G) : S \Rightarrow aSb \Rightarrow aSbaSb \Rightarrow a\varepsilon ba\varepsilon b \Rightarrow abab$

**Esempio** Prendiamo ora quest'altra grammatica:

$$\begin{aligned} E &\to E + T \,|\, T \\ T &\to T \times F \,|\, F \\ F &\to (E) \,|\, a \end{aligned}$$

Tale grammatica ha tre simboli non-terminali ($E$, $T$ e $F$), cinque simboli terminali ((, ), $+$, $\times$ e $a$) e sei produzioni.
Vediamo come questa grammatica ci permette di generare la stringa $a + a \times a$:




La frontiera del *parse tree* generato consiste nella stringa che volevamo costruire. Vediamo ora lo stesso procedimento tramite derivazione:

$$\begin{aligned} E &\Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \\ &\Rightarrow a + T \Rightarrow a + T \times F \Rightarrow a + F \times F \\ &\Rightarrow a + a \times F \Rightarrow a + a \times a \end{aligned}$$

**Esempio** Cerchiamo una grammatica per il seguente linguaggio.

$$\{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$$

Per la parte a sinistra, abbiamo già visto che possiamo usare la seguente grammatica: $S \to 0S1 \mid \varepsilon$; in modo analogo, per la parte destra possiamo usare la seguente grammatica: $T \to 1T0 \mid \varepsilon$.
Siamo in grado di trovare una grammatica che ne riconosca il linguaggio? Dal momento che, se una stringa appartiene al linguaggio riportato sopra, deve appartenere o alla parte sinistra o alla parte destra (o potenzialmente a entrambe), di conseguenza possiamo modellare la seguente grammatica:

$$V \to S \mid T$$

dove $V$ è lo *start symbol* e i simboli $S$ e $T$ sono i simboli relativi alle grammatiche riportate sopra.

In generale, possiamo dire che la classe dei linguaggi context-free è chiusa rispetto all'operazione di unione.

**Teorema**

| Esiste un algoritmo che dato in input un DFA $M$ mi produce una CFG $G$ tale che $L(G) = L(M)$. |
|---|

Ciò implica che i linguaggi regolari sono un sottinsieme stretto dei linguaggi context-free.

L'algoritmo procede in questo modo:

1. Per ogni stato $q_i$ del DFA, genera un non-terminale $R_i$.
2. Se $\delta(q_i, a) = q_j$, allora genero la produzione $R_i \to aR_j$.
3. Se $q_k$ è uno stato accettante, genero la produzione $R_k \to \varepsilon$.

### 4.1.1 Ambiguità

Immaginiamo di avere la seguente grammatica.

$$EXPR \to EXPR + EXPR \mid EXPR \times EXPR \mid (EXPR) \mid a$$

Come la grammatica vista precedentemente, essa descrive formule matematiche. Vogliamo derivare la stringa $a + a \times a$, ma nel farlo ci accorgiamo che esistono due possibili *parse tree* che permettono di derivarla:

I due *parse tree* sembrano equivalenti, tuttavia quello a destra è quello più formalmente corretto perché rappresenta meglio l'aspetto della precedenza dell'operatore $\times$ rispetto all'operatore $+$.

Diciamo che una grammatica $G$ è **ambigua** se e solo se esiste una stringa $w \in L(G)$ tale che $w$ ha due *parse tree* differenti.

Il libro di testo propone anche la seguente definizione: una grammatica $G$ è ambigua se e solo se esiste una stringa $w \in L(G)$ tale che $w$ ha due derivazioni a sinistra differenti. Con derivazione a sinistra intendiamo una derivazione in cui, ad ogni passo, noi andiamo a riscrivere sempre il non-terminale più a sinistra. Vediamo le due derivazioni a sinistra per questo esempio.

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow a + E \Rightarrow a + E \times E \\ &\Rightarrow a + a \times E \Rightarrow a + a \times a \end{aligned}$$

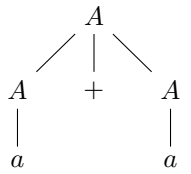
$$\begin{aligned} E &\Rightarrow E \times E \Rightarrow E + E \times E \Rightarrow a + E \times E \\ &\Rightarrow a + a \times E \Rightarrow a + a \times a \end{aligned}$$

Il problema dell'ambiguità si limita alle derivazioni a sinistra. Supponiamo di voler utilizzare la seguente grammatica per derivare la stringa $a + a$:

$$A \to A + A \mid a$$

$$\begin{aligned} A &\Rightarrow A + A \Rightarrow a + A \Rightarrow a + a \\ A &\Rightarrow A + A \Rightarrow A + a \Rightarrow a + a \end{aligned}$$

Da un punto di vista formale, queste sono due derivazioni diverse, ma in realtà questa grammatica non è ambigua perché il *parse tree* per generare $a + a$ è sempre uno solo:




**Esempio** Trovare una grammatica per il seguente linguaggio: $\{a^i b^j c^k \mid i = j \vee j = k\}$. Tale linguaggio è detto **inerentemente ambiguo**, perché si può dimostrare che non esistono grammatiche che lo descrivano che non siano ambigue.

Possiamo riscrivere questo linguaggio come l'unione di due linguaggi.

$$\{a^n b^n c^k \mid n, k \geq 0\} \cup \{a^k b^n c^n \mid n, k \geq 0\}$$

Procediamo a definire una grammatica per ciascuno dei due linguaggi e successivamente un'ulteiore grammatica che ne costituisca l'unione.

$$\begin{aligned} S_1 &\to AC & S_2 &\to BD \\ A &\to aAb \mid \varepsilon & B &\to aB \mid \varepsilon \\ C &\to cC \mid \varepsilon & D &\to bDc \mid \varepsilon \end{aligned}$$

Il risultato finale sarà dunque il seguente:

$$S \to S_1 \mid S_2$$

### 4.1.2 Forma normale di Chomsky

**Definizione**

Una CFG è in **forma normale di Chomsky** se e solo se ogni produzione è in una delle seguenti forme:

- $A \to BC$ dove $B, C \neq S$
- $A \to a$
- $S \to \varepsilon$

**Teorema**

Ogni CFG può essere riscritta in forma normale di Chomsky.

**Dimostrazione** Diamo un algoritmo di trasformazione.

1. Genero un nuovo *start symbol* $S_0$ e inserisci una produzione $S_0 \to S$.
   Fondamentalmente, questo passo non cambia il funzionamento della grammatica, tuttavia è necessario affinché venga rispettata la prima condizione della forma normale di Chomsky, secondo la quale lo *start symbol* non può stare a destra di nessuna produzione.

2. Elimino le regole della forma $A \to \varepsilon$ con $A \neq S_0$.
   Chiaramente, limitandoci ad eliminare le regole esistenti in questa forma finiremmo per cambiare il linguaggio della grammatica; supponiamo di avere le seguenti produzioni:

   $$A \to \varepsilon$$
   $$R \to uAvAw$$

   Dopo la conversione, ci dovremmo trovare con le seguenti produzioni:

   $$R \to uvAw$$
   $$R \to uAvw$$
   $$R \to uvw$$

   Nel generare le nuove regole, abbiamo delineato tre diverse casistiche in cui, a partire dalla sequenza di caratteri a destra della prima produzione, viene sostituita con la stringa vuota o la prima $A$, o la seconda o entrambe.
   Se abbiamo due produzioni del tipo $A \to \varepsilon$ e $R \to A$, eliminiamo la prima introducendo $R \to \varepsilon$, e ripetiamo tutti i passaggi appena compiuti finché non riusciamo ad eliminare tutte le $\varepsilon$-produzioni.

3. Elimino le regole nella forma $A \to B$, dette regole unitarie.
   La forma normale di Chomsky, infatti, ci dice che per ogni produzione comprendente caratteri non-terminali, essi devono esistere esattamente nel numero di due, e nessuna di queste può essere lo *start symbol*.
   Se la nostra grammatica comprende le regole $A \to B$ e $B \to u$, esse vanno sostituite con la nuova regola $A \to u$. Anche in questo caso, come nel punto precedente, è possibile che l'eliminazione di regole unitarie provochi la creazione di altre: anche qui, dunque, è necessario riapplicare l'algoritmo in modo che converga e ritorni come risultato finale una grammatica nella forma normale di Chomsky.

4. Sostituisco tutte le regole della forma $A \to u_1 u_2 \dots u_k$, con $k \geq 3$, nel seguente modo:

$$\begin{aligned} &A \to u_1 A_1 \\ &A_1 \to k_2 A_2 \\ &\dots \\ &A_{k-2} \to u_{k-1} u_k \end{aligned}$$

5. Rimpiazzo ogni $u_i$ che è terminale con un $U_i$ non-terminale ed inserisco $U_i \to u_i$.

**Esempio** Sia data la seguente grammatica.

$$\begin{aligned} &S \to ASA \,|\, aB \\ &A \to B \,|\, S \\ &B \to b \,|\, \varepsilon \end{aligned}$$

Si converta la grammatica appena data nella forma normale di Chomsky.

$$\begin{aligned} &S \to AA_1 \mid UB \mid a \mid SA \mid AS \\ &A \to b \mid AA_1 \mid UB \mid a \mid SA \mid AS \\ &B \to b \\ &S_0 \to AA_1 \mid UB \mid a \mid SA \mid AS \\ &A_1 \to SA \\ &U \to a \end{aligned}$$

## 4.2 Automi a Pila

Gli **automi a pila** (o **PDA**, ***Pushdown Automaton***) stanno ai linguaggi context-free come gli NFA ai linguaggi regolari: costituiscono in una "macchinetta" che accetta le stringhe che fanno parte di un linguaggio context-free. PDA e CFG sono ugualmente espressivi, allo stesso modo di come DFA, NFA e *regexp* sono ugualmente espressivi. La dimostrazione verrà fornita più avanti.

I PDA sono fondamentalmente costituiti da un NFA e da uno *stack* infinito associato ad esso, ovvero una struttura dati caratterizzata dalla politica LIFO e dotata quindi delle operazioni *push* e *pop*. Lo *stack* funge da memoria del PDA: se negli NFA che abbiamo studiato fino ad ora l'unica memoria è caratterizzata dai loro stati, ora i PDA possono attingere ad una quantità di memoria infinita data appunto dallo *stack*.

Abbiamo già visto che il linguaggio $\{0^n 1^n \mid n \geq 0\}$ non è regolare, e che le stringhe che ne fanno parte possono essere derivate da una CFG. Tale linguaggio viene riconosciuto da un PDA: data per esempio la stringa 000111, tale PDA carica sullo *stack* tutti gli 0 che trova; quando poi comincia a parsare gli 1, cambia stato e rimuove gli zeri dallo *stack* per ogni 1 che riceve in input.

**Definizione**

Un PDA è una sestupla $(Q, \Sigma, \Gamma, \delta, q_0, F)$ dove:
- $Q$ è un insieme finito di stati
- $\Sigma$ è un alfabeto finito
- $\Gamma$ è un insieme finito di simboli per lo *stack*
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathscr{P}(Q \times \Gamma)$ è la funzione di transizione
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati accettanti

dove $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ e $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$.

Anche qui, come nel caso degli NFA, dobbiamo considerare anche le $\varepsilon$-transizioni, che possono avvenire sia per mano dell'input sia per mano dello *stack*.

**Definizione**

Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ un PDA. Allora, $M$ accetta la stringa $w$ se e solo se $w$ si può scrivere come $w_1 \dots w_m$ $(\forall\, i,\ w_i \in \Sigma_\varepsilon)$ ed esistono sequenze di stati $r_0, r_1, \dots, r_m \in Q$ e di stringhe $s_0, s_1, \dots, s_m \in \Gamma^*$ tali che:
- $r_0 = q_0$ e $s_0 = \varepsilon$
- $r_m \in F$; l'ultimo stato deve essere accettante, ma non ci sono vincoli sullo stato dello *stack* alla fine della computazione
- Per ogni $i = 0, \dots, m-1$ abbiamo $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$ con $s_i = at$ e $s_{i+1} = bt$ per qualche $a, b \in \Gamma_\varepsilon$ e $t \in \Gamma^*$

Distinguiamo quattro casi di computazione sulla base dei valori di $a$ e $b$:

- Se $a \neq \varepsilon$ e $b \neq \varepsilon$, la computazione prevede una *pop* e una *push*.
- Se $a = \varepsilon$ e $b \neq \varepsilon$, l'operazione effettuata è una *push*.
- Se $a \neq \varepsilon$ e $b = \varepsilon$, l'operazione effettuata è una *pop*.
- Se $a = b = \varepsilon$, allora $s_i = s_{i+1} = t$, cioè lo *stack* non viene toccato.

**Esempio** Arriviamo finalmente ad un esempio concreto. Vogliamo progettare un PDA che riconosca il linguaggio $\{0^n 1^n \mid n \geq 0\}$.
Il meccanismo dell'automa è semplice: partendo dallo stato iniziale, attraverso una $\varepsilon$-mossa ci spostiamo automaticamente verso lo stato $q_2$, effettuando un'operazione di *push* sullo *stack* di un carattere speciale, che denotiamo con \$, e che ci funge da *flag* per riconoscere che lo *stack* è vuoto. L'operazione di *push* viene effettuata sull'arco che indica la transizione: la formula $\varepsilon \to \$$ si può tradurre con "Fai spontaneamente una *push* di \$".
Entriamo quindi in un *loop* dello stato $q_2$, su cui rimaniamo fintanto che viene soddisfatta la condizione presente sul suo cappio: "Se leggo uno 0 in input, faccio spontaneamente una *push* di 0". Quando leggiamo il primo 1 cambiamo stato: "Se leggo un 1 in input e in cima allo *stack* ho uno 0, faccio la *pop*". Allo stesso modo, rimaniamo in $q_3$ fintanto che è valida la condizione del suo cappio: "Se leggo un 1 in input e ho uno 0 in cima allo *stack*, faccio una *pop*".
Infine, le condizioni sulla transizioni da $q_3$ a $q_4$ indicano di cambiare spontaneamente stato verso $q_4$ qualora legga il simbolo \$ inserito all'inizio della computazione. Gli stati accettanti sono, appunto, $q_4$, che indica una lettura di $n$ 0 e $n$ 1, ma anche $q_1$, perché la stringa vuota fa parte del linguaggio.

**Esempio** Sia dato il linguaggio $\{a^i b^j c^k \mid i, j, k \geq 0 \wedge (i = j \vee i = k)\}$. Tale linguaggio comprende tutte le stringhe formate dalla concatenazione di zero o più caratteri $a$, $b$ e $c$, in cui vi sono le stesse occorrenze dei caratteri $a$ e $b$ o dei caratteri $a$ e $c$. Determiniamo un automa a pila non deterministico che riconosca tale linguaggio.

Il meccanismo di tale PDA è sostanzialmente il medesimo del precedente: un carattere speciale \$ ci rappresenta lo *stack* vuoto. Dopo aver raggiunto $q_2$, ci rimaniamo fintanto che riceviamo delle $a$, e le aggiungiamo allo *stack*. Con una biforcazione identifichiamo i due casi possibili da riconoscere: quello in cui il numero di $b$ è pari al numero di $a$ o quello in cui il numero di $c$ è pari al numero di $a$. Attraverso una serie di controlli incrociati sul carattere ricevuto in input e sul carattere in cima allo *stack*, effettuiamo delle operazioni su di esso e delle $\varepsilon$-transizioni verso uno dei due stati accettanti, nei quali la computazione termina con esito positivo se, al loro raggiungimento, non ci sono altri caratteri da parsare (diversi da $c$ se siamo in $q_4$), con esito negativo altrimenti.

## 4.3 Equivalenza di PDA e CFG

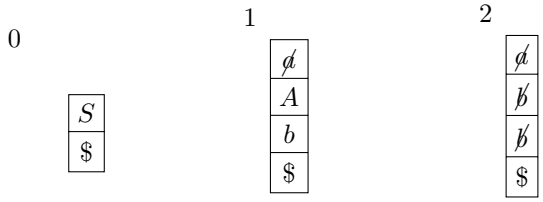
**Teorema**

| Un linguaggio $A$ è *context-free* se e solo se esiste un PDA $P$ tale che $L(P) = A$. |
|---|

**Dimostrazione** Dal momento che anche tale teorema è costituito da una doppia implicazione, la sua dimostrazione avviene dimostrandole entrambe.

**Lemma** Se $A$ è *context-free*, allora esiste un PDA $P$ tale che $L(P) = A$.

**Dimostrazione** Immaginiamo di avere una derivazione di questo tipo: $S \Rightarrow aAb \Rightarrow aabb$. Vogliamo cercare di imitare questa derivazione tramite lo *stack* di un PDA. Se la generazione di questa stringa dovesse avvenire tramite un PDA, la seguente sarebbe la sua esecuzione.




Tramite un abuso di notazione, quando scriviamo $\varepsilon \to S\$$ indichiamo di voler inserire più di un carattere all'interno dello *stack*; in questa notazione, il carattere più a sinistra è quello che si troverà più in alto sullo *stack*.
Al primo passo della computazione, inseriamo il carattere speciale \$ e lo *start symbol*. Ad ogni passo, il nostro obiettivo è quello di "far esplodere" il primo carattere dello *stack* se esso è un carattere non-terminale, cioè sostituirlo nello *stack* con la stringa $w$ per ogni regola della forma $A \to w$, oppure di riconoscerlo se è lo stesso carattere ricevuto in input, e in tal caso ne viene effettuata una *pop*. Il *loop* continua fintanto che esistono regole per ogni carattere non-terminale sullo *stack* e finché non raggiungiamo il simbolo \$. A questo punto vuol dire che potremo accettare la stringa passata in input.

**Lemma** Se un linguaggio è riconosciuto da un PDA $P$, allora esso è *context-free*.

**Dimostrazione** La dimostrazione di questo lemma è più laboriosa di quella del lemma precedente, e ciò si deve, tra le altre cose, al fatto che la sintassi dei PDA è molto più vasta e complicata di quella delle grammatiche, allo stesso modo di come la grammatica degli NFA è più ampia di quella delle *regexp* (ci siamo trovati in un contesto simile anche nella dimostrazione della loro equivalenza). Per questo motivo, definiamo l'idea di come procedere per trovare un algoritmo di conversione:

1. Disciplinare il PDA: non vogliamo lavorare con PDA arbitrari, ma vogliamo porre su di essi delle assunzioni senza perdita di generalità (qualunque PDA noi vogliamo convertire, esso può essere disciplinato).
2. Generare una CFG con non-terminali indicati come $A_{pq}$ dove $p$, $q$ sono stati del PDA: per ogni coppia di stati, vogliamo definire un nuovo non-terminale; ciò porterà necessariamente ad una grammatica ampia, ma a noi non interessa l'efficienza.
3. $A_{pq} \Rightarrow^* x$ se e solo se $P$ riconosce $x$ partendo da uno *stack* vuoto ed arrivando in uno *stack* vuoto.
4. Definire $A_{q_0,\, q_{accept}}$ come *start symbol* della CFG: $q_0$ è lo stato iniziale del PDA, e $q_{accept}$ è lo stato accettante.

Analizziamo un punto alla volta.

**1) Disciplinare il PDA** Devono valere le seguenti proprietà.

- Il PDA da convertire deve avere un unico stato accettante chiamato $q_{accept}$.
  Questa proprietà può essere garantita creando un nuovo stato, rendendolo accettante e accessibile dagli altri stati accettanti tramite delle $\varepsilon$-transizioni e rendendo questi ultimi non più accettanti. La presenza di un solo stato accettante ci aiuta a definire lo *start symbol* senza il rischio di incorrere in ambiguità.
- La pila deve essere vuota prima di accettare la stringa.
  Per garantire questa proprietà è sufficiente creare uno stato intermedio che funga da *buffer* prima di spostarci verso lo stato accettante. Tale condizione ci aiuta a garantire la correttezza del terzo punto dell'algoritmo.
- Ciascuna transizione è una *push* oppure una *pop*.
  Tale condizione può essere violata da due tipi di transizioni: quelle della forma $\varepsilon \to \varepsilon$, che non fanno operazioni sullo *stack*, e quelle della forma $a \to b$, che eseguono contemporaneamente una *pop* e una *push* atomiche. Le ultime vanno "disaccopiate" tramite l'inserimento di uno stato intermedio, e trasformate in una *pop* seguita da una *push*, cioè trasformate nella forma $a \to \varepsilon$ (*pop*) e $\varepsilon \to b$ (*push*); le prime, analogamente, vanno convertite in una *push* e una *pop* di uno stesso simbolo *dummy*.

Tali proprietà non hanno perdita di generalità: ogni PDA può essere convertito in un PDA equivalente che le soddisfa.

**2) Generare la CFG** Possiamo generare tre tipi di produzioni:

1. Se $(r, u) \in \delta(p, a, \varepsilon)$ e $(q, \varepsilon) \in \delta(s, b, u)$, allora genero la produzione $A_{pq} \to aA_{rs}b$.
2. Per ogni $p$, $q$, $r$, genero la produzione $A_{pq} \to A_{pr}A_{rq}$.
3. Per ogni $p$, viene generata la produzione $A_{pp} \to \varepsilon$.

Le prime due possibilità descrivono le modalità di computazione del PDA:

1. Partiamo dallo *stack* vuoto e terminiamo nello *stack* vuoto, senza avere mai svuotato lo *stack*.

2. Partiamo dallo *stack* vuoto e terminiamo nello *stack* vuoto, e nel mezzo abbiamo svuotato lo *stack* almeno una volta.

**3) Condizione critica**

**Lemma** $A_{pq} \Rightarrow^* x$ se e solo se $P$ parte da $p$ con *stack* vuoto ed arriva in $q$ con *stack* vuoto processando $x$.

**Parte 1** Se $x$ porta $P$ da $p$ con *stack* vuoto a $q$ con *stack* vuoto, allora $A_{pq} \Rightarrow^* x$.

**Dimostrazione** Avviene per induzione sul numerodi passi della computazione del PDA $P$.

- **Caso base**: 0 passi di computazione. In questo caso $p$ con *stack* vuoto da in $p$ con *stack* vuoto processando $x = \varepsilon$. Osserviamo che $A_{pp} \Rightarrow \varepsilon$ tramite la produzione $A_{pp} \to \varepsilon$.
- **Passo induttivo**: assumo che la proprietà valga per computazioni di al più $k$ passi e la dimostro per $k+1$ passi. Distinguo due casi:
  - La pila è vuota solo all'inizio ed alla fine. Vengono eseguiti i seguenti passi:
    1. All'inizio della computazione, $P$ parte da $p$ con *stack* vuoto, fa una *push* di $u$ e passa nello stato $r$. Assumiamo che il carattere processato sia $a$.
    2. Successivamente, $P$ parte da $r$ con $u$ sullo *stack* ed arriva a $s$ con $u$ sullo *stack*. Assumiamo che la stringa processata sia $y$.
    3. Infine, $P$ parte da $s$ con $u$ sullo *stack*, fa la *pop* di $u$ e passa nello stato $q$. Assumiamo che il carattere processato sia $b$.

    Dal momento che vogliamo dimostrare l'invarianza della proprietà su una computazione di $k+1$ passi, sapendo che i punti 1. e 3. della computazione sono costituiti da un solo passo, ne deriviamo che il punto 2. è costituito da $k-1$ passi; tuttavia, non possiamo chiamare l'ipotesi induttiva su questo punto, perché la proprietà ci dice che all'inizio e alla fine della computazione lo *stack* è vuoto, ma qui non lo è. In realtà, l'ipotesi induttiva ci dice che se $P$ parte da $r$ con *stack* vuoto, esso arriva a $s$ con *stack* vuoto in $k-1$ passi: a noi poco importa quale sia il contenuto dello *stack* (il carattere $u$), in quanto la computazione eseguita dal punto 2. rispetta l'ipotesi induttiva: $A_{rs} \Rightarrow^* y$.
    Se l'input è la stringa $x = ayb$, dobbiamo poter definire da dove vengono generate le sue sottostringhe. Abbiamo appena dimostrato come può essere derivata $y$, mentre $a$ e $b$ sono derivabili per definizione della CFG:

    $$A_{pq} \Rightarrow aA_{rs}b \Rightarrow^* ayb = x$$

  - La pila si svuota in uno step intermedio. Assumiamo che lo *stack* si svuoti in $r$. Allora, $p$ computa come segue:
    1. $P$ parte da $p$ con *stack* vuoto e va in $r$ con *stack* vuoto leggendo $y$.
    2. $P$ parte da $r$ con *stack* vuoto e va in $q$ con *stack* vuoto leggendo $z$.

    Dal momento che la computazione è costituida da $k+1$ passi, ciascuno dei due punti sarà costituito da al più $k$ passi: per ipotesi induttiva, $A_{pr} \Rightarrow^* y$ e $A_{eq} \Rightarrow^* z$. Otteniamo quindi, per definizione della CFG, la seguente derivazione:

    $$A_{pq} \Rightarrow A_{pr}A_{rq} \Rightarrow^* yz = x$$

**Parte 2** Se $A_{pq} \Rightarrow^* x$, allora $x$ porta $P$ da $p$ con lo *stack* vuoto a $q$ con lo *stack* vuoto.

**Dimostrazione** Anche in questo caso, la dimostrazione avviene per induzione sul numero di passi della derivazione di $x$ da $A_{pq}$.

- **Caso base**: 1 passo di computazione. In questo caso, la derivazione deve usare una regola che non contenga simboli variabili nella sua parte destra. Le uniche regole in $G$ che non contengono variabili nella parte destra sono del tipo $A_{pp} \to \varepsilon$. L'input $\varepsilon$ porta $P$ da $p$ con *stack* vuoto a $p$ con *stack* vuoto, quindi il caso base è dimostrato.
- **Passo induttivo**: assumiamo che l'enunciato sia vero per le derivazioni di lunghezza al più $k$, con $k \geq 1$, e proviamo che esso è vero per derivazioni di lunghezza $k+1$.
  Supponiamo che $A_{pq} \Rightarrow^* x$ in $k+1$ passi. Il primo passo in questa derivazione è $A_{pq} = aA_{rs}b$ oppure $A_{pq} \Rightarrow A_{pr}A_{rq}$, per definizione della CFG. Trattiamo questi due casi separatamente.
  - Nel primo caso, consideriamo la parte $y$ di $x$ che $A_{rs}$ genera, quindi $x = ayb$. Poiché $A_{rs} \Rightarrow^* y$ in $k$ passi, l'ipotesi induttiva ci dice che $P$ può andare da $r$ con la pila vuota a $s$ con la pila vuota. Poiché $A_{pq} \to aA_{rs}b$ è una regola di $G$, $\delta(p, a, \varepsilon)$ contiene $(r, u)$ e $\delta(s, b, u)$ contiene $(q, \varepsilon)$, per qualche simbolo di pila $u$. Quindi, se $P$ inizia in $p$ con la pila vuota, dopo aver letto $a$ può andare nello stato $q$ ed eliminare $u$ dalla prila. Pertando, $x$ può portare $P$ da $p$ con lo *stack* vuoto a $q$ con lo *stack* vuoto.
  - Nel secondo caso, consideriamo le parti $y$ e $z$ di $x$ che $A_{pr}$ e $A_{rq}$ rispettivamente generano, quindi $x = yz$. Poiché $A_{pr} \Rightarrow^* y$ in al più $k$ passi e $A_{rq} \Rightarrow^* z$ in al più $k$ passi, l'ipotesi induttiva ci dice che $y$ può portare $P$ da $p$ a $r$ e $z$ può portare $P$ da $r$ a $q$ con lo *stack* vuoto all'inizio e alla fine della computazione. Quindi $x$ può portare $P$ da $p$ con lo *stack* vuoto a $q$ con lo *stack* vuoto. Questo completa la prova del passo induttivo.

**Corollario**

| Tutti i linguaggi regolari sono anche *context-free*. |
|---|

**Dimostrazione** Ogni linguaggio regolare è riconosciuto da un automa a stati finiti, e ogni automa a stati finiti è un automa a pila che semplicemente ignora la sua pila. Di conseguenza gli automi a pila riconoscono anche i linguaggi regolari.

## 4.4 Il *pumping lemma* per linguaggi *context-free*

In modo analogo a quanto abbiamo visto per il *pumping lemma* nello studio dei linguaggi regolari, ne esiste una variante che è molto utile per dimostrare che un linguaggio $B$ non è *context-free*.

**Definizione**

Se $A$ è un linguaggio *context-free*, allora esiste un numero $p$, detto *pumping length*, tale che se $s \in A$ e $s$ ha lunghezza almeno $p$ ($|s| \geq p$), allora $s$ si può scrivere come $uvxyz$ dove:

1. $\forall\, i \geq 0,\ uv^i xy^i z \in A$
2. $|vy| > 0$
3. $|vxy| \leq p$

**Idea della dimostrazione**

Sia $A$ un linguaggio *context-free*. Consideriamo $s \in A$ "molto lunga", allora essa deve avere un *parse tree* "molto alto". Poiché questo *parse tree* è "molto alto", esso deve contenere una variabile $R$ che si ripete.

Prendendo questo *parse tree*, possiamo fare due tipi di operazioni della categoria *"cuci-e-taglia"*: *pumping up* e *pumping down*.

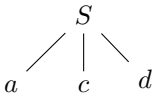
Operazione di *pumping up*. Operazione di *pumping down*.

**Dimostrazione**

Sia $A$ un CFL e sia $G$ la sua grammatica tale che $L(G) = A$. Sia $b$ il massimo numero di simboli che occorrono nella parte destra di una produzione. Assumiamo $b \geq 2$: se così non fosse, la grammatica non potrebbe generare un numero infinito di stringhe, ma ogni variabile verrebbe sostituita al più con un singolo carattere terminale o con la stringa vuota, situazione che renderebbe il teorema trivialmente vero.
Esemplifichiamo la questione. Data la seguente grammatica,

$$S \to acd \,|\, aS,$$

vale che $b = 3$, perché il massimo numero di caratteri che stanno a destra di una produzione della grammatica è 3. In questo caso, la lunghezza della stringa prodotta da un *parse tree* di altezza 1 è proprio pari a $b = 3$:




In generale, possiamo dire che se ho un *parse tree* di altezza $h$, esso genera una stringa di al più $b^h$ caratteri. Questo significa anche che se ho una stringa di $b^h + 1$ caratteri, tutti i suoi *parse tree* devono avere altezza pari ad almeno *h+1*.

Sia $|V|$ il numero di variabili di $G$. Definisco la *pumping length* $p = b^{|V|+1}$. Dimostriamo che valgono le tre condizioni citate nella definizione del *pumping length*.
Sia $s \in A$ una stringa di lunghezza almeno $p = b^{|V|+1}$. Osserviamo prima di tutto che:

$$b^{|V|+1} \geq b^{|V|} + 1 \quad (\text{per } b \geq 2)$$

Per il fatto enunciato, tutti i *parse tree* di $s$ devono avere altezza almeno $|V| + 1$.

Consideriamo $\tau$ il *parse tree* di $s$ con il minimo numero di nodi. Consideriamo ora il suo cammino più lungo: tale cammino deve avere lunghezza almeno $|V| + 1$. Esso deve comprendere almeno $|V| + 2$ nodi, di cui $|V| + 1$ sono nodi interni (variabili) e una foglia (terminale). Di conseguenza, esiste una variabile $R$ che si ripete lungo questo cammino. Assumo che $R$ si ripeta fra le $|V| + 1$ variabili più in basso del mio cammino.

Procediamo a dimostrare le tre condizioni.

1. Dividiamo $s$ in $uvxyz$ come abbiamo visto nel disegno.
2. $|vy| > 0$
   Assumiamo per assurdo che $|vy| = 0$. Allora $s = uxz$, ma questo è assurdo perché $\tau$ ha il minor numero di nodi ed il *parse tree* $\tau'$ ottenuto tramite l'operazione di *pumping down* sarebbe un *parse tree* per $s = uxz$ con meno nodi di $\tau$.
3. $|vxy| \leq p$: $vxy$ è generato dal sotto-*parse tree* $\tau''$ più grande radicato in $R$. Noi sappiamo che $R$ occorre fra le $|V| + 1$ variabili più in basso nel cammino più lungo di $\tau$. L'altezza di $\tau''$ è minore o uguale a $|V| + 1$. Di conseguenza, $|vxy| \leq b^{|V|+1} = p$.

### Applicazione del *pumping lemma*

**Esempio B.** Vediamo un esempio di utilizzo del *pumping lemma* per dimostrare che il linguaggio $B = \{0^n 1^n 2^n \mid n \geq 0\}$ non è *context-free*.

Supponiamo per assurdo che $B$ sia un linguaggio *context-free*. Allora esiste una *pumping length* $p$ tale che se prendo $s \in B$ con $|s| \geq p$, $s$ può essere divisa in $uvxyz$ dove:

1. $\forall\, i \geq 0,\ uv^i xy^i z \in B$
2. $|vy| > 0$
3. $|vxy| \leq p$

Considero $s = 0^p 1^p 2^p$. Ragioniamo per casi a partire dall'osservazione che $|vy| > 0$:

- Almeno una fra $v$ e $y$ contiene più di un tipo di carattere.

  $$\underbrace{0\,0\ldots}_{u}\underbrace{0\,1}_{v}\underbrace{1\ldots1}_{x}\underbrace{2\,2\ldots}_{y}\underbrace{2}_{z}$$

  Facendo un'operazione di *pumping up*, avremo dei caratteri in ordine sbagliato, per cui $uv^2xy^2z \notin B$:

  $$0\,0\ldots\underbrace{0\,1\,0\,1}_{v^2}1\,1\ldots$$

- Sia $v$ che $y$ contengono esattamente un solo tipo di carattere. Allora, esiste un carattere fra 0, 1 e 2 che non occorre in $v$ o in $y$. Assumiamo che il carattere non raprpesentato sia il 2: allora, $uv^2xy^2z \notin B$ perché essa comprende meno 2 rispetto agli 0 oppure agli 1.

**Esempio C.** Dimostriamo che $C = \{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}$ non è *context-free*. Assumiamo che $C$ sia *context-free* e sia $p$ la sua *pumping length*. Scegliamo $s = a^p b^p c^p = uvxyz$, e distinguiamo diversi casi:

- Almeno una fra $v$ e $y$ contiene più di un tipo di simbolo:

  $$\underbrace{a\ldots}_{u}\underbrace{a\,b}_{v}\underbrace{\ldots b}_{x}\underbrace{c\ldots}_{y}\underbrace{c}_{z}$$

  In tal caso, un'operazione di *pumping up* porterebbe ad un mescolarsi dei caratteri che viola la proprietà descritta dal linguaggio:

  $$a\ldots\underbrace{a\,b\,a\,b}_{v^2}\ldots b\,c\ldots c \notin C$$

- Sia $v$ che $y$ comprendono un solo tipo di simbolo. In questo caso, uno fra $a$, $b$ e $c$ non occorre né in $v$ né in $y$. Delineiamo ulteriori sottocasi:
  - $a$ non occorre né in $v$ né in $y$; allora, la stringa $uxz$, cioè il risultato di un'operazione di *pumping down* su $s$, non appartiene a $C$, in quanto avrei meno occorrenze di $b$ o di $c$ rispetto alle $a$.
  - $c$ non occorre né in $v$ né in $y$; allora, la stringa $uv^2xy^2z$, cioè il risultato di un'operazione di *pumping up* su $s$, non appartiene a $C$, in quanto ci sarebbero più occorrenze di $a$ o di $b$ rispetto alle $c$.

- $b$ non occorre né in $v$ né in $y$; in questo caso, la situazione è più delicata e sorge la necessità di studiare altri due casi:
  - $a$ occorre in $v$ o in $y$: dal momento che $b$ non occorre né in $v$ né in $y$, facendo un'operazione di *pumping up* otterremmo una stringa $uv^2xy^2z$ che avrà più occorrenze del carattere $a$ (che abbiamo incrementato) rispetto alle $b$ (che sono rimaste invariate).
  - $c$ occorre in $v$ o in $y$: dal momento che $b$ non occorre né in $v$ né in $y$, facendo un'operazione di *pumping down* otterremo una stringa $uxz$ che avrà più occorrenze del carattere $b$ (che è rimasto invariato) rispetto alle $c$ (che abbiamo decrementato).

## 4.5 CFL e operazioni regolari

**Teorema**

La classe dei linguaggi *context-free* è chiusa rispetto all'operazione di unione $\cup$.

**Dimostrazione**

Siano $A$ e $B$ due linguaggi *context-free*. Vogliamo dimostrare che $A \cup B$ è un linguaggio *context-free*.

Poiché $A$ e $B$ sono CFL, esistono due grammatiche $G, H$ tali che $L(G) = A$ e $L(H) = B$. Siano $G = (V_1, \Sigma_1, R_1, S_1)$ e $H = (V_2, \Sigma_2, R_2, S_2)$. Costruisco $I = (V_3, \Sigma_3, R_3, S_3)$ tale che $L(I) = A \cup B$, dove:

- $V_3 = V_1 \cup V_2 \cup \{S_3\}$ $(S_3 \notin V_1 \cup v_2)$
- $\Sigma_3 = \Sigma_1 \cup \Sigma_2$
- $R_3 = R_1 \cup R_2 \cup \{S_3 \to S_1,\ S_3 \to S_2\}$
- $S_3$ è lo *start symbol* appena aggiunto

Tutto questo assumendo che $V_1 \cap V_2 = \emptyset$, in modo che non vi sia perdita di generalità; qualora queste due grammatiche avessero variabili identificate con lo stesso nome, vanno rinominate, magari aggiungendo un pedice che ci permette di capire a quale linguaggio appartengono originariamente i non-terminali.

**Esercizio** Dimostrare la chiusura dei linguaggi *context-free* rispetto alle operazioni di concatenazione e star.

**Concatenazione - $\circ$**

Siano $A$ e $B$ due linguaggi *context-free*. Vogliamo dimostrare che $A \circ B$ è un linguaggio *context-free*.

Poiché $A$ e $B$ solo CFL, esistono due grammatiche $G, H$ tali che $L(G) = A$ e $L(H) = B$. Siano $G = (V_1, \Sigma_1, R_1, S_1)$ e $H = (V_2, \Sigma_2, R_2, S_2)$. Costruisco $I = (V_3, \Sigma_3, R_3, S_3)$ tale che $L(I) = A \circ B$, dove:

- $V_1 \cup V_2 \cup \{S_3\}$ $(S_3 \notin S_1 \cup S_2)$
- $\Sigma_3 = \Sigma_1 \cup \Sigma_2$
- $R_3 = R_1 \cup R_2 \cup \{S_3 \to S_1 S_2\}$
- $S_3$ è lo *start symbol* appena aggiunto

Va fatta, anche in questa dimostrazione, la medesima considerazione effettuata nella dimostrazione precedente per quanto riguarda la ridenominazione delle variabili di $G$ e $H$ aggiungendovi un relativo pedice, in modo da trovarci nella situazione ideale nella quale $V_1 \cap V_2 = \emptyset$, che ci permette di procedere senza perdita di generalità.

**Star - $*$**

Sia $A$ un linguaggio *context-free*. Vogliamo dimostrare che $A^*$ è un linguaggio *context-free*.

Poiché $A$ è una CFL, esiste una grammatica $G$ tale che $L(G) = A$. Sia $G = (V_1, \Sigma_1, R_1, S_1)$. Costruisco $H = (V_2, \Sigma_2, R_2, S_2)$ tale che $L(H) = A^*$, dove:

- $V_2 = V_1 \cup \{S_2\}$
- $\Sigma_2 = \Sigma_1$
- $R_2 = R_1 \cup \{S_2 \to S_1 S_2 \,|\, \varepsilon\}$
- $S_2$ è lo *start symbol* appena aggiunto

**Esercizio** Dimostrare che la classe dei linguaggi *context-free* non è chiusa rispetto all'operazione di intersezione $\cap$.

Consideriamo i seguenti linguaggi: $A = \{a^m b^m c^n \,|\, m, n \geq 0\}$ e $B = \{a^m b^n c^n \,|\, m, n \geq 0\}$.
I due linguaggi sono *context-free*, e per dimostrarlo progettiamo le seguenti grammatiche che li riconoscono:

$$\underbrace{\begin{aligned} S &\to AC \\ A &\to aAb \,|\, \varepsilon \\ C &\to cC \,|\, \varepsilon \end{aligned}}_{L(G)=A} \qquad \underbrace{\begin{aligned} S &\to AB \\ A &\to aA \,|\, \varepsilon \\ B &\to bBc \,|\, \varepsilon \end{aligned}}_{L(H)=B}$$

Il linguaggio costituito dalla loro intersezione è il seguente: $A \cap B = \{a^n b^n c^n \,|\, n \geq 0\}$. Tale linguaggio non è *context-free*, e l'abbiamo dimostrato con l'utilizzo del *pumping lemma*.

**Teorema**

| La classe dei CFL non è chiusa rispetto all'operazione di complemento. |
|---|

**Dimostrazione**

Assumiamo per assurdo la chiusura rispetto all'operazione di complemento. Siano $C, D$ due linguaggi *context-free*. Per ipotesi, $\bar{C}$ e $\bar{D}$ sono a loro volta linguaggi *context-free*. Per il risultato mostrato prima, possiamo dire che anche $\bar{C} \cup \bar{D}$ è un linguaggio *context-free*. Possiamo ulteriormente dire che, per ipotesi, $\overline{\bar{C} \cup \bar{D}}$ è a sua volta un linguaggio *context-free*. Per le leggi di De Morgan, possiamo altresì dire che $\bar{\bar{C}} \cap \bar{\bar{D}}$ è un linguaggio *context-free*. Tuttavia, questo è un assurdo, perché $\bar{\bar{C}} \cap \bar{\bar{D}} = C \cap D$, e noi abbiamo appena dimostrato che la classe dei linguaggi *context-free* non è chiusa rispetto all'operazione di intersezione. Siamo quindi giunti ad un assurdo e possiamo dire che la classe dei linguaggi non è chiusa nemmeno rispetto all'operazione di complemento.

**Esercizio** Dato $\Sigma = \{0, 1\}$, dimostrare che sono CFL:

- L'insieme delle stringhe che contengono almeno tre 1.

$$S \to A1A1A1A$$
$$A \to 0A \,|\, 1A \,|\, \varepsilon$$

- L'insieme delle stringhe che iniziano e finiscono con lo stesso simbolo.

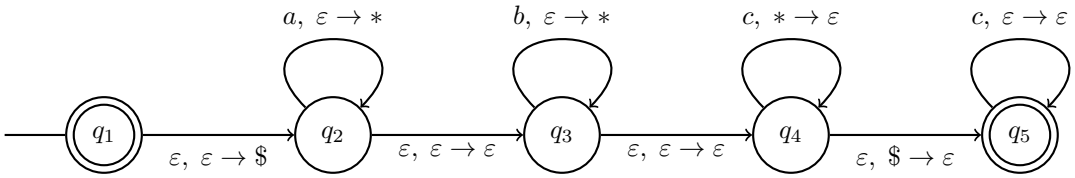
$$S \to 0A0 \,|\, 1A1 \,|\, 0 \,|\, 1$$
$$A \to 0A \,|\, 1A \,|\, \varepsilon$$

- L'insieme delle stringhe di lunghezza dispari.

$$S \to 0A \,|\, 1A$$
$$A \to 00A \,|\, 01A \,|\, 10A \,|\, 11A \,|\, \varepsilon$$

**Esercizio** Consideriamo il linguaggio $\{a^i b^j c^k \,|\, i, j, k \geq 0 \wedge i + j \leq k\}$.

Costruire un PDA per tale linguaggio.




Fornire ora una CFG per tale linguaggio.

$$S \to AC$$
$$C \to cC \,|\, \varepsilon$$
$$A \to aAc \,|\, B$$
$$B \to bBc \,|\, \varepsilon$$

**Esercizio** Dimostrare che il linguaggio $B = \{0^n \# 0^{2n} \# 0^{3n} \,|\, n \geq 0\}$ non è *context-free*.

Assumiamo per assurdo che $B$ sia *context-free*, e sia $p$ la sua *pumping length*. Consideriamo $s = 0^p \# 0^{2p} \# 0^{3p}$: essa è un buon candidato, perché appartiene a $B$ ed è lunga almeno $p$, precisamente $6p+2$. Poniamo $s = uvxyz$, osserviamo che $|vy| > 0$ per la seconda condizione del *pumping lemma* e ragioniamo due casi:

1. $v$ oppure $y$ contiene il carattere $\#$. Allora, compiendo un'operazione di *pumping up* otteniamo $uv^2xy^2z$, che contiene più di due simboli $\#$. Quindi $uv^2xy^2z \notin B$.
2. Sia $v$ che $y$ contengono solo il carattere 0. Osserviamo che $|vxy| \leq p$: $vxy$ può costituire la prima sequenza di zeri oppure può essere compresa nella seconda o nella terza sequenza di zeri. In ogni caso, almeno uno dei tre segmenti non è mai toccato da $vxy$, in qualunque modo noi effettuiamo la divisione. Allora, effettuare un'operazione di *pumping up* viola la proporzione 1:2:3 che sussiste tra i segmenti, per cui $uv^2xy^2z \notin B$.

# 5 Lexing e Parsing

La teoria dei linguaggi formali ha importanti applicazioni pratiche nel mondo dei compilatori e dei linguaggi di programmazione: i compilatori, per definire la sintassi di un linguaggio, implementano un meccanismo di riconoscimento dell'input basandosi su una combinazione di CFG ed espressioni regolari.

$$\begin{aligned}
&\text{Decl} \rightarrow \text{Keyword Id} = \text{Expr}\\
&\text{Expr} \rightarrow \text{Expr} + \text{Atom} \mid \text{Term}\\
&\text{Term} \rightarrow \text{Term} * \text{Atom} \mid \text{Atom}\\
&\text{Atom} \rightarrow (\text{Expr}) \mid \text{Num} \mid \text{Id}
\end{aligned}$$

$$\begin{aligned}
&\text{Keyword} \rightarrow \text{int} \mid \text{float}\\
&\text{Num} \rightarrow [0-9]+ \mid [0-9]+.[0-9]+\\
&\text{Id} \rightarrow [\text{a}-\text{z}]([\text{a}-\text{z}] \mid [0-9])*
\end{aligned}$$

Distinguiamo tre tipi di errori.

- **Errori lessicali**: identificabili tramite una *regexp*.

```
int 12c = 5 + 3;
```

- **Errori sintattici:** identificabili tramite una CFG.

```
int x = 5 + (3;
```

- **Errori semantici**: al di là delle possibilità di *regexp* e CFG.

```
int x = 5 + 3.4;
```

## 5.1 Lexing

Il **lexing** è il processo di conversione da una sequenza di caratteri ad una seguenza di **token** con un significato stabilito. Un token normalmente consiste in una coppia che associa ciascun **lessema** del programma al *tipo* di lessema, ad esempio keyword (KW), identificatore (ID), operatore (OP) *et cetera*.
Un compilatore che riceve in input la stringa "`int x = 12;`", in seguito al processo di lexing genererà in output una stringa simile alla seguente:

```
[(KW, int), (ID, x), (OP, =), (NUM, 12), (SEP, semi)]
```

La componente del compilatore che si occupa del lexing si deve occupare della risoluzione di due problemi:

- **Descrizione e riconoscimento**: risolvibile per messo di espressioni regolari e DFA.
- **Estrazione dei token**: è un problema apparentemente facile, ma bisogna fare attenzione alle ambiguità; supponiamo di dover estrarre dei token dalle seguenti stringhe:
  - `myvar12`: bisogna estrarre un token `[(ID, myvar12)]` o due token `[(ID, myvar), (NUM, 12)]`?
  - `int`: consiste in una keyword `[(KW, int)]` o in un identificatore `[(ID, int)]`?

Le soluzioni più comuni per tali problematiche prevedono l'implementazione di un meccanismo di priorità e l'uso di un algoritmo di tipo *longest math rule*.

### 5.1.1 Errori lessicali e sintattici

Vediamo come un lexer gestisce gli errori.

- – Input: `int 12c = 5 + 3;`
  – Output: errore lessicale rilevato dal lexer, perché `12c` non è generato da alcuna *regexp*.
- – Input: `int x = 5 + (3;`
  – Output: `[(KW, int), (ID, x), (OP, =), (NUM, 5), (OP, +), (SEP, Ipar), (NUM, 3), (SEP, semi)]`. Non viene rilevato alcun errore lessicale: l'errore è sintattico e verrà rilevato dal parser.

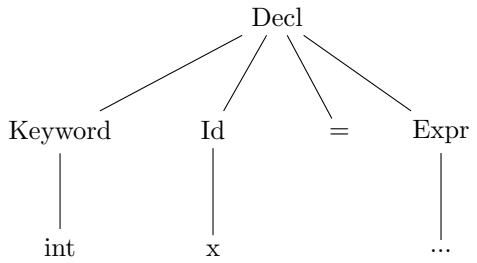
### 5.1.2 Generatori di lexer

Un generatore di lexer come Lex opera come segue:

1. Data una serie di *regexp* ed **azioni** ad esse associate, costruisce un NFA equivalente all'unione delle *regexp*.
2. Il NFA viene convertito in un DFA equivalente e minimizzato.
3. Il DFA è convertito in un estrattore di token, tenendo conto della *longest match rule*, delle prorità delle *regexp* ed altre sottigliezze.
4. Quando un lessema è riconosciuto da una delle *regexp* originali, il lexer esegue l'azione corrispondente (per esempio, ritornare il token).

## 5.2 Parsing

Il **parsing** è il processo di conversione da una sequenza di token in un **albero di sintassi astratta**, che è "quasi" un *parse tree*.
Dato, per esempio, l'input `[(KW, int), (ID, x), (OP, =), (NUM, 12), (SEP, semi)]`, il parser ritornerà il seguente albero:



### 5.2.1 Algoritmi di parsing

Il processo di parsing è molto più difficile del processo di lexing. Avviene tramite una delle seguenti metodologie:

- **Parsing top-down**: costruisce l'AST discendendo dalla radice (*start symbol*) verso le foglie (token); è un processo semplice ed intuitivo, tuttavia purtroppo inefficiente.
- **Parsing bottom-up**: costruisce l'AST risalendo dalle foglie (token) verso la radice (*start symbol*); è un processo molto più complicato, ma di gran lunga più efficiente nella pratica.

Nonostante queste due tecniche, esistono delle CFG che non possono essere gestite dagli algoritmi di parsing.

**Esempio - Bottom-up Parsing: Shift-Reduce**

### 5.2.2 Generatori di parser

Un generatore di parser bottom-up come Yacc opera come segue:

1. Prende in input una CFG le cui produzioni sono annotate con delle **azioni** e costruisce una tabella di parsing corrispondente.
2. La tabella di parsing guida la scelta di shift / reduce sulla base dell'input e dello stato interno del parser.
3. Quando viene effettuata una reduce, il parser esegue l'azione della produzione corrispondente (per esempio, costruisce un nodo dell'AST).
4. Se alla fine dell'input è rimasto solo lo *start symbol* sullo *stack*, allora il parsing è andato a buon fine.

# 6 Calcolabilità

Nello studio dei linguaggi formali, abbiamo conosciuto diverse classi di linguaggi: i linguaggi regolari, sui quali operano delle macchinette deterministiche che abbiamo definito DFA, e i linguaggi *context-free*, sui quali operano delle macchinette non deterministiche dotate di memoria infinita sotto forma di *stack*, i PDA.
Entrambe queste macchine sono, in un senso lasco, dei calcolatori, ma si differenziano dai computer con cui abbiamo a che fare ogni giorno. Tali macchine non possono riconoscere un linguaggio come $\{a^n b^n c^n \mid n \geq 0\}$, ma un calcolatore moderno è invece in grado di comperci operazioni sopra. In questa parte del corso, andremo a studiare un nuovo modello computazionale, che vedremo essere potente quanto un computer moderno, in grado appunto di riconoscere linguaggi come quello appena riportato. Questo modello prende il nome di **Macchina di Turing**, in onore dell'informatico britannico Alan Turing che negli anni '30 formalizzò il concetto di computazione.

## 6.1 Macchine di Turing

Ad alto livello, una Macchina di Turing non è così diversa da un automa che abbiamo visto fin'ora: possiamo vederlo come un sistema dotato di una testina posizionata su una cella di un nastro di lunghezza infinita, e di uno stato interno.

Come gli automi, anche la Macchina di Turing riceve un input, ma invece di leggere un carattere alla volta lo riceve in maniera continua dal nastro; preso un istante qualsiasi, la macchina sposta la testina sul nastro, lo aggiorna e cambia stato in base all'input letto dalla cella su cui è posizionata la testina. Ogni operazione può portare alla modifica dei dati sul nastro e dello stato interno, finché la macchina non entra in uno stato di *accept* o di *reject*.

A differenza degli automi, la Macchina di Turing ha una memoria infinita ed utilizzabile in modo arbitrario. Possiamo anche vedere questa macchina come un'astrazione dei computer moderni, in cui la memoria è, appunto, finita, per cui l'assenza di un termine del nastro serve per astrarre il concetto di memoria.
Inoltre, gli stati di *accept* e di *reject* hanno effetto immediato: negli automi, l'accettazione o il rifiuto dell'input sono determinati dalla fine del flusso di caratteri che lo costituiscono; siccome qui non esiste il concetto di serializzabilità dell'input, nonappena lo stato assunto dalla macchina è di *accept* o di *reject* la computazione termina con l'esito relativo allo stato finale.

Passiamo ad un esempio concreto: prendiamo il linguaggio $B = \{w\#w \mid w \in \{0, 1\}^*\}$; esso descrive la concatenazione di due stringhe binarie uguali, separate da un delimitatore (il simbolo #). Tale linguaggio non è *context-free*, eppure è riconoscibile dalla Macchina di Turing. Vediamo in che modo è possibile accettare, per esempio, la stringa 011000#011000.

La computazione avviene nel seguente modo: la macchina legge il primo carattere sul nastro, cambia il suo stato interno per memorizzare l'informazione del carattere che ha appena letto e lo "cancella", cioè pone al suo posto sul nastro un carattere arbitrario, supponiamo una `x`; successivamente, scorre il nastro finché non trova il simbolo delimitatore # (se non lo trova, la macchina entra in uno stato di *reject* e la computazione termina), e legge il carattere immediatamente alla sua destra: se è diverso dal carattere appena letto, entra in uno stato di *reject* e la computazione termina, altrimenti prosegue.
Se la computazione prosegue, viene eliminato allo stesso modo il carattere appena letto presente a destra del delimitatore. Il nastro viene riavvolto in modo che la testina si posizioni sulla `x` che ha inserito prima, e legge il carattere immediatamente alla sua destra; nel farlo, cambia stato sulla base del carattere letto e lo cancella, per poi scorrere di nuovo il nastro a destra finché non trova il primo carattere diverso da `x`. Viene effettuato nuovamente il confronto e la computazione prosegue in questo modo fino alla fine dell'input.
Quando sono stati letti tutti i simboli a sinistra del delimitatore, la macchina controlla che non vi siano altri caratteri non letti alla sua destra: se ce ne sono, lo stato assunto diventa quello di *reject*, altrimenti quello di *accept*.

Per completezza, forniamo il meccanismo della macchina come descritto dal libro di testo. Sia $M_1$ la Macchina di Turing.
Allora, $M_1$ = Su input $w$:

1. Muoviti a zig-zag lungo il nastro, raggiungendo posizioni corrispondenti sui due lati del #. Se le due posizioni contengono caratteri diversi, rifiuta. Se non trovi #, rifiuta. Cancella con una `x` gli elementi corrispondenti altrimenti.
2. Quando tutti gli elementi a sinistra di # sono `x`, verifica la presenza di caratteri diversi da `x` dopo il #. Se non ce ne sono accetta, altrimenti rifiuta.

Entrambe le spiegazioni sul funzionamento della macchina tuttavia sono parziali, omettono dettagli di funzionamento, motivo per cui è necessario dare una definizione formale di Macchina di Turing.

**Definizione formale di Macchina di Turing**

Una **Macchina di Turing** o **MdT** è una settupla $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ dove:
- $Q$ è un insieme finito di stati
- $\Sigma$ è l'alfabeto finito di input ($\sqcup \notin \Sigma$)
- $\Gamma$ è l'alfabeto finito del nastro ($\sqcup \in \Gamma \ \wedge \ \Sigma \subseteq \Gamma$)
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
- $q_0 \in Q$ è lo stato iniziale
- $q_{accept} \in Q$ è lo stato accettante
- $q_{reject} \in Q$ è lo stato rifiutante ($q_{accept} \neq q_{reject}$)

La Macchina di Turing è un calcolatore **deterministico**, e lo si capisce dal fatto che a destra della funzione di transizione non abbiamo un insieme di possibilità, bensì una possiiblità sola.
$L$ e $R$ sono due bit che ci indicano uno spostamento a destra o a sinistra della testina rispetto alla posizione dell'ultima lettura, e il simbolo $\sqcup$, che chiamiamo *blank*, indica l'assenza di caratteri sulla cella.

La computazione di una Macchina di Turing, in generale, avviene attraverso questi step:

A. Partiamo dallo stato iniziale $q_0$, con la testina il più a sinistra possibile e con l'input $w$ sul nastro (assumiamo un numero infinito di $\sqcup$ a destra).

B. Computa secondo $\delta$.

C. Se arrivi in $q_{accept}$, accetta; se arrivi in $q_{reject}$, rifiuta. In questa situazione si potrebbe creare un potenziale *loop* nel caso in cui la Macchina di Turing si sposti sempre a destra una volta raggiunta la sequenza infinita di $\sqcup$.

**Configurazione** Per come funzionano le MdT, sappiamo che in ogni istante il nastro conterrà delle informazioni, la macchina si troverà in un certo stato e la testina sarà posizionata su una cella del nastro, come nella figura seguente.

Tutte queste informazioni possono essere rappresentate tramite una stringa che prende il nome di **configurazione**. Per esempio, la configurazione della MdT in figura, in quel preciso istante, sarà la seguente:

$$1011q_701111$$

Diciamo che una configurazione $C_1$ **produce** la configurazione $C_2$ tramite la notazione $C_1 \rightsquigarrow C_2$ per indicare il seguente comportamento. Siano $a, b, c \in \Gamma$, $u, v \in \Gamma^*$ e $q_i, q_j \in Q$. Supponiamo che una nostra ipotetica MdT abbia la configurazione data da $C_1 = uaq_ibv$: allora, data la transizione $\delta(q_i, b) = (q_j, c, L)$, la configurazione di partenza si evolve nella configurazione $C_2 = uq_jacv$ (abbiamo effettuato una computazione del carattere sotto la testina, la $b$, trasformandola in una $c$, seguita da una mossa a sinistra partendo dallo stato $q_i$ verso lo stato $q_j$).

$$\frac{\delta(q_i, b) = (q_j, c, L)}{uaq_ibv \rightsquigarrow uq_jacv}$$

Data invece $\delta(q_i, b) = (q_j, c, R)$, partendo dalla configurazione $C_1 = uaq_ibv$ arriveremo alla nuova configurazione $C_2 = uacq_jv$. Ciò definisce la semantica della Macchina di Turing.

$$\frac{\delta(q_i,\, b) = (q_j,\, c,\, R)}{uaq_ibv \rightsquigarrow uacq_jv}$$

Nel caso in cui ci troviamo nella posizione più a sinistra e vogliamo effettuare un'ulteriore mossa a sinsitra, la macchina effettuerà tutte le dovute trasformazioni di stato e del nastro senza però variare la posizione della testina.

$$\frac{\delta(q_i,\, b) = (q_j,\, c,\, L)}{q_ibv \rightsquigarrow q_jcv}$$

È formalmente necessario definire anche una regola che definisca lo spostamento a destra della testina qualora essa si trovi all'inizio del nastro.

$$\frac{\delta(q_i,\, b) = (q_j,\, c,\, R)}{q_ibv \rightsquigarrow cq_jv}$$

**Definizione formale di Computazione di una MdT**

Una MdT $M = (Q,\, \Sigma,\, \Gamma,\, \delta,\, q_0,\, q_{accept},\, q_{reject})$ accetta l'input $w$ se e solo se esiste una sequenza di configurazioni $C_1,\, C_2,\, \ldots,\, C_k$ tali che:

- $C_1 = q_0w$ è la configurazione iniziale
- $\forall\, i,\ C_i \rightsquigarrow C_{i+1}$
- $C_k = uq_{accept}v$ per qualche $u,\, v \in \Gamma^*$ è una configurazione accettante.

Indichiamo con $L(M)$ il linguaggio delle stringhe accettate dalla MdT $M$.

**Definizione - Linguaggio Turing-riconoscibile**

Un linguaggio $A$ è **Turing-riconoscibile** (o **ricorsivamente enumerabile**) se e solo se esiste una MdT $M$ tale che $L(M) = A$.

**Definizione - Decisore**

Se una MdT $M$ **termina** per ogni input $w$, essa si dice **decisore**.

**Definizione - Linguaggio decidibile**

Un linguaggio $B$ è **decidibile** (o **ricorsivo**) se e solo se esiste un decisore $M$ tale che $L(M) = B$.

## 6.2 Esempi di linguaggi decidibili

**Esempio 1.** Il linguaggio $A = \{0^{2^n} \mid n \geq 0\}$, che contiene tutte le sequenze di 0 di lunghezza pari a una potenza di 2, è decidibile. La Macchina di Turing che lo riconosce opera nel seguente modo.
$M_2$ = Su input $w$:

1. Muovi la testina da sinistra a destra cancellando gli 0 alternativamente.
2. Se hai trovato un solo 0, accetta.
3. Se hai trovato un numero dispari di 0 ($> 1$), rifiuta.
4. Riavvolgi la testina

Nel cercare di riportare una rappresentazione più formale di quella appena data della MdT che riconosce $A$, per evitare il processo macchinoso di generare una settupla prevista dalla definizione formale, è possibile rappresentare graficamente le Macchine di Turing in modo analogo a quanto abbiamo visto per DFA e NFA.

Se ci troviamo nello stato iniziale e leggiamo un carattere blank, vuol dire che abbiamo ricevuto una stringa vuota, che dobbiamo rifiutare in quanto 0 non è una potenza di 2; ci muoveremo quindi da $q_1$ a $q_{reject}$ attraverso $\sqcup \to R$ (abbiamo deciso di spostare la testina a destra per convenzione, avremmo potuto benissimo decidere di spostarla verso sinistra).
Se invece leggiamo uno 0, lo marchiamo con un simbolo canarino (useremo lo stesso simbolo blank) e spostiamo la testina a destra. Ci troviamo ora in $q_2$: se leggiamo un altro 0, lo sostituiamo con una `x` e spostiamo la testina a destra, portanto la macchina allo stato $q_3$. Gli stati $q_3$ e $q_4$ si alternano per lo scorrimento dell'input verso sinistra: se siamo in $q_3$ e leggiamo uno 0 ci limitiamo a spostare la testina a destra, mentre se ci troviamo in $q_4$ oltre a spostare la testina a destra dovremo anche cancellare lo 0 appena trovato.
Se ci troviamo in $q_4$ e leggiamo un carattere blank, vuol dire che abbiamo contato un numero dispari di 0: dobbiamo rifiutare perché la lunghezza della stringa non può costituire una potenza di 2; se invece siamo in $q_3$, vuol dire che abbiamo letto un numero pari di 0, che potenzialmente potrebbero essere una potenza di 2: dobbiamo riavvolgere la stringa a sinistra finché non troviamo il carattere blank che abbiamo posto come canarino all'inizio della stringa. Questo riavvolgimento è dato dallo stato $q_5$, in cui viene spostata la testina finché troviamo 0 o `x`. Quando invece troveremo il carattere canarino, torneremo verso $q_2$ spostando la testina a destra.
Tale stato ignora tutte le `x` che trova, scorrendo la testina a destra per ognuna (ragionamento applicato anche agli stati $q_3$ e $q_4$), e se arriva alla fine della stringa trovando solo `x` (cioè è stato letto un numero di 0 pari ad una potenza di 2), quando troverà il primo carattere blank presente alla fine della stringa porterà la macchina verso lo stato accettante.
La sequenza di configurazioni generate dalla macchina per il riconoscimento della stringa 0000 è la seguente:

**Esempio 2.** Grafichiamo una Macchina di Turing che riconosca il linguaggio presentato a inizio sezione, $\{w\#w \mid w \in \{0, 1\}^*\}$.

Il meccanismo di funzionamento della macchina rappresentata incarna l'algoritmo risolutivo descritto all'inizio della sezione.
Non è inserito nel diagramma della MdT lo stato di *reject* in quanto, essendo la Macchina di Turing deterministica, quando negli stati mancano delle transizioni è come se implicitamente queste transizioni mancanti portassero allo stato di *reject*.

**Esempio 3.** Nel libro è riportato il funzionamento della Macchina di Turing che riconosce il linguaggio $\{a^i b^j c^k \mid i \times j = k \wedge i, j, k \geq 1\}$.

## 6.3 Varianti della Macchina di Turing

### 6.3.1 Macchina di Turing senza movimento della testina

Cambiando la definizione della funzione di transizione, è possibile introdurre una variante della Macchina di Turing che, oltre ad offrire i classici movimenti verso destra e verso sinistra della testina, permette anche di non muovere la testina in seguito al parsing di un carattere dell'input.

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

La mossa $S$, che sta per *Stay*, intuitivamente permette di mantenere la testina nella posizione in cui si trovava prima del processing del carattere corrente.

Formalmente, tale variante è espressivamente equivalente alla Macchina di Turing tradizionale: una Macchina di Turing tradizionale è equivalente ad una macchina appartenente a questa variante che non fa mai uso della mossa $S$, e analogamente tale mossa si può emulare in una MdT tradizionale effettuando prima uno spostamento della testina verso destra e successivamente un suo spostamento verso sinistra (e non il contrario, perché potremmo trovarci all'inizio del nastro).

### 6.3.2 Macchina di Turing multinastro

Immaginiamo di avere una macchina con $k$ nastri, ciascuno dotato di una testina indipendente. In macchine appartenenti a questa categoria, il primo nastro è quello che contiene l'input, mentre gli altri $k-1$ nastri sono inizialmente vuoti e fungono da aree di memoria ausiliarie su cui la macchina può eseguire operazioni.
Potremmo pensare che una macchina del genere sia più espressiva di una MdT tradizionale, tuttavia non è questo il caso. L'idea che regge la dimostrazione si basa sul fatto che, siccome i nastri delle Macchine di Turing hanno lunghezza infinita, tutti i possibili $k$ nastri sono rappresentabili con un nastro solo. In particolare, è possibile concatenare i nastri di una macchina multinastro in un nastro solo, con il contenuto originario di ciascun nastro propriamente separato da un carattere deliminatore.

Nella conversione non ci dobbiamo preoccupare dello stato (anche nei sistemi multinastro la macchina si trova, in ogni istante, in uno stato solo), tuttavia, siccome anche i nastri delle macchine multinastro sono infiniti e quindi potremmo aggiungere caratteri in coda alla stringa, bisogna fare opportuna attenzione a *shiftare* tutti gli elementi del nastro unico, a partire dal termine della porzione di nastro a cui vogliamo aggiungere un carattere.
Infine, bisogna anche considerare la rappresentazione delle posizioni (potenzialmente diverse) delle testine sui $k$ nastri. Nella situazione in figura, la macchina $M$ ha la prima testina in seconda posizione, la seconda in terza posizione e la terza in prima posizione; quest'informazione viene riportata nel nastro unico tramite un simbolo, noi abbiamo scelto un pallino, per denotare le celle su cui si trovano le porzioni di nastro derivanti dalla macchina multinastro.

Alla luce di ciò, possiamo affermare che la funzione di transizione per una Macchina di Turing multinastro dotata di $k$ nastri è la seguente:

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$$

**Esempio di simulazione**

Su input $w_1 \dots w_n$:

1. Popola il nastro con:
$$\dot{w}_1 \dots w_n \# \underbrace{\dot{\sqcup} \# \dots \# \dot{\sqcup}}_{k-1} \#$$
2. Per simulare una mossa, scorri tutto il nastro e segna nel tuo stato i $k$ simboli marcati con $\dot{\bullet}$. Poi scorri di nuovo il nastro per aggiornare il suo contenuto rispettando $\delta$.
3. Se per qualche motivo ti trovi a dover scrivere su un #, *shifta* a destra il contenuto del nastro e libera una cella.

### 6.3.3 Macchina di Turing non deterministica

Per far uscire la Macchina di Turing dai limiti del determinismo, è possibile effettuare alla funzione di transizione la stessa modifica che abbiamo applicato a quella dei DFA per definire gli NFA.

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

Si può dimostrare, in modo analogo a quanto sussiste tra DFA e NFA, che Macchine di Turing deterministiche e non deterministiche hanno lo stesso potere espressivo.

**Teorema**

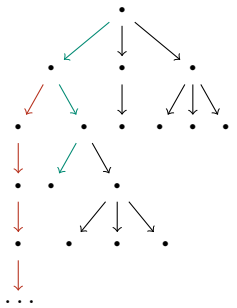
Per ogni Macchina di Turing non deterministica, esiste una Macchina di Turing deterministica equivalente.

**Idea della dimostrazione**

Il concetto di non-determinismo si basa sulla possibilità di generare diverse esecuzioni a partire da ogni passo della computazione. Una computazione non-deterministica si può quindi rappresentare come un albero, in cui ogni nodo ha come figli i passi successivi delle computazioni che genera.
Lo scopo della dimostrazione è quello di poter simulare tramite una MdT deterministica un cammino radice-foglia che rappresenta una computazione accettante; tuttavia, il meccanismo di visita su tale albero delle esecuzioni non potrà essere costituito da una visita in profondità: potremmo trovarci nella situazione in cui un'esecuzione entri in un *loop* infinito, per esempio facendo rimbalzare la testina avanti e indietro tra due celle del nastro. Occorre quindi implementare un meccanismo di ricerca in ampiezza.

La dimostrazione procede mediante la generazione di una Macchina di Turing dotata di tre nastri, attraverso i quali vengono simulate tutte le possibili esecuzioni della macchina non determinisitca. In particolare,

- Il primo nastro contiene l'input originario, e non viene mai toccato.
- Il secondo nastro è quello su cui avvengono le simulazioni delle esecuzioni: all'inizio è popolato con l'input, in maniera identica al primo nastro, ma successivamente può essere modificato in quanto vengono fatte compiute su di esso le possibili computazioni.
- Il terzo nastro è quello che risolve il non-determinismo: esso tiene traccia del cammino corrente, cioè dell'esecuzione che stiamo simulando in questo momento, opportunamente codificato tramite una stringa di interi. Tale cammino prende il nome di **indirizzo**.



Preso come esempio l'albero delle computazioni sovrastante, appare evidente che la ricerca del cammino deve essere effettuata in ampiezza: se venisse eseguita in profondità, la macchina entrerebbe nel *loop* derivante dalla computazione determinata dal cammino più a sinistra dell'albero, quella evidenziata in rosso, e non terminerebbe mai.
La macchina termina l'esecuzione quando avrà simulato la computazione relativa al cammino evidenziato in verde, che supponiamo sia accettante. Tale cammino, per esempio, viene rappresentato con la stringa 121, in quanto, a partire dall'origine, prosegue verso il primo figlio più a sinistra, poi verso il secondo figlio più a sinistra di quest'ultimo e infine verso il primo figlio più a sinistra di quest'ultimo.

Tale macchina funzionerà in questo modo:

1. Inizialmente metti sul nastro 1 l'input $w$, assumi che il nastro 2 sia vuoto e inizializza il nastro 3 a $\varepsilon$.
2. Copia il nastro 1 sul nastro 2.
3. Simola la MdT non deterministica sul nastro 2, usando il contenuto del nastro 3 per risolvere il non-determinismo. Se accetti, termina con *accept*. Se rifiuti o il nastro 3 contiene un indirizzo invalido, vai al punto 4.
4. Aggiorna il contenuto del nastro 3 con la prossima stringa e vai al punto 2.

La generazione degli indirizzi è ciò che determina la visita in ampiezza dell'albero: essi vengono generati rispettando l'ordine lessicografico, ma assicurandosi di aver generato tutte le stringhe di lunghezza $k$ prima di cominciare a generare le stringhe di lunghezza $k+1$; una possibile sequenza di generazione degli indirizzi è la seguente: $\varepsilon$, 1, 2, 3, 11, 12, 13, 21, 22, 23, ....

### 6.3.4 Enumeratore

In maniera molto informale, possiamo dire che un **enumeratore** consiste in una Macchina di Turing con una stampante annessa, che può usare come un dispositivo di output per stampare delle stringhe.

Mentre una Macchina di Turing parte con l'input su nastro ed effettua la sua computazione (che può terminare con l'*accept* o il *reject* dell'input o in un *loop*), l'enumeratore non parte con un input su nastro, ma genera stringhe su di esso, *enumerandole* appunto, con l'idea che dopo un dato lasso di tempo la stringa su nastro venga mandata alla stampante e appunto stampata. Il suo linguaggio è l'insieme delle stringhe stampate.
Possiamo vedere una Macchina di Turing come un riconoscitore di stringhe, e un enumeratore come un generatore di stringhe: può generare un numero infinito di stringhe, stampando anche più volte la stessa stringa e senza un ordine arbitrario (per definizione di insieme, non esiste un ordine e i duplicati vengono ignorati).

**Teorema**

| Un linguaggio è Turing-riconoscibile se e solo se esiste un enumeratore che lo genera. |
|---|

Conseguenza di ciò è che Macchine di Turing ed enumeratori hanno la stessa potenza espressiva.

**Dimostrazione**

**Parte 1 -** $\Leftarrow$) Assumiamo che un enumeratore $E$ tale che $L(E) = A$. Costruisco una MdT $M$ tale che $L(M) = A$, descritta in questo modo:
$M$ = Su input $w$:

1. Esegui $E$. Quando $E$ mi genera una stringa $w'$, confronta $w$ con $w'$.
2. Se $w = w'$, accetta. Altrimenti passa alla stringa successiva.

**Parte 2 -** $\Rightarrow$) Assumiamo di avere una MdT $M$ tale che $L(M) = A$. Costruisco un enumeratore $E$ tale che $L(E) = A$.
Prima di presentare un principio di funzionamento corretto per l'enumeratore, ne proponiamo uno che intuitivamente appare corretto, tuttavia non lo è.
$E$ = Per tutte le stringhe $s_1$, $s_2$, ...:

1. Esegui $M$ su $s_i$.
2. Se $M$ accetta, stampa $s_i$. Altrimenti passa alla stringa successiva.

Tale meccanismo soffre della stessa problematica che abbiamo visto nella dimostrazione dell'equivalenza tra MdT deterministiche e non: se una certa stringa causa la formazione di un *loop* all'interno di $M$, le stringhe successive non verranno mai viste. Occorre quindi modificare il meccanismo appena proposto.
$E$ = Per $i = 1, 2, 3, \ldots$:

1. Esegui $M$ per $i$ passi di computazione su $s_1$, $s_2$, ..., $s_i$.

2. Se esiste $s_j$ dove $M$ termina con *accept*, stampa $s_j$ (per tutte tali $s_j$).

La macchina funziona in questo modo:

1. Simula $M$ su $s_1$ per 1 passo.
2. Simula $M$ su $s_1$, $s_2$ per 2 passi.
3. Simula $M$ su $s_1$, $s_2$, $s_3$ per 3 passi.
4. ...

Ciò permette di non fermare la computazione di $M$ su una stringa che potenzialmente la può mandare in *loop*, ma ad ogni colpo viene dato il beneficio del dubbio alla stringa successiva per aumentare le probabilità di successo.

## 6.4 Tesi di Church-Turing

Alan Turing non fu l'unico informatico a teorizzare modelli computazionali. Ricordiamo il modello del **$\lambda$-calcolo**, che possiamo informalmente definire come un'astrazione dei linguaggi di programmazione funzionali, e il formalismo delle **funzioni ricorsive**.
È stato dimostrato che tutti i modelli sopracitati progettati per studiare la computazione sono espressivamente equivalenti. Macchine di Turing deterministiche, non deterministiche, multinastro, $\lambda$-calcolo, funzioni ricorsive *et cetera* sono tutti formalismi equivalentemente potenti per riconoscere una determinata classe di linguaggi. Ciò indusse alla formulazione della **Tesi di Church-Turing**, il cui enunciato viene riportato di seguito.

| Algoritmo = MdT = Linguaggio |
|---|

Questa tesi non è stata dimostrata, per cui non consiste in un teorema comprovato e verificato. Consiste in una congettura basata sull'osservazione dell'equivalenza espressiva di tutti i modelli computazionali ritenuti ragionevoli sopracitati, e siccome è stato osservato empiricamente che sono tutti strumenti buoni per modellare gli algoritmi, si è arrivati alla conclusione che, sebbene abbiamo tanti modi di computare, essi si riconducono alla definizione di **algoritmo**, ovverosia una sequenza di istruzioni che richiedono un tempo finito che noi eseguiamo per risolvere un problema.

Le nozioni di algoritmo e di linguaggio sono strettamente collegate, e per provarlo ci serviamo di un aneddoto storico. Nel 1900, il matematico tedesco David Hilbert propose il seguente problema: dato un polinomio $p$, dare un algoritmo che verifica se $p$ ha una radice intera. La versione originale del problema riguarda polinomi a più variabili; per semplicità, nel nostro caso di studio ci limitiamo a polinomi a singola variabile.
Per esprimere questo algoritmo attraverso un linguaggio, occorre fare alcune considerazioni.
Innanzitutto, è necessario osservare che un polinomio è un elemento dotato di una certa struttura, costituito da termini combinati con addizioni e sottrazioni; i linguaggi invece, come abbiamo visto, sono costituiti da stringhe. Quando scriviamo un programma, è a nostra discrezione scegliere in che modo rappresentare le informazioni necessarie alla risoluzione del problema. Supponiamo di voler trovare un programma che risolva un'equazione di secondo grado: potremmo voler rappresentare i coefficienti con una stringa di interi separati da virgole, e la trasformazione da istanza reale del problema a stringa viene rappresentata tramite la seguente notazione:

$$3x^2 + x + 1 \overset{<\cdot>}{\rightsquigarrow} 1, 1, 3$$

Il problema di Hilbert chiede sostanzialmente di dimostrare che esiste un decisore $M$ tale che $L(M) = \{< p > \mid p \text{ è un polinomio con radici intere}\}$

Esistono tre modi per descrivere una Macchina di Turing che risolva il problema.

- **Formale**: una descrizione formale prevede di riportare una MdT delineata secondo la sua definizione formale, cioè tramite una settupla, in cui è specificata anche la funzione di transizione $\delta$; anche un diagramma grafico consiste in una rappresentazione formale di una MdT.
- **Implementativo**: una rappresntazione di questo tipo consiste in una descrizione a parole di come operiamo sul nastro.
- **Alto livello**: tale rappresentazione sfrutta la tesi di Church-Turing per descrivere una MdT come un algoritmo.

Proviamo a fornire una descrizione ad alto livello di una macchina che risolva il problema di Hilbert, cioè che riconosca il linguaggio $A = \{< p > \mid p \text{ è un polinomio con radici intere}\}$.

$M =$ Su input $< p >$, cioè l'encoding del polinomio $p$:

1. Valuta $p$ con $x$ posto successivamente a 0, -1, +1, -2, +2, ...
2. Se trovi che il polinomio vale 0, accetta.

Tale macchina non è un decisore, perché può andare in *loop*, tuttavia dimostra che $A$ è Turing-riconoscibile. È possibile trasformare $M$ in un decisore calcolando i limiti entro i quali sono contenute le radici del polinomio, e limitare la ricerca all'interno di tale intervallo, in modo che la macchina non entri in un *loop* infinito. Questo intervallo è compreso tra i valori $\pm k\frac{c_{max}}{c_1}$, dove $k$ è il numero di termini del polinomio, $c_{max}$ è il coefficiente avente il massimo valore assoluto e $c_1$ è il coefficiente del termine di ordine più elevato.

## 6.5 Esercizi

**Esercizio 1.** Definire formalmente gli enumeratori ed il linguaggio che essi riconoscono.

Un **enumeratore** è una settupla $(Q, \Sigma, \Gamma, \delta, q_0, q_p, q_h)$ dove:
- $Q$ è un insieme finito di stati
- $\Sigma$ è un alfabeto finito di simboli per il nastro di output
- $\Gamma$ è un alfabeto finito di simboli per il nastro di lavoro
- $\delta : Q\backslash\{q_p, q_h\} \times \Gamma \to Q \times \Gamma \times \{L, R\} \times (\Sigma \cup \{\varepsilon\})$ è la funzione di transizione
- $q_0 \in Q$ è lo stato iniziale
- $q_p \in Q$ è lo stato di stampa
- $q_h \in Q$ è lo stato di terminazione $(q_p \neq q_h)$

La computazione di un enumeratore $E$ è definita in modo analogo a quella di una MdT tradizionale, eccetto per le seguenti considerazioni. Esso ha due nastri, un nastro di lavoro e uno di stampa, entrambi inizialmente vuoti. Ad ogni passo, la macchina può scrivere un simbolo di $\Gamma$ sul nastro di output, o nessuno, sulla base di $\delta$. Se $\delta(q, a) = (r, b, L, c)$, significa che nello stato $q$, leggendo $a$, l'enumeratore $E$ entra nello stato $r$, scrive $b$ sul nastro di lavoro, muove la testina del nastro di lavoro a sinistra, scrive $c$ sul nastro di output e muove la testina del nastro di output a destra (se $c \neq \varepsilon$). Ogni volta che la macchina entra nello stato $q_p$, il nastro di output viene ripulito e la testina ritorna all'estremità sinistra. La macchina si ferma quando entra nello stato $q_h$. Il linguaggio di $E$ viene definito nel seguente modo: $L(E) = \{w \in \Sigma^* \mid w \text{ appare nel nastro di lavoro se si entra in } q_p\}$, insieme esprimibile più formalmente tramite la notazione $L(E) = \{w \in \Sigma^* \mid (q_0, \varepsilon) \mapsto^* (uq_pv, w) \text{ per qualche } u, v \in \Sigma^*\}$.

La configurazione di un enumeratore deve contenere le informazioni di nastro di lavoro, stato corrente, posizione della testina del nastro di lavoro e nastro di output (che possiamo chiamare anche *buffer*): una configurazione di un enumeratore si identifica quindi con la coppia $(uq_iv, w)$, dove $u, v \in \Gamma^*$, $q_i \in Q$ e $w \in \Gamma^*$.
Valgono inoltre le seguenti regole per l'evoluzione delle configurazioni:

- Funzione di transizione con spostamento della testina a sinistra:

$$\frac{\delta(q_i, b) = (q_j, c, L, d)}{(uaq_ibv, w) \mapsto (uq_jacv, wd)}$$

- Funzione di transizione con spostamento della testina a destra:

$$\frac{\delta(q_i,\, b) = (q_j,\, c,\, R,\, d)}{(uaq_ibv,\, w) \mapsto (uacq_jv,\, wd)}$$

- Funzione di transizione con spostamento della testina a sinistra (testina a inizio nastro):

$$\frac{\delta(q_i,\, b) = (q_j,\, c,\, L,\, d)}{(q_ibv,\, w) \mapsto (q_jcv,\, wd)}$$

- Funzione di transizione con spostamento della testina a destra (testina a inizio nastro):

$$\frac{\delta(q_i,\, b) = (q_j,\, c,\, R,\, d)}{(q_ibv,\, w) \mapsto (cq_jv,\, wd)}$$

- Effetto dello stato di stampa:

$$(uq_pv,\, w) \mapsto (q_0uv,\, \varepsilon)$$

**Esercizio 2.** Descrivere a livello formale e implementativo una MdT che riconosca le stringhe binarie con lo stesso numero di 0 e di 1.

Partiamo dalla descrizione a livello implementativo.

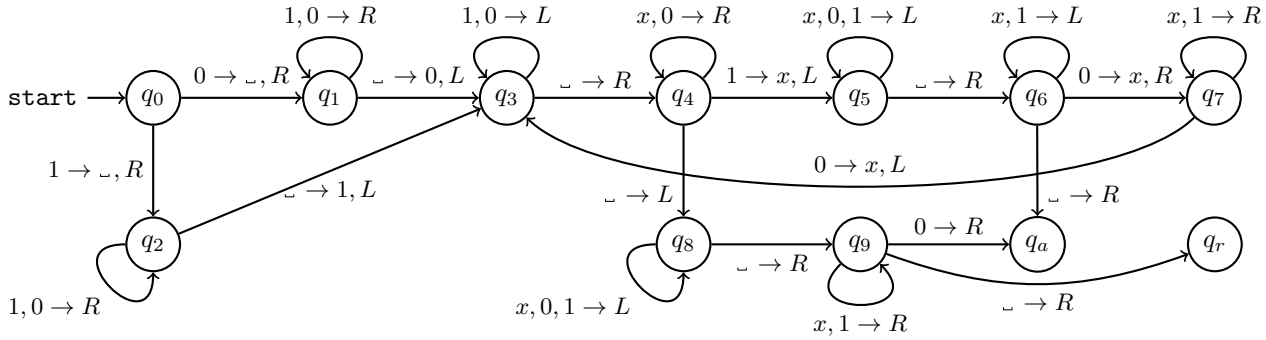
1. Scorri il nastro fino a trovare uno 0. Se non lo trovi, vai al passo 4.
2. Altrimenti cancellalo, riavvolgi il nastro e cerca un 1 da cancellare. Se non lo trovi, rifiuta.
3. Riavvolgi il nastro e torna al punto 1.
4. Riporta il nastro all'inizio e scorrilo tutto. Se trovi un 1, rifiuta. Altrimenti, accetta.

La descrizione formale di tale macchina è riportata di seguito. Consiste in una traduzione diretta della descrizione implementativa, l'unica parte itneressante è quella a sinistra (stati $q_0$, $q_1$, $q_2$ e $q_3$), in cui andiamo a marcare l'inizio del nastro con il simbolo $\llcorner$; il primo simbolo viene poi copiato in fondo al nastro, in modo da mantenere invariato il numero di 0 e di 1 rispettivamente.

**Esercizio 3.** Descrivere a livello implementativo una MdT che riconosca l'insieme delel stringhe binarie che NON contengono il doppio degli 0 rispetto agli 1.

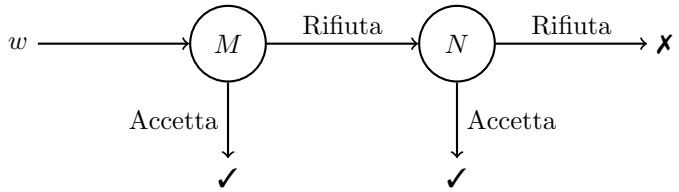
1. Scorri il nastro fino a trovare un 1. Se non lo trovi vai al punto 4, altrimenti cancellalo e vai al punto 2.
2. Riporta il nastro all'inizio e cerca uno 0 da cancellare. Se lo trovi vai al punto 3, altrimenti accetta.
3. Continua a scorrere cercando un altro 0 da cancellare. Se lo trovi, vai al punto 1. Altrimenti, accetta.
4. Riavvolgi il nastro e scorrilo tutto. Se trovi uno 0, accetta. Altrimenti, rifiuta.

Viene di seguito proposta una descrizione formale della macchina attraverso il suo diagramma.




**Esercizio 4.** Dimostrare che la classe dei linguaggi Turing-riconoscibili è chiusa rispetto alle operazioni di:

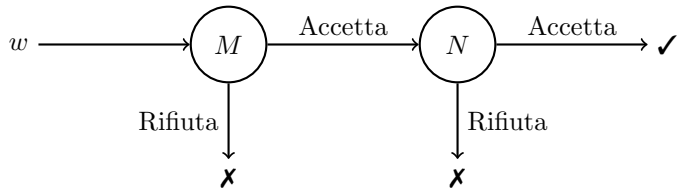
- **Unione**: Siano $A$ e $B$ due linguaggi TR, vogliamo dimostrare che $A \cup B$ è TR. Poiché $A$ e $B$ sono TR, esistono due MdT $M$, $N$ tali che $L(M) = A$ e $L(N) = B$.
  Un'idea intuitiva, seppur sbagliata, prevede di creare una macchina che consiste nella concatenazione delle due MdT $M$ e $N$: la stringa viene ricevuta prima da $M$, e se la rifiuta la passa ad $N$ nella speranza che la accetti.




  Tale soluzione non riconosce l'unione dei linguaggi qualora la macchima $M$ entri in un *loop* infinito: in questo modo, anche se $N$ dovesse accettare la stringa passata in input, non le arriverebbe nemmeno.
  Un'idea migliore prevede di utilizzare una MdT dotata di due nastri, inizializzati entrambi con la stringa in input, sui quali agiranno rispettivamente le macchine $M$ ed $N$. Non avrebbe senso far eseguire prima la macchina $M$ e poi la macchina $N$ per il motivo dell'idea errata

appena proposta, per cui la computazione complessiva della macchina prevede l'alternarsi di un passo di computazione alla volta per la macchina $M$ e la macchina $N$, in modo da evitare il blocco della nuova macchina dovuto al *loop* di $M$ o di $N$. Se una fra le due macchine accetta, accettiamo anche noi, altrimenti rifiutiamo.

- **Intersezione**: Siano $A$ e $B$ due linguaggi TR, vogliamo dimostrare che $A \cap B$ è TR. Poiché $A$ e $B$ sono TR, esistono due MdT $M$, $N$ tali che $L(M) = A$ e $L(N) = B$.
  Contrariamente al caso dell'unione, qui è possibile utilizzare una concatenazione intuitiva delle due macchine $M$ e $N$.




  Non ci dobbiamo preoccupare del caso in cui una delle due macchine entri in un *loop*, perché non è possibile che se $M$ non riconosce il linguaggio $N$ possa farlo.

- **Concatenazione**: Siano $A$ e $B$ due linguaggi TR, vogliamo dimostrare che $A \circ B$ è TR. Poiché $A$ e $B$ sono TR, esistono due MdT $M$, $N$ tali che $L(M) = A$ e $L(N) = B$.
  Vogliamo creare una MdT $O$ tale che $L(O) = A \circ B$. Assumiamo che $O$ sia non-deterministica, e definiamo in modo seguente il suo funzionamento:
  Su input $w$:

  1. Separa non-deterministicamente $w$ in $w_1 w_2$.
  2. Esegui $M$ su $w_1$. Se accetta, vai al punto 3. Se rifiuta, rifiuta.
  3. Esegui $N$ su $w_2$. Se accetta, vai in *accept*. Se rifiuta, rifiuta.

  Neanche in questo caso dobbiamo preoccuparci di un eventuale *loop* da parte di $M$ o di $N$: per il non-determinismo, ci sarà un ramo delle computazioni che terminerà con l'accettazione della stringa, a prescindere dalle altre eventuali computazioni che rifiutano o entrano in *loop*.

- **Star**: Sia $A$ un linguaggio TR, vogliamo dimostrare che $A^*$ è TR. Poiché $A$ è TR, esiste una MdT $M$ tale che $L(M) = A$. Vogliamo costruire una MdT $N$ tale che $L(N) = A^*$. Anche in questo caso, assumiamo che $N$ sia non-deterministica, e definiamo in modo seguente il suo funzionamento: Su input $w$:

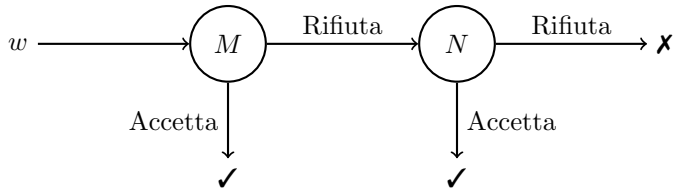
  1. Separa non-deterministicamente $w$ in $w_1 w_2 \dots w_n$, per qualche $n \leq |w|$.
  2. Esegui $M$ su $w_i$ per ogni $i \leq n$. Se $M$ le accetta tutte, allora vai in *accept*. Altrimenti, rifiuta.

  Se esiste un modo per dividere $w$ in sottostringhe tali che $M$ le accetta tutte, allora $w$ appartiene a $A^*$ e $N$ accetterà $w$ dopo un numero finito di passaggi.

**Esercizio 5.** Dimostrare che la classe dei linguaggi decidibili è chiusa rispetto alle operazioni di:

- **Unione**: Siano $A$ e $B$ due linguaggi decidibili, vogliamo dimostrare che $A \cup B$ è decidibile. Poiché $A$ e $B$ sono decidibili, esistono due decisori $M$ e $N$ tali che $L(M) = A$ e $L(N) = B$. Siccome siamo nel contesto dei linguaggi decidibili, sappiamo che né $M$ né $N$ potranno mai

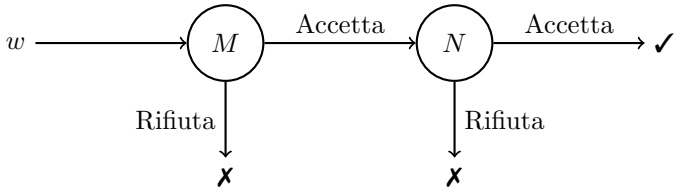
trovarsi in una situazione di *loop*, per cui possiamo implementare, per questa dimostrazione, un decisore $O$ costituito dalla concatenazione di $M$ e $N$ tale che $L(O) = A \cup B$ secondo l'idea iniziale che abbiamo avuto per la chiusura dei linguaggi Turing-riconoscibili rispetto all'operazione di unione.



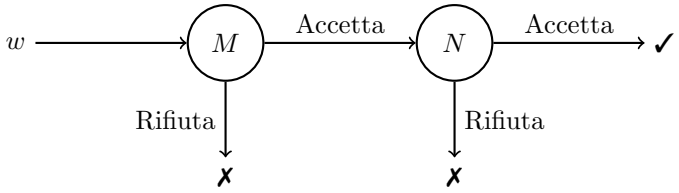

$O$ opererà nel seguente modo:

Su input $w$:

1. Esegui $M$ su $w$. Se lo accetta, vai in *accept*. Altrimenti, vai al punto 2.
2. Esegui $N$ su $w$. Se lo accetta, vai in *accept*, altrimenti rifiuta.




- **Intersezione**: Siano $A$ e $B$ due linguaggi decidibili, vogliamo dimostrare che $A \cap B$ è decidibile. Poiché $A$ e $B$ sono decidibili, esistono due decisori $M$ e $N$ tali che $L(M) = A$ e $L(N) = B$. Vogliamo creare un decisore $O$ che riconosca il linguaggio $L(O) = A \cap B$.
  Anche in questo caso, possiamo riciclare la costruzione di $O$ come nel caso dei linguaggi Turing-riconoscibili: un decisore $O$ costituito dalla concatenazione di due decisori $M$ e $N$ riconosce l'intersezione dei loro linguaggi se e solo se entrambi $M$ e $N$ accettano la stringa passata in input a $O$.



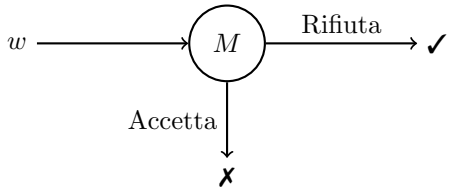

Nemmeno qui dobbiamo preoccuparci di eventuali *loop* di $M$ o $N$, in quanto, essendo decisori, devono necessariamente terminare. Il funzionamento di $O$ si può descrivere in questo modo:

Su input $w$:

1. Esegui $M$ su $w$. Se la rifiuta, rifiuta. Altrimenit, vai al passo 2.
2. Esegui $N$ su $w$. Se la accetta, vai in *accept*. Altrimenti, rifiuta.

- **Concatenazione**:

- **Star**:
- **Complemento**: Sia $A$ un linguaggio decidibile, vogliamo dimostrare che $\bar{A}$ è decidibile. Poiché $A$ è decidibile, esiste un decisore $M$ tale che $L(M) = A$. Vogliamo creare un decisore $N$ tale che $L(N) = \bar{A}$.
  Affinché $N$ riconosca il complemento di $A$, è sufficiente considerare l'esito opposto della computazione di $M$ sulla stringa in input $w$.




  La macchina $N$ funzionerà in questo modo:

  Su input $w$:

  1. Esegui $M$ su $w$. Se $M$ accetta, rifiuta. Altrimenti, vai in *accept*.

# 7 Decidibilità

## 7.1 Algoritmi per linguaggi regolari e *context-free*

**Accettazione - DFA** Definiamo $A_{DFA} = \{< B, w > \mid B \text{ è un DFA che accetta } w\}$. Questo linguaggio, che ricordiamo essere costituito da un insieme di stringhe, contiene tutte le rappresentazioni sotto forma di stringhe (notiamo l'uso dei simboli $<$ e $>$) delle coppie di DFA $B$ e linguaggi $w$ tali che $B$ riconsoce $w$. Non è importante come noi decidiamo di memorizzare $B$, l'importante è che ci accordiamo sul formato con cui noi lo vogliamo rappresentare (*e.g.*, potremmo descriverlo come una quintupla utilizzando i formati JSON o XML).
Tale linguaggio si associa ad un problema di tipo algoritmico: è un problema di accettazione (per convenzione, i problemi di accettazione sono identificati col nome $A$) per un determinato tipo di computazione, in questo caso del modello del DFA (notare il pedice, sempre per convenzione).
$A_{DFA}$ è un linguaggio decidibile: per dimostrarlo, è necessario costruire un decisore per tale linguaggio. Costruiamo il decisore $M$ tale che $L(M) = A_{DFA}$, che possiamo descrivere in questo modo:

$M =$ Su input $< B, w >$:

1. Simula $B$ su $w$.
2. Se la simulazione termina con l'accettazione, accetta. Altrimenti, rifiuta.

Tale dimostrazione sembra una tautologia, eppure ci sono dei punti su cui è necessario riflettere. Questa simulazione è possibile, per una Macchina di Turing, perché abbiamo concordato il formato dei DFA, e sappiamo come farne il parsing, cioè come riconoscerne la struttura e di comprenderne la funzione di transizione. Grazie a queste informazioni, siamo in grado di poter usare il nastro della MdT per tenere traccia dello stato in cui entra il DFA man mano che consumiamo i caratteri di $w$. Per via del comportamento deterministico dei DFA, sappiamo che dopo un certo numero finito di passi la computazione necessariamente terminerà in uno stato di *accept* oppure di *reject*.

**Accettazione - NFA** Consideriamo una variante del linguaggio appena visto: $A_{NFA} = \{< B, w > \mid B \text{ è un NFA e } w \text{ è una striga accettata da esso}\}$. Anche questo linguaggio risulta decidibile, e la dimostrazione avviene in modo analogo a quanto abbiamo appena visto. Essendo $B$ una macchina non deterministica, vogliamo determinare una MdT non deterministica $N_1$ che si comporta come segue:

$N_1 =$ Su input $< B, w >$:

1. Simula $B$ su $w$.
2. Ritorna il suo risultato.

Risolvere il non determinismo attraverso l'impiego di una macchina non determinisitca consiste sicuramente nella scelta più facile, tuttavia è possibile dimostrare che $A_{NFA}$ è un linguaggio decidibile anche attraverso una MdT deterministica. Descriviamo il comportamento di una MdT deterministica $N_2$ che si presta a risolvere questo problema:

$N_2 =$ Su input $< B, w >$:

1. Converti $B$ in un DFA equivalente $C$.

2. Dai in input $< C, w >$ al decisore per $A_{DFA}$.
3. Ritorna il suo output.

Abbiamo dimostrato all'inizio del corso che NFA e DFA sono equivalentemente espressivi, fornendo un algoritmo costruttivo, di traduzione uno a uno da NFA a DFA, per cui l'algoritmo di funzionamento di $N_2$ è corretto e sfrutta il risultato dimostrato prima per $A_{DFA}$.
Quello che abbiamo appena fatto prende il nome di **dimostrazione per riduzione**: abbiamo trasformato un problema in un altro problema, in questo caso molto vicino ad esso, sfruttando l'algoritmo risolutivo di quest'ultimo problema.

**Accettazione - *Regexp*** Consideriamo il seguente caso dello studio di algoritmi per linguaggi regolari: sia definito il linguaggio $A_{REX} = \{< R, w > \mid R$ è una *regexp* e $w$ una stringa generata da $R\}$. Anche questa è un'instanza di un linguaggio decidibile, e la dimostrazione avviene in modo analogo al caso appena visto. Vogliamo costruire una MdT $O$ tale che $L(O) = A_{REX}$; essa sarà definita nel seguente modo:

$O =$ Su input $< B, w >$:

1. Converti $B$ in un NFA equivalente $C$.
2. Dai in input $< C, w >$ al decisore per $A_{NFA}$.
3. Ritorna il suo output.

***Emptiness* - DFA** Consideriamo ora il seguente linguaggio: $E_{DFA} = \{< A > \mid A$ è un DFA tale che $L(A) = \emptyset\}$. Il linguaggio appena proposto consiste in un'istanza del **problema dell'*emptiness***, condizione indicata dalla lettera $E$, per il caso dei DFA (notare il pedice); tale linguaggio riconosce tutte le rappresentazioni di DFA che non riconoscono alcun linguaggio, e si può dimostrare che anche questo linguaggio è decidibile. Per dimostrarlo, vogliamo costruire un decisore $T$ per $E_{DFA}$. Tale decisore si comporterà nel seguente modo:

$T =$ Su input $< A >$:

1. Marca lo stato iniziale di $A$.
2. Finché è possibile marcare nuovi stati:
    3. Marca ogni stato che è raggiungibile con una transizione da uno stato marcato.
4. Se nessuno stato accettante è stato marcato, accetta. Altrimenti, rifiuta.

Tale algoritmo è implementabile da una Macchina di Turing, perché siamo a conoscenza degli stati di $A$ e della sua funzione di transizione. Scopo di questo algoritmo è determinare se esiste un cammino dallo stato iniziale ad uno stato accettante all'interno di $A$: se non dovesse esistere, significa che $A$ non può accettare nessun linguaggio, e ciò rispetterebbe la condizione del problema dell'*emptiness*.
Siamo sicuri che il ciclo dato dal punto 3. prima o poi terminerà, dato che gli stati del DFA esistono in numero finito, per cui $T$ è necessariamente un decisore.

***Equity* - DFA** Consideriamo il linguaggio $EQ_{DFA} = \{< A, B > \mid A, B \text{ sono DFA tali che } L(A) = L(B)\}$. Vogliamo dimostrare che anche $EQ_{DFA}$ è decidibile. Chiaramente, non possiamo testare tutte le possibili stringhe dei linguaggi di $A$ e $B$, in quanto, se dovesse effettivamente verificarsi la condizione $L(A) = L(B)$, l'algoritmo della MdT non terminerebbe perché le stringhe dei linguaggi sono infinite. Vogliamo costruire un decisore per $EQ_{DFA}$ sfruttando la seguente proprietà: $A = B$ se e solo se $(L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B)) = \emptyset$. A parole, la proprietà ci dice che $A$ e $B$ riconoscono lo stesso linguaggio se non esistono stringhe che stanno in $L(A)$ e nel complemento di $L(B)$ o in $L(B)$ e nel complemento di $L(A)$, cioè stringhe che appartengono a $L(A)$ ma non a $L(B)$ oppure a $L(B)$ ma non a $L(A)$. Sfruttando questa proprietà, definiamo il comportamento del decisore $F$:

$F$ = Su input $< A, B >$:

1. Costruisci un DFA $C$ tale che $L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$.
2. Esegui il decisore per $E_{DFA}$ su $C$.
3. Ritorna il suo risultato.

Possiamo costruire il DFA $C$ perché la classe dei linguaggi regolari è chiusa rispetto alle operazioni di unione, intersezione e complemento, di conseguenza siamo sicuri che i risultati di queste operazioni tra linguaggi regolari restituiscono altri linguaggi regolari riconosciuti da un qualche DFA. Inoltre, l'operazione eseguita al punto 2. ci permette di capire se il linguaggio di $C$ è vuoto o meno, cioè ci permette di verificare la proprietà che vogliamo sfruttare.

**Accettazione - CFG** Facciamo un passo avanti, nella direzione dei linguaggi *context-free*, e definiamo il linguaggio $A_{CFG} = \{< G, w > \mid G \text{ è una CFG e } w \in L(G)\}$. Anche $A_{CFG}$ è decidibile, e per dimostrarlo vogliamo provare due approcci, uno scorretto e uno corretto: uno potrebbe pensare di voler generare tutte le derivazioni di $G$, e verificare che, tra tutte le derivazioni generate, se ne trovi una che riconosca la stringa $w$. Prendiamo un esempio per dimostrare che tale approccio non è corretto: sia $G$ la grammatica data dalla regola $S \to 0S1 \mid \varepsilon$. Ci chiediamo di verificare che la stringa 00111 appartenga al linguaggio riconosciuto da $G$, generando le seguenti derivazioni:

$$
\begin{aligned}
&S \Rightarrow \varepsilon \\
&S \Rightarrow 0S1 \Rightarrow 01 \\
&S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011 \\
&\dots
\end{aligned}
$$

Andando avanti all'infinito, ci accorgeremmo che la stringa che vogliamo verificare non viene mai generata e il decisore non terminerebbe, dal momento che $w \notin L(G) = \{0^n 1^n \mid n \geq 0\}$. Qualcuno potrebbe pensare di sfruttare il numero di 0 di $w$ e limitare su di esso il numero di derivazioni, ma questo non è un approccio algoritmico, in quanto fa leva di una conoscenza pregressa sulla natura dell'input.
L'approccio corretto prevede un ragionamento più fine: l'idea alla base della dimostrazione consiste nel fatto che possiamo convertire $G$ nella forma normale di Chomsky. Ricordiamo che tale forma normale prevede l'esistenza di tre soli tipi di regole in una grammatica:

- $A \to BC (B, C \neq S)$
- $A \to a$
- $S \to \varepsilon$

Si può dimostrare che se $G$ è il forma normale di Chomsky e $w$ ha lunghezza almeno 1, allora $w$ è derivabile in $2|w| - 1$ passi. Se $w$ è la stringa vuota, essa può essere derivata in un passo.
Forti di questo teorema, che dimostreremo successivamente, possiamo definire il comportamento del decisore $S$ tale che $L(S) = A_{CFG}$:

$S$ = Su inpit $< G, w >$:

1. Converti $G$ in forma normale di Chomsky.
2. Se $w = \varepsilon$, genera le derivazioni di un passo. Altrimenti genera le derivazioni di $2|w| - 1$ passi.
3. Se hai trovato una derivazione per $w$ accetta, altrimenti rifiuta.

Il meccanismo di $S$ consiste in un'applicazione di un *upper bound* al numero di derivazioni possibili per generare $w$, in modo tale da avere una Macchina di Turing che sicuramente termina con esito positivo o negativo.

Dimostriamo, per completezza, il teorema appena utilizzato. La seconda parte del teorema è vera perché se $w$ è una stringa vuota, allora la forma normale di Chomsky ci assicura che essa è generabile attraverso una derivazione data dall'unica regola che ha, nella parte destra, la stringa vuota: $S \to \varepsilon$.
Nel caso in cui $w \neq \varepsilon$, immaginiamo di voler generare una stringa di lunghezza 3. Il processo di derivazione sarà simile al seguente:

$$A \Rightarrow \underbrace{BC \Rightarrow BDE}_{\text{2 step: non-terminali}} \Rightarrow \underbrace{aDE \Rightarrow acE \Rightarrow acb}_{\text{3 step: terminali}}$$

Questo esempio ci dovrebbe far capire che, per generare una stringa di lunghezza 3, abbiamo impiegato 2 step per generare dei caratteri non-terminali e altri 3 step per convertire questi non-terminali in terminali. In linea di principio, data una stringa lunga $n$, occorrono $n - 1$ passi per introdurre il numero sufficiente di non-terminali ed altri $n$ passi per convertire tali non-terminali nei corrispondenti terminali.

***Emptiness* - CFG** Consideriamo il linguaggio $E_{CFG} = \{< G > \mid G \text{ è una CFG e } L(G) = \emptyset\}$. Anche questo linguaggio è decidibile, e per dimostrarlo occorre progettare un decisore per esso.
L'idea alla base della dimostrazione consiste nel considerare le **variabili generanti**, cioè le variabili che possono generare stringhe di soli terminali. Supponiamo di avere un algoritmo che ci sappia dire quali sono le variabili generanti di $G$: dato questo algoritmo, il problema di partenza è risolto, in quanto scopriremmo se lo *start symbol* è generante oppure no, e se non dovesse esserlo allora il linguaggio della grammatica sarebbe l'insieme vuoto.
Descriviamo il funzionamento del decisore $R$, e implementiamo successivamente l'algoritmo appena citato.

$R$ = Su input $< G >$:

1. Marca tutti i terminali di $G$.
2. Finché puoi marcare qualcosa:
    3. Marca i non-terminali $A$ tali che esiste una produzione $A \to U_1 \dots U_k$ dove $U_1 \dots U_k$ sono tutti marcati.
4. Se lo *start symbol* $S$ è marcato rifiuta, altrimenti accetta.

Tale algoritmo capisce, per ogni variabile, se essa è generante o meno, in modo simile seppur leggermente diverso dall'algoritmo mostrato in $E_{DFA}$: partendo dai caratteri terminali, che vengono marcati, si marcano tutti i caratteri non-terminali che, nelle loro regole di derivazione, hanno nella parte destra solamente simboli marcati. Se, alla fine dell'algoritmo (che sicuramente termina in quanto le regole e i simboli di una grammatica sono finiti), lo *start symbol* è marcato, allora il linguaggio della grammatica non è vuota, dunque va rifiutato; altrimenti, va accettato.

***Equity* - CFG** Consideriamo il seguente linguaggio: $EQ_{CFG} = \{< G, H > \mid G, H$ sono CFG e $L(G) = L(H)\}$. Tale linguaggio, che contiene coppie di CFG che riconoscono lo stesso linguaggio, non è decidibile, e per il momento non abbiamo gli strumenti necessari per dimostrarne la non-decidibilità.

**Teorema**

| Ogni linguaggio *context-free* è decidibile. |
|---|

Questo teorema delinea quindi la seguente gerarchia tra i linguaggi che abbiamo studiato finora.

**Dimostrazione**

Sia $A$ un linguaggio *context-free*. Allora esiste uan CFG $G$ tale che $L(G) = A$. Vogliamo dimostrare che esiste un decisore il cui linguaggio è $A$. Sia $M_G$ tale decisore, e sia esso definito in questo modo:

$M_G =$ Su input $w$:

1. Esegui il decisore per $A_{CFG}$ su $< G, w >$.
2. Ritorna il suo output.

Per definizione del problema $A_{CFG}$, è quindi evidente che $L(M_G) = L(G)$, in quanto la stringa $w$ viene accettata da $M_G$ se viene riconosciuta dalla grammatica $G$.

Questa non è una dimostrazione costruttiva: non abbiamo dato una definizione per $G$, non sappiamo come opera, tuttavia per definizione di linguaggio *context-free* siamo sicuro che essa esiste e riconosce $w$, e questo per i nostri scopi dimostrativi è sufficiente.

## 7.2 Indecidibilità

Quelli mostrati nella *subsection* precedente sono esempi di problemi decidibili, tali per cui esiste un algoritmo che ne determina la decidibilità, fatta eccezione per $EQ_{CFG}$. Esso non è l'unico problema che possiamo definire indecidibile: esistono molte istanze di problemi indecidibili, alcuni con risvolti pratici più evidenti di altri, per cui la determinazione della decidibilità o della non decidibilità di un problema non consiste in un mero esercizio di stile prettamente matematico e teoretico.

Consideriamo il problema dell'accettazione per le Macchine di Turing, definito nel seguente modo: $A_{TM} = \{< M, w > \mid M \text{ è una MdT e } w \in L(M)\}$. Capire se una Macchina di Turing accetta una stringa oppure no è un problema indecidibile. Esso tuttavia è Turing-riconoscibile: la MdT $N$ che riconosce tale linguaggio è definita nel seguente modo.

$N$ = Su input $< M, w >$:

1. Simula $M$ su $w$.
2. Ritorna il suo output.

$N$ è una cosiddetta **Macchina di Turing universale**, in quanto simula l'esecuzione di un'altra Macchina di Turing. Tuttavia, questa macchina non è un decisore: qualora $w$ non dovesse terminare su $M$, anche $N$ non terminerebbe mai.

Dimostriamo che $A_{TM}$ non è decidibile, usando una tecnica matematica che prende il nome di **diagonalizzazione**. Diciamo che due insiemi $A$ e $B$ hanno la stessa cardinalità se e solo se esiste una biezione (cioè una funzione iniettiva e suriettiva) $f : A \to B$; essendo $f$ una biezione, vale che:

- $\forall\, x, y \in A,\ x \neq y \Rightarrow f(x) \neq f(y)$
- $\forall\, y \in B,\ \exists x \in A : f(x) = y$

Possiamo utilizzare questa definizione per dimostrare, per esempio, che l'insieme dei numeri naturali $\mathbb{N}$ e l'insieme dei numeri pari hanno la stessa cardinalità. La funzione che ci aiuta in questa dimostrazione è $f(n) = 2n$, funzione che mappa, per esempio, 1 a 2, 2 a 4, 3 a 6 *et cetera*.

Facciamo un passo avanti, e definiamo il concetto di **insieme numerabile**. Un insieme $A$ è numerabile se e solo se è finito o ha la stessa cardinalità di $\mathbb{N}$.
Si può dimostrare che anche l'insieme dei numeri razionali $\mathbb{Q} = \{\frac{m}{n} \mid m, n \in \mathbb{N}\}$ è numerabile. A differenza dell'insieme dei numeri pari, qui non è facile trovare una funzione di *mapping*: l'idea della dimostrazione consiste nel rappresentare $\mathbb{Q}$ come una matrice, dove sull'asse verticale sono riportati i numeratori e in quello orizzontale i numeratori.

Vogliamo trovare un modo di enumerarli: partire dalla prima cella in alto a sinistra e procedere con un'enumerazione orizzontale è un approccio sbagliato, perché gli elementi presenti su tale riga sono infiniti. Un approccio corretto prevede un'enumerazione diagonale, come illustrato nella figura seguente.

Tale enumerazione non è una funzione che riusciamo ad esprimere in maniera matematica formalmente concisa, tuttavia consiste in una funzione biettiva, in quanto ad ogni numero dell'insieme associamo uno ed un solo numero che lo "indicizza".
Nell'immagine in figura è mostrato il fatto che vengono ignorati numeri che sono equivalenti a numeri già enumerati (*e.g.*, $\frac{2}{2}, \frac{4}{2}, \frac{3}{3}, \frac{2}{4} \dots$).

Procediamo a dimostrare il fatto che l'insieme dei numeri reali $\mathbb{R}$ non è numerabile. Questo risultato ci dice che, informalmente, l'insieme dei numeri reali è più grande dell'insieme dei numeri naturali. La dimostrazione prevede l'uso della tecnica della diagonalizzazione: assumiamo per assurdo che $\mathbb{R}$ sia numerabile. Allora, esiste una biezione $f : \mathbb{N} \to \mathbb{R}$: proviamo a raffigurarla, associando ad ogni $n$ un numero appartenente a $\mathbb{R}$.

Procediamo a costruire un elemento di $\mathbb{R}$ che non compare nella colonna $f(n)$, condizione che violerebbe la suriettività della funzione. Costruiamo il numero $x$ tale che sia compreso tra 0 e 1, riportandone la sua forma decimale. Poiché la prima cifra decimale di $f(1)$ è 1, scegliamo una cifra diversa da 1, supponiamo 4. Successivamente, osserviamo la seconda cifra di $f(2)$: è pari a 5, quindi ne scegliamo una diversa, diciamo 6. Poi, osserviamo la terza cifra di $f(3)$: è pari a 3, per cui scegliamo un numero diverso, diciamo 4, e proseguiamo in questo modo.

Per il modo in cui $x$ è stato costruito, ovvero scegliendo, per ogni $i$-esima posizione decimale, una cifra diversa dalla $i$-esima cifra decimale per il numero $f(i)$, sappiamo che questo numero è diverso da tutti i numeri inseriti nella tabella. Questo è assurdo, perché $f$ dovrebbe essere una funzione suriettiva.

**Teorema**

| Esistono dei linguaggi che non sono Turing-riconoscibili. |
|---|

**Dimostrazione**

Vogliamo dimostrare che esistono più linguaggi che Macchine di Turing. Facciamo due osservazioni:

- L'insieme di tutte le stringhe (su un alfabeto $\Sigma$) è numerabile.
  Dato, per esempio, $\Sigma = \{0, 1\}$, vogliamo avere la facoltà di numerare le stringhe costruite su $\Sigma$; un esempio di enumerazione può essere il seguente: $\varepsilon$, 0, 1, 00, 01, 10, 11, 000, ....
- Corollario: l'insieme delle MdT è a sua volta numerabile. Questo perché ogni MdT ha una sua rappresentazione come stringa, cioè ho un sottoinsieme delle stringhe che codifica tutte le MdT.
  Abbiamo assunto, ma si può dimostrare, che sottoinsiemi di insiemi numerabili sono a loro volta numerabili.

L'aver dimostrato e compreso che l'insieme delle MdT è numerabile è un punto importante per la dimostrazione. Dimostriamo ora che l'insieme dei linguaggi non è numerabile.

Introduciamo il seguente linguaggio: $A = \{0, 00, 01, 000, 001, \dots\}$. Data, per esempio, l'enumerazione delle stringhe costruite su $\Sigma^*$ con $\Sigma = \{0, 1\}$ vista nella prima osservazione della corrente dimostrazione, è possibile rappresentare il linguaggio $A$ come una stringa di bit, dove l'$i$-esimo bit, corrispondente all'$i$-esimo elemento dell'enumerazione, è posto a 1 se tale elemento fa parte di $A$, a 0 altrimenti.

L'insieme delle stringhe infinite di bit non è numerabile: assumiamo per assurdo che lo sia; allora, possiamo costruire una tabella analoga a quella che abbiamo visto per la dimostrazione della non numerabilità di $\mathbb{R}$.

| $n$ | $f(n)$ |
|---|---|
| 1 | $\boxed{0}$0100... |
| 2 | 1$\boxed{1}$000... |
| 3 | 11$\boxed{1}$11... |
| 4 | 000$\boxed{1}$0... |
| $\vdots$ | ... |

Abbiamo visto che, attraverso la tecnica della diagonalizzazione, possiamo costruire un numero che avrà la forma $x = 1000\dots$ che sappiamo per certo che non sarà presente nella tabella, quindi abbiamo dimostrato che l'insieme delle stringhe infinite di bit non è numerabile.

Essendo l'insieme delle Macchine di Turing numerabile ma l'insieme delle stringhe infinite di bit non è numerabile, abbiamo concluso la dimostrazione del fatto che esistono più linguaggi che Macchine di Turing, e chequindi esistono linguaggi che non sono Turing-riconoscibili.

**Teorema**

$A_{TM} = \{< M, w > \mid M \text{ è una MdT e } w \in L(M)\}$ è indecidibile.

**Dimostrazione**

Assumiamo per assurdo che $A_{TM}$ sia decidibile. Esiste allora un suo decisore chiamato $H$:

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

Usiamo $H$ per costruire un altro decisore $D$:

$D =$ Su input $< M >$:

1. Esegue $H$ su $< M, < M >>$.
2. Inverte il suo output.

$$D(< M >) = \begin{cases} accept & \text{se } M \underline{\text{non}} \text{ accetta } < M > \\ reject & \text{altrimenti} \end{cases}$$

Allora,

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

Tuttavia, il comportamento di $D$ passatogli l'encoding di sé stesso è assurdo: la macchina deve rifiutare l'input se lo accetta, e viceversa, che è assurdo.

Spieghiamo meglio cosa avviene, rappresentando graficamente una matrice che indica l'accettazione di alcune Macchine di Turing delle loro rappresentazioni come stringhe. I valori di accettazione sono stati inseriti in maniera arbitraria. Dove non è presente alcun valore, la macchina rifiuta o entra in uno stato di *loop*.

| | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\cdots$ |
|---|---|---|---|---|---|
| $M_1$ | *accept* | | *accept* | | |
| $M_2$ | *accept* | *accept* | *accept* | *accept* | |
| $M_3$ | | | | | $\cdots$ |
| $M_4$ | *accept* | *accept* | | | |
| $\vdots$ | | | $\vdots$ | | |

Mostriamo ora una matrice simile, che rappresenta non l'esecuzione delle MdT in funzione delle varie rappresentazioni, bensì l'esecuzione del decisore $H$ passandogli come input $< M_i, < M_j >>$.

| | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\cdots$ |
|---|---|---|---|---|---|
| $M_1$ | *accept* | *reject* | *accept* | *reject* | |
| $M_2$ | *accept* | *accept* | *accept* | *accept* | $\cdots$ |
| $M_3$ | *reject* | *reject* | *reject* | *reject* | |
| $M_4$ | *accept* | *accept* | *reject* | *reject* | |
| $\vdots$ | | | $\vdots$ | | |

Quello che fa sostanzialmente la funzione $H(< M_i, < M_j >>)$ è riempire i buchi dati dal rifiuto o dal *loop* di una macchina nella tabella precedente, essendo $H$ un decisore.
È possibile inserire in questa tabella anche una *entry* per il decisore $D$, e sappiamo definirne il comportamento sull'encoding delle altre MdT semplicemente invertendo gli elementi della diagonale. Tuttavia, anche l'encoding di $D$, essendo essa una MdT, sarà presente tra gli altri encoding: non possiamo definire l'output, perché trovandosi sulla diagonale, dovrebbe essere l'opposto di sé stesso, il che risulta assurdo.

| | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\cdots$ | $\langle D \rangle$ | $\cdots$ |
|---|---|---|---|---|---|---|---|
| $M_1$ | *accept* | *reject* | *accept* | *reject* | | *accept* | |
| $M_2$ | *accept* | *accept* | *accept* | *accept* | $\cdots$ | *accept* | $\cdots$ |
| $M_3$ | *reject* | *reject* | *reject* | *reject* | | *reject* | |
| $M_4$ | *accept* | *accept* | *reject* | *reject* | | *accept* | |
| $\vdots$ | | | $\vdots$ | | $\ddots$ | | |
| $D$ | *reject* | *reject* | *accept* | *accept* | | ? | |
| $\vdots$ | | | $\vdots$ | | | | $\ddots$ |

**Teorema**

| $A$ è decidibile se e solo se sia $A$ che $\bar{A}$ sono Turing-riconoscibili. |
|---|

**Dimostrazione**

$\Rightarrow$) Sia $A$ decidibile. Allora, $\bar{A}$ è decidibile perché la classe dei linguaggi decibili è chiusa rispetto all'operazione di complemento. Ma allora, sia $A$ che $\bar{A}$ sono Turing-riconoscibili, essendo i linguaggi decidibili un sottoinsieme dei linguaggi Turing-riconoscibili.

$\Leftarrow$) Siano $A$ e $\bar{A}$ Turing-riconoscibili, e siano $M$ e $N$ due MdT tali che $L(M) = A$ e $L(N) = \bar{A}$. Voglio costruire un decisore $O$ tale che $L(O) = A$: non è possibile simulare prima $M$ o $N$ su $w$, in quanto se una delle due entra in un *loop*, allora anche $O$ entra in un *loop*, ma non può farlo perché è un decisore.
La soluzione consiste nel costruire $O$ come una MdT con due nastri, su cui vengono eseguiti alternando un passo d'esecuzione alla volta $M$ e $N$ su $w$, definita dal seguente comportamento:
$O =$ Su input $w$:

1. Simula alternativamente $M$ su $w$ sul primo nastro e $N$ su $w$ sul secondo.
2. Se $M$ accetta, accetta. Se $N$ accetta, rifiuta.

**Corollario**

| $\overline{A_{TM}}$ non è Turing-riconoscibile. |
|---|

**Dimostrazione**

$A_{TM}$ è Turing-riconoscibile, e l'abbiamo dimostrato a inizio sezione. Allora, $\overline{A_{TM}}$ non è Turing-riconoscibile: se lo fosse, allora $A_{TM}$ sarebbe riconoscibile per il teorema appena visto.

## 7.3 Esercizi sulla decidibilità

**Esercizio 1** Dimostrare che il problema di verificare se un DFA $D$ ed una *regexp* $R$ sono equivalenti è decidibile.

Il testo ci chiede di dimostrare che il linguaggio $A = \{< D, R > \mid D$ è un DFA, $R$ è una *regexp*, $L(D) = L(R)\}$ è decidibile. Per dimostrarlo, occorre trovare un decisore che riconosca tale linguaggio. Sia $M$ tale decisore; allora, il suo comportamento verrà definito in questo modo:

$M =$ Su input $< D, R >$:

1. Converti $R$ in un DFA equivalente $D'$.
2. Esegui il decisore per $EQ_{DFA}$ su $< D, D' >$.
3. Ritorna il suo output.

Il primo passo è giustificato dal teorema che abbiamo visto che ci permette di convertire qualsiasi *regexp* in un DFA equivalente, e il secondo passo è giustificato dal fatto che siamo riusciti a dimostrare che esiste un decisore per il problema di $EQ_{DFA}$.

**Esercizio 2** Dimostrare che il problema di determinare se un DFA accetta tutte le possibili stringhe (su un alfabeto $\Sigma$ fissato) è decidibile.

Il testo ci chiede di dimostrare che il linguaggio $B = \{< D > \mid D$ è un DFA $\wedge\ L(D) = \Sigma^*\}$ è decidibile. Per dimostrarlo, occorre trovare un decisore che riconosca tale linguaggio. Sia $M$ tale decisore; allora, il suo comportamento è definito in questo modo:

$M =$ Su input $< D >$:

1. Costruisci un DFA $D_{ALL}$ tale che $L(D_{ALL}) = \Sigma^*$.
2. Esegui il decisore per $EQ_{DFA}$ su $< D, D_{ALL} >$.
3. Ritorna l'output.

Sappiamo che il linguaggio di $D_{ALL}$, $\Sigma^*$, è regolare per almeno due motivi: il primo è che $\Sigma^*$ è il risultato di un'operazione regolare di Star, e sappiamo che la classe dei linguaggi regolari è chiusa rispetto a tale operazione. Ulteriormente, è possibile dimostrarne la regolarità disegnando un DFA dotato di un singolo stato, che è anche accettante, verso cui esiste una transizione per ogni carattere dell'alfabeto $\Sigma$, in modo che accetti qualsiasi stringa costruita su tale alfabeto.

Tale problema si può risolvere anche attraverso un altro approccio. Definiamo un ulteriore decisore $N$ tale che $L(N) = A$, che avrà il seguente comportamento:

$N$ = Su input $< D >$:

1. Costruisci un DFA $\overline{D}$ tale che $L(\overline{D}) = \overline{L(D)}$.
2. Esegui il decisore per $E_{DFA}$ su $< \overline{D} >$.
3. Ritorna l'output.

Dal momento che abbiamo verificato che la classe dei linguaggi è regolari è chiusa rispetto all'operazione di complemento, siamo giustificati per la creazione del DFA che riconosce il linguaggio complemento di $A$.

**Esercizio 3** Considerare il linguaggio $\overline{E}_{TM} = \{< M > \mid M$ è una MdT tale che $L(M) \neq \emptyset\}$. Dimostrare che $\overline{E}_{TM}$ è Turing-riconoscibile.

Un'idea intuitiva, seppur sbagliata, per affrontare l'esercizio, prevede il seguente ragionamento: poiché viene chiesta una generica MdT e non un decisore, possiamo testare tutte le stringhe. Una sua possibile implementazione può essere la seguente:

Su input $< M >$:

1. Per tutte le stringhe $s_1, s_2, \ldots$
   - Esegui $M$ su $s_i$
   - Se $M$ accetta, dai *accept*. Altrimenti passa alla stringa successiva.

Anche se dovesse esistere una stringa riconosciuta dal linguaggio, c'è la possibilità che qualche stringa causi un *loop* di $M$ prima che venga processata la stringa che verrebbe accettata. Per far funzionare tale approccio, bisogna modificare il metodo di esplorazione dello spazio degli stati, e apportare le seguenti modifiche alla soluzione:

Su input $< M >$:

Per $i = 1, 2, 3, \ldots$

1. Esegui $M$ su $s_1, s_2, \ldots, s_i$ per $i$ passi di computazione.
2. Se $M$ ha accettato almeno una stringa $s_j$ $(j \leq i)$, accetta.

**Esercizio 4** Considerare il linguaggio $C = \{< R > \mid R$ è una *regexp* tale che $L(R)$ contiene almeno una stringa con tre 1 consecutivi$\}$. Dimostrare che questo linguaggio è decidibile.

Il decisore che riconosce tale linguaggio è definito nel seguente modo:

Su input $< R >$:

1. Costruisci $R' = (0 \cup 1)^*111(0 \cup 1)^*$.
2. Converti $R$ in un DFA equivalente $D$ e $R'$ in un DFA equivalente $D'$.
3. Costruisci un DFA $D''$ tale che $L(D'') = L(D) \cap L(D')$.
4. Esegui il decisore per $E_{DFA}$ su $< D'' >$.

5. Ritorna l'opposto dell'output.

Il nocciolo di questa soluzione consiste nel verificare che $L(R)$, intersecato con $L(R')$, dove $R'$ è una *regexp* che identifica tutte le stringhe che contengono tre 1 consecutivi, non è un insieme vuoto. È possibile creare $R'$ ed essere sicuri che riconosce un linguaggio regolare perché è il risultato delle operazioni di unione e star, per cui abbiamo verificato la chiusura rispetto alla classe dei linguaggi regolari. Anche le operazioni di conversione delle *regexp* a DFA sono lecite, in quanto abbiamo visto un teorema che lo sancisce con tanto di algoritmo di conversione, ed anche la creazione di $D''$ è legittima in quanto abbiamo verificato la chiusura anche dell'operazione di intersezione. Infine, ci riduciamo al problema dell'*emptiness* per i DFA, di cui dobbiamo ritornare il risultato opposto.

**Esercizio 5** Considerare il linguaggio $\{< R, S > \mid R, S$ sono *regexp* tali che $L(R) \subseteq L(S)\}$. Dimostrare che questo linguaggio è decidibile.

Non possiamo sfruttare direttamente il risultato ottenuto dallo studio del problema dell'*equity*, in quanto non è chiesto di dimostrare l'uguaglianza dei tue linguaggi, bensì un rapporto di inclusione. Per aiutarci nella risoluzione dell'esercizio, osserviamo che $L(R) \subseteq L(S) \Leftrightarrow L(R) = L(R) \cap L(S)$. Definiamo quindi il comportamento del decisore che riconosce il linguaggio dato.

Su input $< R, S >$:

1. Converti $R, S$ in DFA equivalenti $D_R$, $D_S$.
2. Costruisci un DFA $D$ tale che $L(D) = L(D_R) \cap L(D_S)$.
3. Esegui il decisore per $EQ_{DFA}$ su $D_R, D$.
4. Ritorna l'output.

**Esercizio 6** Data una grammatica $G$, diciamo che la variabile $A$ è **usabile** se e solo se essa occorre nella derivazione di almeno una stringa $w \in L(G)$. Dimostrare che determinare se $A$ è usabile o meno è un problema decidibile.

Partiamo compiendo un'osservazione. Supponiamo di avere la seguente grammatica.

$$S \to A$$
$$A \to a \mid aA$$
$$B \to b$$

I tale grammatica, la variabile $B$ non è usabile, dal momento che non si può derivare dallo *start symbol*. Possiamo esprimere questa condizione come conseguenza di un teorema:

| Una variabile è **raggiungibile** se e solo se lo *start symbol* può generare una stringa che la contiene. |
|---|

Ma è sicuro che tutto ciò che è raggiungibile sia anche effettivamente usabile? Mostriamo un altro esempio di una situazione problematica.

$$S \to A \mid B$$
$$S \to aA$$
$$B \to b$$

In questo caso, la variabile $A$ presenta una ricorsione infinita: in essa manca il caso base, e l'unica regola prevede un suo reinserimento nella derivazione successiva. Ciò impedisce alla grammatica di generare una stringa composta solamente da caratteri terminali. Occorre definire un altro concetto.

Una variabile è **generante** se e solo se essa può derivare una stringa di soli terminali.

Immaginiamo ora di avere due algoritmi che ci sappiano dire, data una grammatica, se una sua variabile è raggiungibile e se è generante. Allora, sarebbe possibile verificare l'usabilità nel seguente modo:

1. Elimina da $G$ tutte le variabili non generanti e le produzioni che le contengono.
   Eliminare le variabili non generanti non cambia il linguaggio che le stringhe devono riconoscere, in quanto esse, per definizione, non permettono di generare una stringa di soli terminali.
2. Verifica se $A$ è raggiungibile.
   L'aver eliminato le variabili non generanti, se una variabile è raggiungibile allora lo *start symbol* può generare una stringa di terminali attraverso una derivazione che la contiene.

$$S \Rightarrow^* uAv \Rightarrow^* w$$

Fondamentalmente, quindi, una variabile risulta usabile se e solo se è sia raggiungibile che generante.

Per trovare le variabili raggiungibili, utilizziamo il seguente algoritmo:

1. Definisci l'insieme $Reach = \{S\}$.
2. Se esiste una produzione $B \to v$ con $B \in Reach$, allora $Reach = Reach \cup \{Vars(v)\}$.
3. Itera finché puoi estendere *Reach*.

Per trovare le variabili generanti, utilizziamo il seguente algoritmo:

1. Definisci l'insieme $Gen = \emptyset$.
2. Se esiste una produzione $B \to v$ dove $v$ comprende solo terminali e variabili in $Gen$, allora $Gen = Gen \cup \{B\}$.
3. Itera finché puoi estendere $Gen$.

I cicli di questi algoritmi, definiti **di punto fisso** in quanto continuano ad estendere un insieme finché non troviamo un punto dove l'applicazione dell'algoritmo non cambia il contenuto dell'insieme, convergono perché i simboli variabili della grammatica sono un numero finito.

**Esercizio 7** Dimostrare che il linguaggio $\{< D > \mid D$ è un DFA che non accetta stringhe con un numero dispari di 1$\}$ è decidibile.

Il decisore dovrà essere implementato in questo modo:

Su input $< D >$:

1. Costruisci una *regexp* $R = 0^*1(0^*10^*10^*)^*0^*$.
2. Converti $R$ in un DFA equivalente $D'$.
3. Costruisci un DFA $D''$ tale che $L(D'') = L(D) \cap L(D')$.
4. Esegui il decisore per $E_{DFA}$ su $< D'' >$.
5. Ritorna il risultato.

La soluzione del libro è la seguente.

"On input $\langle M \rangle$:

**1.** Construct a DFA $O$ that accepts every string containing an odd number of 1s.
**2.** Construct a DFA $B$ such that $L(B) = L(M) \cap L(O)$.
**3.** Test whether $L(B) = \emptyset$ using the $E_{\text{DFA}}$ decider $T$ from Theorem 4.4.
**4.** If $T$ accepts, *accept*; if $T$ rejects, *reject*."

# 8 Riducibilità

## 8.1 Il concetto di riducibilità

Nello studio delle categorie di linguaggi, ci siamo imbattuti in diverse strategie per dimostrare che un dato linguaggio appartenga o meno ad una certa categoria.
Abbiamo visto che possiamo dimostrare l'appartenenza di un linguaggio alla classe dei linguaggi regolari tramite lo sviluppo di un DFA, di un NFA o di una *regexp* che lo riconoscano, oppure la non appartenenza attraverso il *pumping lemma*.
Abbiamo inoltre scoperto di poter dimostrare l'appartenenza di un linguaggio alla classe dei linguaggi *context-free* sviluppando un PDA o una grammatica che lo descrivano, e analogamente la sua non appartenenza a tale classe tramite una simile applicazione del *pumping lemma*.
Per quanto riguarda i linguaggi decidibili, abbiamo imparato a riconoscerne l'appartenenza, tramite lo sviluppo di un decisore che riconosca il linguaggio in questione, ma non la non appartenenza: non è infatti sufficiente trovare una semplice Macchina di Turing che lo riconosca, in quanto l'esistenza di una MdT che lo riconosca (e che quindi proverebbe la sua appartenenza alla classe dei linguaggi Turing-riconoscibili) non esclude l'esistenza di un decisore per tale linguaggio; abbiamo visto che $A_{TM}$, il problema dell'accettazione di una Macchina di Turing, appartiene a questa categoria, ma la dimostrazione con cui l'abbiamo provato risulta macchinosa e non sarebbe conveniente utilizzare un metodo simile per dimostrare la non-decidibilità di ogni linguaggio di cui ci domandiamo la natura. In modo analogo, non abbiamo ancora scoperto i mezzi per capire se un linguaggio non è nemmeno Turing-riconoscibile.

Uno strumento più potente per determinare la non-decidibilità di un linguaggio prende il nome di **riducibilità**. Informalmente, possiamo dire che un problema $A$ è riducibile ad un problema $B$ ($A \leq B$) se una soluzione per $B$ mi consente di risolvere $A$. Il testo esemplifica la questione dando le seguenti possibili definizioni per $A$ e $B$.

$A$ : Orientarsi a Parigi

$B$ : Ottenere una mappa di Parigi

La riducibilità non ci dice nulla su quanto un problema, in assoluto, è complicato: tutto ciò che la relazione di riducibilità ci dice, per esempio nel caso mostrato, è che il problema $A$ non può essere più complicato di $B$, perché una soluzione per $B$ ci consente di risolvere anche $A$.
Un'importante conseguenza della relazione di riducibilità è che, dati due problemi $A$ e $B$ tali che $A \leq B$ e $B$ è decidibile, allora anche $A$ è decidibile. Analogamente, ed è un risultato importante, se $A \leq B$ e $A$ è indecidibile, allora anche $B$ è indecidibile: se fosse decidibile, significherebbe che $A$ dovrebbe essere difficile al più quanto $B$, ma $A$ non è decidibile per ipotesi. In questo modo, avendo a nostra disposizione un insieme di problemi indecidibili, è possibile dimostrare la non-decidibilità di altri problemi attraverso opportune riduzioni ad altri problemi che sappiamo essere indecidibili.

Vediamo un esempio di applicazione di questa tecnica, introducendo il seguente problema.

$$HALT_{TM} = \{< M, w > \mid M \text{ è una MdT che termina su } w\}$$

Il problema $HALT_{TM}$, che prende il nome di **problema della fermata** nel caso della Macchina di Turing, consiste nel determinare se una Macchina di Turing si ferma accettanto o rifiutando la stringa data in input.

**Teorema**

| $HALT_{TM}$ è indecidibile. |
|---|

**Dimostrazione**

Assumiamo per assurdo che $HALT_{TM}$ sia decidibile, e sia $R$ il suo decisore.
Vogliamo dimostrare che $A_{TM}$ è riducibile ad $HALT_{TM}$, $A_{TM} \leq HALT_{TM}$, da cui concluderemo che anche $HALT_{TM}$ deve essere indecidibile, ma questo è assurdo in quanto contraddice l'ipotesi.

Costruiamo quindi il seguente decisore per $A_{TM}$.

$S =$ Su input $< M, w >$:

1. Esegui $R$ su $< M, w >$.
2. Se $R$ rifiuta, rifiuta.
3. Se $R$ accetta, simula $M$ su $w$ e ritorna il suo output.

Essendo $R$ un decisore, il punto 1 terminerà sicuramente in un tempo finito e ci fornirà l'informazione riguardante la terminazione di $M$ su input $w$. Se tale esecuzione dà esito negativo, $S$ rifiuterà l'input, altrimenti simulerà l'esecuzione di $w$ su $M$ e ne ritornerà l'output, operazioni queste che terminano sicuramente e possono essere eseguite in un tempo finito.

Siamo quindi riusciti a dimostrare la decidibilità di $A_{TM}$, ma questo va contro alla dimostrazione per diagonalizzazione compiuta in precedenza che ne determina l'indecidibilità: siamo giunti ad un assurdo, per cui necessariamente anche l'ipotesi che $HALT_{TM}$ sia decidibile è assurda.

Tale dimostrazione è servita a mostrare un caso d'uso della riducibilità per dimostrare la non appartenenza di un linguaggio alla classe dei linguaggi decidibili. In questo caso, il decisore progettato per $A_{TM}$ risulta particolarmente intuitivo vista la vicinanza dei due problemi coinvolti, $A_{TM}$ e $HALT_{TM}$, tuttavia tale metodo è facilmente generalizzabile ed applicabile su instanze di problemi molto più lontane da $A_{TM}$ di $HALT_{TM}$.

Vediamo ora un altro esempio. Sia definito il seguente linguaggio.

$$E_{TM} = \{< M > \mid M \text{ è una MdT tale che } L(M) = \emptyset\}$$

$E_{TM}$ consiste in un'instanza del problema dell'*emptiness* applicata al caso delle Macchine di Turing. Anche questo problema, come $HALT_{TM}$, è indecidibile, e la dimostrazione avviene in maniera analoga.

**Teorema**

| $E_{TM}$ è indecidibile. |
|---|

**Dimostrazione**

Supponiamo per assurdo che $E_{TM}$ sia decidibile. Vogliamo costruire un decisore per $A_{TM}$ per dimostrare che $A_{TM} \leq E_{TM}$, che sappiamo essere assurdo in quanto $A_{TM}$ è indecidibile.
Sia $R$ il decisore per $E_{TM}$, e costruiamo il seguente decisore per $A_{TM}$.

$S$ = Su input $< M, w >$:

1. Esegui $R$ su $< M >$.
2. Se $R$ accetta, rifiuta.
3. Se $R$ rifiuta, *broken*.

Arrivati al punto 3, il sistema si rompe: il risultato di $R$ su $< M >$ ci dà solamente l'informazione che il linguaggio di $M$ non è vuoto, e che quindi accetta qualche tipo di stringa, ma non sappiamo nulla sull'appartenenza di $w$ al linguaggio di $M$: il sistema potrebbe benissimo accettarla, rifiutarla o *loopare* all'infinito, per cui tale approccio non è a noi utile per la dimostrazione.

Procediamo a formalizzare un'idea migliore. Tale idea prevede di implementare una nuova MdT $M_1$, che si comporta in questo modo: $M_1$ rifiuta tutti gli input tranne $w$, e su $w$ si comporta come $M$.

$M_1$ = Su input $x$:
1. Se $x \neq w$, rifiuta.
2. Altrimenti simula $M$ su $w$ e ritorna il suo output.

$$L(M_1) = \begin{cases} \{w\} & \text{se } M \text{ accetta } w \\ \emptyset & \text{altrimenti} \end{cases}$$

Definiamo ora il decisore.

$S$ = Su input $< M, w >$:

1. Costruisci una nuova MdT $M_1$ a partire da $M$ e $w$.
2. Esegui $R$ su $< M_1 >$.
3. Inverti il suo output.

$S$ consiste in un decisore che termina sicuramente, perché si basa su passi d'esecuzione terminabili in tempo finito. In particolare, l'esecuzione di $R$ su $< M_1 >$ ci permette di distinguere in quale dei due casi che definiscono il linguaggio di $M_1$ ci troviamo: se ritorna esito positivo, significa che il suo linguaggio è l'insieme vuoto, per cui dovremo rifiutare; altrimenti, accettiamo.

Osserviamo un altro esempio. Sia definito il linguaggio $REG_{TM}$.

$$REG_{TM} = \{< M > \mid M \text{ è una MdT tale che } L(M) \text{ è regolare}\}$$

**Teorema**

| $REG_{TM}$ è indecidibile. |
|---|

**Dimostrazione**

Assumiamo per assurdo che $REG_{TM}$ sia decidibile e sia $R$ il suo decisore. Vogliamo costruire un decisore per $A_{TM}$, risultato che ci porterebbe all'assurdo.

Vogliamo mettere in atto la stessa tecnica utilizzata nella dimostrazione precedente, sviluppando una nuova macchina che rispetti il seguente vincolo sul suo linguaggio:

$$L(M_2) = \begin{cases} \text{REGOLARE} & \text{se } M \text{ accetta } w \\ \text{NON REGOLARE} & \text{altrimenti} \end{cases}$$

Possiamo sviluppare una macchina $M_2$ arbitraria in modo che si comporti come segue.

$M_2 =$ Su input $x$:

1. Se $x$ è nella forma $0^n1^n$, allora accetta.
2. Altrimenti esegui $M$ su $w$ e accetta $x$ se $M$ accetta $w$.

Tenendo conto dell'osservazione compiuta prima sul linguaggio di $M_2$, $L(M_2)$ sarà definito in questo modo:

$$L(M_2) = \begin{cases} \Sigma^* & \text{se } M \text{ accetta } w \\ \{0^n1^n \mid n \geq 0\} & \text{altrimenti} \end{cases}$$

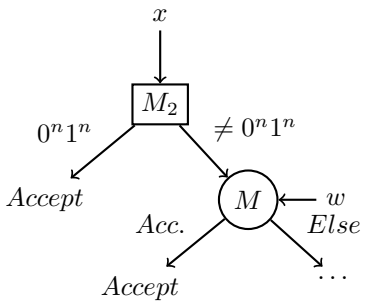
$\Sigma^*$ è sicuramente regolare, perché l'unione di tutti i simboli dell'alfabeto è un'operazione regolare che si può descrivere con un DFA o una *regexp*, mentre $\{0^n1^n \mid n \geq 0\}$ non è regolare, e lo abbiamo dimostrato quando abbiamo introdotto tale classe di linguaggi.

Il decisore per $A_{TM}$ si comporterà dunque nel seguente modo.

$S =$ Su input $< M, w >$:

1. Costruisci una nuova MdT $M_2$ a partire da $M$ e $w$.
2. Esegui $R$ su $< M_2 >$.
3. Restituisci il suo output.

L'utilizzo in questa dimostrazione ed in quella precedente di macchine come $M_1$ e $M_2$ serve per fornire un discriminante in base al quale si possono compiere delle operazioni sul loro linguaggio. Le precise implementazioni delle suddette macchine possono variare, purché venga mantenuta la proprietà richiesta sui loro linguaggi.



Consideriamo ora un'ulteriore linguaggio.

$$EQ_{TM} = \{< M_1, M_2 > \mid M_1,\ M_2 \text{ sono MdT e } L(M_1) = L(M_2))\}$$

**Teorema**

| $EQ_{TM}$ è indecidibile. |
|---|

**Dimostrazione**

Assumiamo per assurdo che $EQ_{TM}$ sia decidibile e sia $R$ il suo decisore. Vogliamo costruire un decisore per $E_{TM}$ per dimostrare la riduzione $E_{TM} \leq EQ_{TM}$, risultato che ci porterebbe all'assurdo.

Il decisore $S$ sarà definito nel seguente modo.

$S$ = Su input $< M >$:

1. Costruisci una MdT $N$ tale che $L(N) = \emptyset$.
2. Esegui $R$ su $< M, N >$.
3. Ritorna il suo output.

La macchina $N$ sarà definita in questo modo.

$N$ = Su input $x$:

1. *Reject.* $(L(N) = \emptyset)$

## 8.2 *Accepting configuration histories*

La riduzione mediante ***accepting configuration histories*** è una tecnica di riduzione che prevede l'uso di alcuni concetti ausiliari.

**Definizione**

Sia $M$ una MdT e $w$ il suo input. Una *accepting configuration history* è una sequenza di configurazioni $C_1, C_2, \ldots, C_k$ dove:
1. $C_1$ è la configurazione iniziale di $M$ su $w$
2. $C_k$ è una configurazione accettante
3. $\forall\, i$, $C_i$ produce $C_{i+1}$

- Se $M$ accetta $w$, ci saranno delle *accepting configuration histories*, o una sola se $M$ è determinisitca.
- Se $M$ rifiuta $w$ o va in *loop* su $w$, non ci sono *accepting configuration histories*.
- Le *accepting configuration histories* sono sequenze finite.

Occorre introdurre anche un nuovo modello computazionale ai fini dell'uso di tale tecnica.

**Definizione**

Un **Linear Bounded Automaton** (**LBA**) è una Macchina di Turing il cui nastro ha la stessa lunghezza dell'input iniziale.

Se la testina si sposta oltre l'estremità destra del nastro di un LBA, assumiamo che resti ferma, in modo analogo a quanto avviene in una Macchina di Turing tradizionale nel caso di uno spostamento oltre l'estremità sinistra del nastro.

Quello dei Linear Bounded Automaton è un modello computazionale meno espressivo di una Macchina di Turing arbitraria, in quanto la memoria è utilizzabile a piacere, tuttavia la quantità di memoria è finita e limitata linearmente dalla dimensione dell'input.

Procediamo a presentare alcuni risultati formali sugli LBA. Sia dato il linguaggio $A_{LBA}$.

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

**Teorema**

| $A_{LBA}$ è decidibile. |
|---|

Per dimostrarlo, dobbiamo fare uso di ulteriori risultati formali.

**Lemma** Se $M$ è un LBA con $q$ stati e $g$ simboli nell'alfabeto del nastro, allora esistono al più $qng^n$ configurazioni di $M$ per un input di lunghezza $n$.

**Dimostrazione** Ricordiamo le informazioni memorizzate in ogni configurazione.

- Stato interno del LBA ($q$ possibili stati).
- Stato del nastro ($g^n$: $g$ possibili caratteri posizionabili in $n$ possibili cele).
- Posizione della testina ($n$ possibili celle che può puntare la testina).

Di conseguenza, il numero totale di configurazioni possibili è $qng^n$.

Abbiamo ora gli strumenti necessari per dimostrare la decidibilità di $A_{LBA}$.

**Dimostrazione - Decidibilità di $A_{LBA}$** Vogliamo costruire un decisore $D$ che riconosca il linguaggio $A_{LBA}$.

$D$ = Su input $< M, w >$ ($M$ è LBA):

1. Simula $M$ su $w$ per $qng^n$ passi, dove $q$ è il numero di stati di $M$, $n$ è la lunghezza di $w$ e $g$ è il numero di simboli per il nastro.
2. Se $M$ si è fermata, dai il suo output. Altrimenti dai *reject*.

Supponiamo di aver generato $qng^n + 1$ configurazioni: ciò significa che qualche configurazione è stata vista necessariamente più di una volta, il che è un chiaro segnale di ingresso della macchina in uno stato di *loop*. Se dovessimo mai trovarci in questa situazione, significa che l'output non può essere accettato.

$$\underbrace{C_1, C_8, C_{62}, \ldots, C_{11}}_{qng^n+1} \Rightarrow \exists i, j \ : i \neq j \wedge C_i = C_j$$

Passiamo ora invece a mettere in pratica la tecnica di riduzione via *accepting configuration histories* per dimostrare l'indecidibilità del seguente linguaggio.

$$E_{LBA} = \{< M > \mid M \text{ è un LBA e } L(M) = \emptyset\}$$

**Teorema**

| $E_{LBA}$ è indecidibile. |
|---|

**Dimostrazione**

Vogliamo ridurre $A_{TM}$ a $E_{LBA}$. Poiché $A_{TM}$ è indecidibile, concluderemo che anche $E_{LBA}$ è indecidibile. Assumiamo quindi per assurdo che $E_{LBA}$ sia decidibile e dimostriamo l'esistenza di un decisore per $A_{TM}$, che risulterebbe in un assurdo. Sia $R$ il decisore per $E_{LBA}$, e costruiamo il seguente decisore per $A_{TM}$.

$N$ = Su input $< M, w >$ ($M$ è MdT):

1. Costruisci un LBA $L$ a partire da $M$ e $w$.
2. Esegui $R$ su $< L >$.
3. Inverti il suo output.

Vogliamo che il linguaggio di $L$ sia l'insieme delle *accepting configuration histories* di $M$ su $w$.

$$R(< L >) = \begin{cases} Accept & L(L) = \emptyset \Rightarrow \nexists \textit{ accepting configuration histories} \text{ di } M \text{ su } w \\ Reject & L(L) \neq \emptyset \Rightarrow \exists \text{ una } \textit{accepting configuration history} \text{ di } M \text{ su } w \end{cases}$$

Per capire come costruire $L$, forniamo un formato di rappresentazione di una *accepting configuration history*.

$$\#C_1\#C_2\#\ldots\#C_k\#$$

Definiamo quindi il modo in cui opera tale LBA.

1. Controlla che $C_1$ sia la configurazione iniziale di $M$ su $w$ (cioè che $C_1 \sim q_0 w$):
   - Controlla che $M$ sia nello stato iniziale.
   - Controlla che il nastro contenga $w$.
   - Controlla che la testina sia a sinistra.
2. Controlla che $C_k$ sia accettante (cioè che $C_k \sim u q_{accept} v$).
3. Controlla che $\forall\, i,\ C_i$ produce $C_{i+1}$.
   In questo punto, la macchina controlla che ogni coppia di configurazioni adiacenti presenti due configurazioni quasi uguali, che differiscono eventualmente nella posizione della testina, nei caratteri ad essa adiacenti e nello stato della macchina, in modo che tra una configurazione e quella successiva venga rispettata la funzione di transizione.

$$\underbrace{aabq_7c}_{C_i} \# \underbrace{aaq_{10}bb}_{C_{i+1}}$$

Esaminiamo ora il problema di capire se una certa grammatica genera tutte le stringhe del suo alfabeto.

$$ALL_{CFG} = \{< G > \mid G \text{ è una CFG tale che } L(G) = \Sigma^*\}$$

**Teorema**

| $ALL_{CFG}$ è indecidibile. |
|---|

**Dimostrazione**

Vogliamo ridurre $A_{TM}$ a $ALL_{CFG}$. Assumiamo per assurdo che $ALL_{CFG}$ sia decidibile e sia $R$ il suo decisore. Vogliamo quindi costruire un decisore per $A_{TM}$ per evidenziare l'assurdo.

Su input $< M, w >$:

1. Costruisci una CFG $G$ a partire da $M$ e $w$.
2. Esegui $R$ su $< G >$.
3. Inverti l'output.

Vogliamo che il linguaggio di $G$ sia l'insieme delle stringhe che non sono *accepting configuration histories* per $M$ su $w$.

$$R(< G >) = \begin{cases} Accept & L(G) \text{ contiene tutte le stringhe} \Rightarrow \nexists \textit{ acc. config. hist.} \text{ per } M \text{ su } w \\ Reject & \text{ci sono stringhe che non appartengono a } L(G) \Rightarrow \exists \text{ una } \textit{acc. config. hist.} \text{ per } M \text{ su } w \end{cases}$$

Vediamo ora come costruire $G$ tale che $L(G)$ contiene le stringhe che non sono *accepting configuration histories* per $M$ su $w$. Anziché dare $G$, costruiamo un PDA $P$ tale che $L(P) = L(G)$. Questo non è un problema, perché abbiamo visto che un PDA si può sempre convertire in una grammatica equivalente e viceversa, senza perdita di generalità.

Affinché una stringa nel formato $\#C_1\#C_2\#\ldots\#C_k\#$ non sia una *accepting configuration history*, deve valere almeno una delle seguenti tre condizioni.

1. $C_1$ non è una configurazione iniziale, oppure
2. $C_k$ non è una configurazione accettante, oppure
3. $\exists\, C_i : C_i$ non produce $C_{i+1}$.

Il PDA deve quindi fare un branch non-deterministico per verificare una fra le tre condizioni.

1. Controllare che $C_1$ non abbia il formato $q_0 w$.
2. Controllare che $C_k$ non abbia il formato $uq_{accept}v$.
3. Dal momento che un PDA lavora con uno *stack*, caricare una configurazione e poi confrontarla con la configurazione successiva non è un'operazione semplice; occorre cambiare formato, e alternare configurazioni memorizzate in formato "dritto" e altre in formato "invertito", come mostrato a lato.

$$\underbrace{abbq_7c}_{C_i}\#\underbrace{abdq_9d}_{C_{i+1}}$$
$$\underbrace{abbq_7c}_{C_i}\#\underbrace{dq_9dba}_{C_{i+1}^R}$$

Proponiamo infine un'ultimo linguaggio di cui vogliamo dimostrare l'indecidibilità.

$$EQ_{CFG} = \{< G, H > \mid G, H \text{ sono CFG e } L(G) = L(H)\}$$

**Teorema**

| $EQ_{CFG}$ è indecidibile. |
|---|

**Dimostrazione**

Per assurdo, sia $R$ un decisore per $EQ_{CFG}$. Vogliamo quindi costruire un decisore per $ALL_{CFG}$.

$S$ = Su input $< G >$:

1. Costruisci una CFG $H$ tale che $L(H) = \Sigma^*$.
2. Ritorna l'output di $R(< G, H >)$.

$H$ dovrà essere definita in questo modo: se $S$ è lo *start symbol*, per ogni terminale $l \in \Sigma$, dovrà esistere la regola $S \to lS$, oltre che alla regola $S \to \varepsilon$. In questo modo, $L(H) = \Sigma^*$.

## 8.3 Riduzione attraverso funzioni

Andando a definire il significato matematico che definisce la relazione di riduzione tra due problemi, è possibile ragionare nello specifico sulle proprietà di tale relazione. Attraverso la scrittura $A \leq_m B$ vogliamo indicare che $A$ è riducibile a $B$ tramite funzione, o che $A$ è mapping-riducibile a $B$.

Avere una definizione di mapping-riduzione più precisa del concetto generale di riduzione ci permette di dare dimostrazioni più approfondite per istanze di dimostrazioni più complesse: tra le altre cose, la riduzione attraverso funzioni ci permette anche di dimostrare la non-Turing-riconoscibilità di alcuni problemi.

**Definizione**

Una funzione $f : \Sigma^* \to \Sigma^*$ si dice **calcolabile** se e solo se esiste una Macchina di Turing $M$ tale che, dato un qualsiasi input $w$, $M$ su $w$ si ferma lasciando $f(w)$ sul nastro.

Sostanzialmente, una funzione è definita calcolabile se essa è simulabile da una Macchina di Turing.

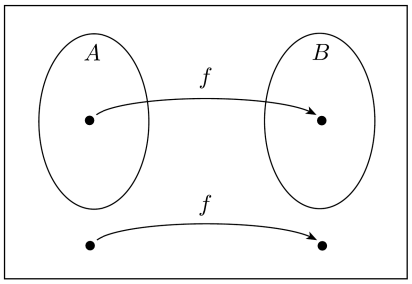
**Definizione**

$A \leq_m B$ se e solo se esiste una funzione calcolabile $f : \Sigma^* \to \Sigma^*$ tale che:

$$\forall\, w \in \Sigma^*,\ w \in A \Leftrightarrow f(w) \in B$$

La funzione $f$ si dice **riduzione**.

La doppia implicazione delinea un rapporto di univocità tra il dominio e il codominio della funzione: "srotolandola", essa ci dice sicuramente che se $w \in A$, allora $f(w) \in B$, ma anche che se $f(w) \in B$, allora $w \in A$, oltre al fatto che se $w \notin A$, allora $f(w) \notin B$.




**Teorema**

Se $A \leq_m B$ e $B$ è decidibile, allora $A$ è decidibile.

**Dimostrazione** Poiché $A \leq_m B$, essite una funzione calcolabile $f$ tale che $forall\, w,\ w \in A \Leftrightarrow f(w) \in B$. Datp che $B$ è decidibile, esiste un decisore $M$ per $B$. Costruiamo un decisore $N$ per $A$:

$N =$ Su input $w$:

1. Calcola $f(w)$.
2. Esegui $M$ su $f(w)$ e ritorna il suo output.

$$w \in A \Rightarrow f(w) \in B \Rightarrow M \text{ dice } accept \Rightarrow N \text{ accetta}$$
$$w \notin A \Rightarrow f(w) \notin B \Rightarrow M \text{ dice } reject \Rightarrow N \text{ rifiuta}$$

**Corollario**

| Se $A \leq_m B$ e $A$ è indecidibile, allora $B$ è indecidibile. |
|---|

Tale corollario sarà lo strumento che più useremo nella dimostrazione dell'indecidibilità dei problemi.

**Dimostrazione alternativa - Indecidibilità di $HALT_{TM}$** Utilizziamo le conoscenze acquisite finora per dare una dimostrazione alternativa dell'indecidibilità di $HALT_{TM}$, dimostrando che $A_{TM} \leq_m HALT_{TM}$. Vogliamo trovare una $f$ calcolabile tale che:

$$< M, w > \in A_{TM} \Leftrightarrow \underbrace{f(< M, w >)}_{<M', w'>} \in HALT_{TM}$$

Costruiamo una MdT $F$ che implementa $f$.

$F =$ Su input $< M, w >$:

1. Costruisci una MdT $M'$ come segue:
   Su input $x$:
   - Esegui $M$ su $x$.
   - Se $M$ accetta, dai *accept*.
   - Se $M$ rifiuta, vai in *loop*.
2. Ritorna $< M', w >$.

Garantiamo la correttezza di entrambi i versi della riduzione:

$< M, w > \in A_{TM} \Rightarrow M$ accetta $w \Rightarrow M'$ accetta $w \Rightarrow M'$ termina su $w \Rightarrow < M', w > \in HALT_{TM}$
$< M, w > \notin A_{TM} \Rightarrow M$ rifiuta $w$ o $M$ va in *loop* su $w \Rightarrow M'$ va in *loop* su $w \Rightarrow < M', w > \notin HALT_{TM}$

**Dimostrazione alternativa - Indecidibilità di $EQ_{TM}$** Procediamo allo stesso modo per dimostrare che $EQ_{TM}$ è indecidibile, verificando che $E_{TM} \leq_m EQ_{TM}$. Vogliamo trovare una $f$ calcolabile tale che $\forall w,\ w \in E_{TM} \Leftrightarrow f(w) \in EQ_{TM}$. Costruiamo quindi una MdT $F$ che implementa $f$:

$F =$ Su input $< M >$:

1. Costruisci una MdT $N$ tale che $L(N) = \emptyset$.
2. Ritorna $< M, N >$.

Garantiamo la correttezza di entrambi i versi della riduzione:

$$< M > \in E_{TM} \Rightarrow L(M) = \emptyset \Rightarrow L(M) = L(N) \Rightarrow < M, N > \in EQ_{TM}$$
$$< M > \notin E_{TM} \Rightarrow L(M) \neq \emptyset \Rightarrow L(M) \neq L(N) \Rightarrow < M, N > \notin EQ_{TM}$$

Facciamo un passo avanti. Ci vogliamo domandare se $A_{TM} \leq E_{TM}$.

Per verificarlo, dobbiamo trovare una $f$ calcolabile tale che $\forall x,\ x \in A_{TM} \Rightarrow f(x) \in E_{TM}$. Costruiamo una MdT $F$ che implementa $f$:

$F$ = Su input $< M, w >$:

1. Costruisci una MdT $M_1$ come segue:
   Su input $y$:
   - Se $y \neq w$, rifiuta.
   - Altrimenti esegui $M$ su $w$ e ritorna il suo output.
2. Ritorna $< M_1 >$.

La macchina $M_1$ è analoga a quella che abbiamo costruito quando abbiamo dimostrato la prima volta l'indecidibilità di $E_{TM}$, tuttavia il sistema presenta un problema: essa non rispetta la proprietà che vogliamo. Possiamo notarlo verificando la correttezza dei versi dell'implicazione:

$$< M, w > \in A_{TM} \Rightarrow M \text{ accetta } w \Rightarrow L(M_1) = \{w\} \Rightarrow L(M_1) \neq \emptyset \Rightarrow < M_1 > \notin E_{TM}$$
$$< M, w > \notin A_{TM} \Rightarrow M \text{ non accetta } w \Rightarrow L(M_1) = \emptyset \Rightarrow < M_1 > \notin E_{TM}$$

Abbiamo ottenuto l'opposto di quello che volevamo ottenere: formalmente, abbiamo dimostrato non che $A_{TM} \leq_m E_{TM}$, bensì che $A_{TM} \leq_m \overline{E}_{TM}$.

Dal punto di vista della decidibilità, effettuare una mapping-riduzione su un problema o sul suo complemento, purché uno dei due sia indecidibile, è equivalente, in quanto se uno dei due fosse decidibile e l'altro no, si potrebbe decidere il problema indecidibile semplicemente invertendo l'output del decisore.

**Teorema**

| Se $A \leq_m B$ e $B$ è Turing-riconoscibile, allora anche $A$ è Turing-riconoscibile. |
|---|

**Dimostrazione**

Se $A \leq_m B$, allora esiste una $f$ calcolabile tale che $\forall w,\ \in A \Leftrightarrow f(w) \in B$. Sia $B$ Turing-riconoscibile, allora esiste una MdT $M$ tale che $L(M) = B$. Costruisco una MdT $N$ tale che $L(N) = A$:

$N$ = Su input $w$:

1. Calcola $f(w)$.
2. Esegui $M$ su $f(w)$ e ritorna il suo output.

In questo caso, $N$ non è un decisore perché $M$ potrebbe *loopare* su $f(w)$, tuttavia non ci interessa perché vogliamo dimostrare la Turing-riconoscibilità.

$$w \in A \Rightarrow f(w) \in B \Rightarrow M \text{ accetta } f(w) \Rightarrow N \text{ accetta } w$$
$$w \notin A \Rightarrow f(w) \notin B \Rightarrow M \text{ non accetta } f(w) \Rightarrow N \text{ non accetta } w$$

**Corollario**

| Se $A \leq_m B$ e $A$ non è Turing-riconoscibile, allora $B$ non è Turing-riconoscibile. |
|---|

Arricchiamo questo corollario facendo una considerazione sul significato della doppia implicazione dovuta alla relazione di mapping-riducibilità.

$$\left.\begin{array}{l} w \in A \Rightarrow f(w) \in B \\ w \notin A \Rightarrow f(w) \notin B \end{array}\right\} \underbrace{\begin{array}{l} w \notin \overline{A} \Rightarrow f(w) \notin \overline{B} \\ w \in \overline{A} \Rightarrow f(w) \in \overline{B} \end{array}}_{\overline{A} \leq_m \overline{B}}$$

Sfruttiamo questo corollario per dimostrare che $EQ_{TM}$ non è Turing-riconoscibile. Sappiamo che $\overline{A}_{TM}$ non è Turing-riconoscibile, per cui un buon approccio sarebbe quello di provare che $\overline{A}_{TM} \leq_m EQ_{TM}$. Per facilità, dimostreremo una condizione equivalente, cioè che $A_{TM} \leq_m \overline{EQ}_{TM}$. Vogliamo trovare una $f$ calcolabile tale che $\forall x, \in A_{TM} \Leftrightarrow f(x) \in \overline{EQ}_{TM}$. Costruiamo $F$ che implementa $f$:

$F$ = Su input $< M, w >$:

1. Costruisci $M_1$ come segue:
   Su input $y$: *Reject*.
2. Costruisci $M_2$ come segue:
   Su input $z$: Esegui $M$ su $w$ e ritorna l'output.
3. Ritorna $< M_1, M_2 >$.

Garantiamo la correttezza di entrambi i versi della riduzione:

$$< M, w > \in A_{TM} \Rightarrow M \text{ accetta } w \Rightarrow L(M_2) = \Sigma^* \Rightarrow \overbrace{L(M_1)}^{\emptyset} \neq L(M_2) \Rightarrow < M_1, M_2 > \in \overline{EQ}_{TM}$$

$$< M, w > \notin A_{TM} \Rightarrow M \text{ non accetta } w \Rightarrow L(M_2) = \emptyset \Rightarrow L(M_1) = L(M_2) \Rightarrow < M_1, M_2 > \in EQ_{TM} \Rightarrow < M_1, M_2 > \notin \overline{EQ}_{TM}$$

Dimostriamo inoltre che $\overline{EQ}_{TM}$ non è Turing-riconoscibile. Per farlo, dobbiamo provare che $\overline{A}_{TM} \leq_m \overline{EQ}_{TM}$, cioè che $A_{TM} \leq_m EQ_{TM}$. Costruiamo quindi una MdT $F$ come segue.

$F$ = Su input $< M, w >$:

1. Costruisci $M_1$ come segue:
   Su input $y$: *Accept*.
2. Costruisci $M_2$ come segue:
   Su input $z$: Esegui $M$ su $w$ e ritorna il suo output.
3. Ritorna $< M_1, M_2 >$.

Possiamo anche in questo caso dimostrare analogamente la correttezza del ragionamento.

$$< M, w > \in A_{TM} \Rightarrow M \text{ accetta } w \Rightarrow L(M_2) = \Sigma^* \Rightarrow L(M_1) = L(M_2) \Rightarrow < M_1, M_2 > \in EQ_{TM}$$

$$< M, w > \in A_{TM} \Rightarrow M \text{ non accetta } w \Rightarrow L(M) = \emptyset \Rightarrow L(M_1) \neq L(M_2) \Rightarrow < M_1, M_2 > \notin EQ_{TM}$$

## 8.4 Turing-riducibilità

Possiamo porci un quesito: $\overline{A}_{TM} \leq_m A_{TM}$? Intuitivamente ci verrebbe da dire di sì: disponendo di un decisore per un problema, la sua variante complementata si dovrebbe poter ottenere invertendo l'output del decisore. Tuttavia, formalmente, questa proprietà non vale: sappiamo che $A_{TM}$ è Turing-riconoscibile, e abbiamo dimostrato che il suo complementare non lo è. Inoltre, abbiamo sempre visto che se un problema e il suo complemento sono Turing-riconoscibili, allora tale problema è decidibile, e sappiamo che non è questo il caso di $A_{TM}$.

Il concetto di mapping-riducibilità non incarna a pieno la nostra concezione intuitiva di riducibilità, per cui viene presentato di seguito un nuovo paradigma di riducibilità, denominato **Turing-riducibilità**. A tale scopo, è necessario introdurre un nuovo modello computazionale.

**Definizione**

Una **Macchina di Turing con oracolo** per un linguaggio $B$ è una MdT che può invocare un dispositivo esterno in grado di caprie se, data una stringa $w$, $w \in B$ o $w \notin B$.

$$M^B = \text{ Una MdT } M \text{ con oracolo per } B$$

Possiamo vedere l'oracolo come una funzione di libreria di un programma, di cui non ci è nota e non ci interessa l'implementazione, ma che ci può fornire a richiesta un'informazione sull'appartenenza di una stringa ad un determinato linguaggio.

**Definizione**

Dati due problemi $A$ e $B$, diciamo che $A$ è Turing-riducibile a $B$ ($A \leq_T B$) se e solo se esiste una MdT con oracolo per $B$ che decide $A$.

**Osservazione** $\forall\, A,\ \overline{A} \leq_T A$

**Dimostrazione** Costruiamo una MdT con oracolo per $A$ che decide $\overline{A}$.

$M^A$ = Su input $w$:

1. Passa $w$ all'oracolo per $A$.
2. Inverti il risultato.

Sotto questo punto di vista dunque, il concetto di Turing-riconoscibilità ci permette di mettere in relazione in maniera intuitiva il problema dell'accettazione per la Macchina di Turing e il suo complemento: $\overline{A}_{TM} \leq_T A_{TM}$.

Facciamo un po' di pratica per familiarizzare col concetto di Turing-riconoscibilità, dimostrando che $E_{TM} \leq_T A_{TM}$. Il decisore per $E_{TM}$ con oracolo per $A_{TM}$ sarà definito nel seguente modo.

$R^{A_{TM}}$ = Su input $< M >$:

1. Costruisci la seguente MdT:

   $N$ = Su input $x$:

   (a) Esegui $M$ in parallelo su tutte le stringhe possibili.

   (b) Se una stringa è accettata, accetta $x$.

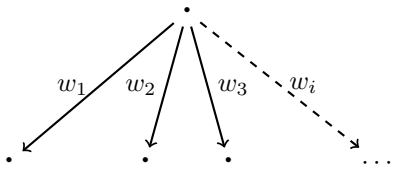
   $$L(N) = \begin{cases} \Sigma^* & \text{se } L(M) \neq \emptyset \\ \emptyset & \text{altrimenti} \end{cases}$$

2. Passa $< N, 0 >$ all'oracolo per $A_{TM}$.

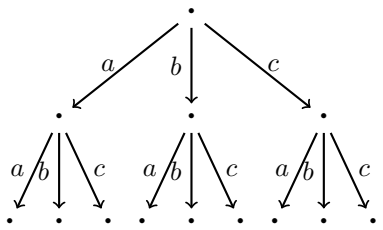
   - Se $< N, 0 > \in A_{TM}$, allora $0 \in L(N)$. Ma allora, siamo nel caso in cui $L(N) = \Sigma^*$, e quindi $L(M) \neq \emptyset$.
   - Se $< N, 0 > \notin A_{TM}$, allora $0 \notin L(N)$. Ma allora, siamo nel caso in cui $L(N) = \emptyset$, e quindi $L(M) = \emptyset$.

3. Inverti il suo output.

Dal momento che vogliamo eseguire $M$ in parallelo su tutte le possibili stringhe, questa macchina ha apparentemente bisogno di un numero infinito di nastri; potremmo pensare di risolvere la situazione tramite il non-determinismo tradizionale, tuttavia un branching non-deterministico del problema dalla radice causa la generazione di un numero infinito di esecuzioni, una per ciascuna stringa, per cui dobbiamo optare per una soluzione sempre non-deterministica, però con un numero di branch finito. Supponiamo di avere un alfabeto $\Sigma = \{a, b, c\}$. Allora, l'albero delle computazioni eseguirà 3 branch ad ogni nodo, uno per ciascun carattere dell'alfabeto, come mostrato in figura.




Branching incorretto




Branching corretto

**Teorema**

| Se $A \leq_T B$ e $B$ è decidibile, allora $A$ è decidibile. |
|---|

**Dimostrazione**

Poiché $A \leq_T B$, allora esiste una MdT con oracolo per $B$ che decide $A$. Poiché $B$ è decidibile, esiste un decisore $N$ tale che $L(N) = B$. Prendo la MdT con oracolo per $B$ e sostituisco le chiamate all'oracolo con chiamate a $N$, ottenendo un decisore per $A$.

**Osservazione**

Se $A \leq_T B$, non è detto che $A \leq_m B$. Abbiamo già visto un esempio: $\overline{A}_{TM} \leq_T A_{TM}$, ma $\overline{A}_{TM} \not\leq_m A_{TM}$.

**Teorema**

| Se $A \leq_m B$, allora $A \leq_T B$. |
|---|

**Dimostrazione**

Se $A \leq_m B$ , allora esiste una $f$ calcolabile tale che $\forall\, w,\ w \in A \Leftrightarrow f(w) \in B$. Costruisco una MdT con oracolo per $B$ che decide $A$.

$M^B$ = Su input $w$:

1. Calcola $f(w)$.
2. Passa $f(w)$ all'oracolo per $B$.
3. Ritorna il suo output.

Come conseguenza, possiamo notare che l'insieme dei problemi mapping-riducibili $\leq_m$ è un sottinsieme dei problemi Turing-riducibili $\leq_T$, e possiamo anche dire che il rapporto di inclusione è stretto, dal momento che abbiamo visto esempi di problemi tra di loro Turing-riducibili ma non mapping-riducibili.
Un vantaggio di questa condizione è sicuramente che la Turing-riducibilità cattura più problemi della mapping-riducibilità, tuttavia uno svantaggio consiste nel fatto che la Turing-riducibilità non c'è d'aiuto nel determinare la non-Turing-riconoscibilità di un linguaggio.

## 8.5 Teorema di Rice

### Enunciato

| |
|---|
| Qualsiasi proprietà non banale del linguaggio di una MdT è indecidibile. |

### Definizione

| |
|---|
| Una **proprietà di una Macchina di Turing** è un insieme di descrizioni della MdT. $\{< M > \mid M \text{ soddisfa } P\}$ |

Procediamo a formalizzare ulteriormente il Teorema.

Sia $P$ un linguaggio di descrizioni di una Macchina di Turing. Assumiamo che:

1. $P$ è non banale: $\exists M, N : < M > \in P \wedge < N > \notin P$.
2. $P$ è proprietà del linguaggio: $\forall\, M, N : L(M) = L(N) \Rightarrow (< M > \in P \Leftrightarrow < N > \in P)$

Allora $P$ è indecidibile.

Scartando una delle due ipotesi, il teorema si spaccherebbe: se la proprietà è banale, o ce l'hanno tutte le Macchine di Turing, o non ce l'ha nessuna, condizione che la rende necessariamente decidibile (o è sempre vera o è sempre falsa); inoltre, se la proprietà fosse della MdT ma non del linguaggio, essa non ci darebbe informazioni sul linguaggio, bensì sulla MdT (*e.g.*, se il numero di stati della MdT è pari).

### Dimostrazione

Per riduzione, assumiamo per assurdo che $P$ sia decidibile e costruiamo un decisore per $A_{TM}$. Sia $R$ il decisore per $P$. Poiché $P$ è non banale, devono esistere due macchine $S, T$ tali che $< T > \in P$ e $< S > \notin P$. Assumiamo, senza perdita di generalità, che $L(S) = \emptyset$. Costruiamo il seguente decisore per $A_{TM}$.

$U =$ Su input $< M, w >$:

1. Costruisci una MdT $N$ a partire da $M$ e $w$.
2. Esegui $R$ su $N$.
3. Ritorna il suo output.

Deve valere la seguente condizione sul linguaggio di $N$.

$$L(N) = \begin{cases} \text{Avere la proprietà } P & \text{se } M \text{ accetta } w \\ \text{Non avere la proprietà } P & \text{altrimenti} \end{cases} \Rightarrow \begin{cases} L(T) & \text{se } M \text{ accetta } w \\ L(S)\ (= \emptyset) & \text{altrimenti} \end{cases}$$

Dunque, $N$ sarà definita in questo modo.

$N =$ Su input $x$:

1. Simula $M$ su $w$. Se $M$ rifiuta, rifiuta.
2. Altrimenti, simula $T$ su $x$ e ritorna il suo output.

## 8.6 Esercizi

**Esercizio 1.** Considerare il linguaggio $EQ_{CFG} = \{< G, H > \mid G, H \text{ sono CFG e } L(G) = L(H)\}$. Dimostrare che $EQ_{CFG}$ è co-Turing-riconoscibile.

Vogliamo dimostrare che $\overline{EQ}_{CFG}$ è Turing-riconoscibile. Costruiamo a tal fine la seguente Macchina di Turing.

Su input $< G, H >$:

Per ogni stringa $w \in \Sigma^*$:

1. Esegui il decisore per $A_{CFG}$ su $< G, w >$.
2. Esegui il decisore per $A_{CFG}$ su $< H, w >$.
3. Se i due decisori danno input diversi, accetta.

Lo scopo di questa macchina è testare l'accettazione di qualsiasi stringa costruibile sull'alfabeto $\Sigma$ per le grammatiche $G$ e $H$. Se viene trovato un *mismatch*, allora le due grammatiche non riconoscono lo stesso linguaggio, per cui si può accettare.

**Esercizio 2.** Se $A \leq_m B$ e $B$ è regolare, allora $A$ è regolare. Dimostrare la verità o la falsità di questo enunciato.

$A \leq_m B$, quindi esiste una $f$ calcolabile tale che $\forall\, w,\ w \in A \Leftrightarrow f(w) \in B$. Il fatto che stiamo parlando di linguaggi regolari, cioè describili attraverso un DFA, ci dovrebbe invitare a pensare che tale enunciato sia falso, in quanto il DFA non ha potere espressivo sufficiente a calcolare $f$. Vogliamo quindi trovare un $B$ regolare e un $A$ non regolare tali che $A \leq_m B$. Prendiamo un classico esempio di linguaggio non regolare: $A = \{0^n 1^n \mid n \geq 0\}$. Dal momento che dobbiamo trovare un linguaggio a cui si possa ridurre $A$, ci conviene sceglierne uno ad esso abbastanza vicino, supponiamo $B = 0^*1^*$. Giunti a questo punto, il nostro obiettivo è dimostrare che $A \leq_m B$.
Definiamo la funzione $f$ come segue.

$$f(w) = \begin{cases} 01 & \text{se } w \text{ ha la forma } 0^n 1^n \\ 10 & \text{altrimenti} \end{cases}$$

La funzione mappa le stringhe $w$, che consideriamo elementi di $A$ quando esistono nella forma $0^n1^n$ o elementi esterni ad $A$ altrimenti, in stringhe presenti in $B$ nel primo caso ($01 \in B$) o assenti da $B$ nel secondo caso ($10 \notin B$).

**Esercizio 3.** Dimostrare la verità o la falsità di $A_{TM} \leq_m E_{TM}$.

Dimostriamo che $A_{TM} \not\leq_m E_{TM}$ per assurdo. Assumiamo per assurdo che $A_{TM} \leq_m E_{TM}$, allora si avrà anche che $\overline{A}_{TM} \leq_m \overline{E}_{TM}$. Ma questo è assurdo, perché abbiamo visto che $\overline{A}_{TM}$ non è Turing-riconoscibile, mentre $\overline{E}_{TM}$ lo è. Vale infatti che $A_{TM} \leq_m \overline{E}_{TM}$, e abbiamo appena dimostrato che $A_{TM} \not\leq_m E_{TM}$.

**Esercizio 4.** Dimostrare che se $A$ è Turing-riconoscibile e $A \leq_m \overline{A}$, allora $A$ è decidibile.

Sfruttiamo la definizione di mapping-riducibilità e il fatto che essa è insensibile al complemento. Poiché $A \leq_m \overline{A}$, otteniamo che $\overline{A} \leq_m \overline{\overline{A}} = A$. Poiché $\overline{A} \leq_m A$ e $A$ è Turing-riconoscibile, otteniamo che $\overline{A}$ è Turing-riconoscibile. Poiché infine sia $A$ che $\overline{A}$ sono Turing-riconoscibili, allora $A$ è decidibile.

**Esercizio 5.** Dimostrare che $A$ è Turing-riconoscibile se e solo se $A \leq_m A_{TM}$.

$\Leftarrow$) Assumiamo che $A \leq_m A_{TM}$. Dato che $A_{TM}$ è Turing-riconoscibile, deduco che $A$ è Turing-riconoscibile.

$\Rightarrow$) Assumiamo che $A$ sia Turing-riconoscibile. Poiché $A$ è Turing-riconoscibile, esiste una MdT $M$ tale che $L(M) = A$. Definisco $f(w) = < M, w >$.

$$w \in A \Rightarrow M \text{ accetta } w \Rightarrow < M, w > \in A_{TM}$$
$$w \notin A \Rightarrow M \text{ non accetta } w \Rightarrow < M, w > \notin A_{TM}$$

**Esercizio 6.** Dimostrare che $A$ è decidibile se e solo se $A \leq_m 0^*1^*$.

$\Leftarrow$) Assumiamo che $A \leq_m 0^*1^*$. Sappiamo che $0^*1^*$ è regolare, quindi esso è anche decidibile. Ma se è decidibile, allora deduco che anche $A$ è decidibile.

$\Rightarrow$) Assumiamo che $A$ sia decidibile. Poiché $A$ è decidibile, esiste un decisore $M$ tale che $L(M) = A$. Costruiamo $f$ come segue.

$$f(w) = \begin{cases} 01 & \text{se } M \text{ accetta } w \\ 10 & \text{altrimenti} \end{cases}$$

$f$ è calcolabile perché $M$ è un decisore.

$$w \in A \Rightarrow M \text{ accetta } w \Rightarrow f(w) = 01 \in B$$
$$w \notin A \Rightarrow M \text{ non accetta } w \Rightarrow f(w) = 10 \notin B$$

**Esercizio 7.** Si consideri il problema di determinare se una MdT con due nastri scrive un simbolo diverso da ␣ sul secondo nastro durante una computazione su un certo input $w$. Formalizzare tale problema come un linguaggio e dimostrarne l'indecidibilità.

$B = \{< M, w > \mid M \text{ è una MdT con due nastri che scrive } a \neq \text{␣ sul secondo nastro mentre computa } w\}$

In questo caso, non potremmo usare il Teorema di Rice perché questa non è una proprietà del linguaggio della Macchina di Turing, bensì è una proprietà della computazione della macchina.

Assumiamo per assurdo che $B$ sia decidibile, e sia $R$ il suo decisore. Costruiamo il seguente decisore per $A_{TM}$.

Su input $< M, w >$:

1. Costruisci una MdT a due nastri $T$ definita come segue:

   $T =$ Su input $x$:

   (a) Simula $M$ su $x$ usando il primo nastro.

   (b) Se $M$ accetta $x$, scrivi 0 sul secondo nastro.

2. Esegui $R$ con input $< T, w >$.

3. Ritorna il suo output.