Linguaggi e computabilità

spitfire

A.A. 2023-2024

Contents

1 Concetti generali della teoria degli automi

Definizione 1.0.1 (Alfabeto) *Con alfabeto si intende un insieme non vuoto di simboli. Esso si indica con* Σ *oppure* Γ.

Definizione 1.0.2 (Stringa) *Con stringa si identifica una sequenza finita di simboli di un alfabeto. Una stringa, tuttavia, non può essere costruita su più alfabeti.*

Definizione 1.0.3 (Stringa vuota) *Viene indicata con* ε *(o* λ*) la stringa vuota. Essa è una sequenza formata da 0 simboli dell'alfabeto*

Definizione 1.0.4 (Lunghezza di una stringa) *La lunghezza di una stringa a è il numero di caratteri da cui è composta. Essa si indica con* $|a|$.

Definizione 1.0.5 (Potenze di un alfabeto) *Sia* Σ *un alfabeto. Possiamo esprimere l'insieme di tutte le stringhe di una certa lunghezza k usando una notazione esponenziale. Definiamo quindi con* Σ^k *come l'insieme delle stringhe di lunghezza k costruite a partire dall'alfabeto* Σ. *In particolare, abbiamo:*

$$\Sigma^0 = \{\varepsilon\}$$

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup ... = \bigcup_{k\geq 0}^{+\infty} \Sigma^k$$

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup ... = \bigcup_{k=1}^{+\infty} \Sigma^k = \Sigma^* \backslash \{\varepsilon\}$$

Definizione 1.0.6 (Concatenazione di stringhe) *Siano x e y delle stringhe. Allora la loro concatenazione xy è data facendo seguire ad una copia di x una copia di y. Essa si indica anche con il simbolo* $\cdot$

Osservazione 1.0.1 $|x \cdot y| = |x| + |y|$

Osservazione 1.0.2 $|x \cdot \varepsilon| = |\varepsilon \cdot x| = |x|$

Osservazione 1.0.3 $(\Sigma^*, \cdot)$ *è un* ***monoide*** *libero con elemento neutro* ε. *Cioè è un semigruppo dotato di base per i suoi elementi.*

Definizione 1.0.7 (Linguaggio) *Dato un alfabeto* Σ, *un linguaggio L su* Σ *è un sottoinsieme di* Σ^*: $L \subseteq \Sigma*$.

Osservazione 1.0.4 *Un linguaggio può contenere un numero infinito di stringhe. Per descrivere tali linguaggio si può usare questa notazione:*

$$L_A = \{x \in \Sigma^* | A(x) = Si\}$$

Con A l'automa che riconosce se se una stringa in input fa parte del linguaggio o meno

Definizione 1.0.8 (Mebership problem) *Dato $L \subseteq \Sigma^*$ e data una strina $x \in \Sigma^*$, $x \in L$?. Di solito il problema viene risolto o tramite il metodo dell'**automa** o oppure si utilizza una grammatica per il Linguaggio L; quindi $x \in L \Leftrightarrow x$ si può derivare dalle regole della grammatica. Il problema è analogo ad un **problema di decisione**; in particolare esso è un **problema di decisione universale**, poiché può essere rappresentato in molteplici modi.*

2 Linguaggi e grammatiche

2.1 Linguaggi Context-Free

I linguaggi Context-Free possono essere utilizzati per definire la **sintassi** dei linguaggi di programmazione e/o di rappresentazione (HTML, XML, ...). Essi hanno una **struttura ricorsiva** e hanno principalmente due regole di composizione:

1. **Concatenazione**: Un linguaggi L è composto da due linguaggi, L_1 e L_2 concatenati l'uno dietro l'altro
2. **Linguaggio contenuto in un altro**: L è un linguaggio che contiene un linguaggio L_1 contenuto all'interno delle stringhe di un altro linguaggio L_2.

2.2 Grammatiche e produzioni

Definizione 2.2.1 *Una Grammatica G è una quadrupla $G = (V, T, P, S)$ dove:*

- V = *Insieme delle variabli*
- T = *Insieme dei simboli terminali*
- P = *Insieme delle regole di produzione*
- S = *"Starting symbol"* $\in V$

In particolare, le regole di produzione hanno questa sintassi: supponendo l'esistenza in V di una variabile X

$$X \to Prod_1 | Prod_2 | ... | Prod_n$$

Mentre una particolare derivazione ha questa sintassi:

$$X \Rightarrow Prod_1 \Rightarrow ...$$

Osservazione 2.2.1 *Il processo di derivazione **non è controllato**; cioè la grammatica, applicando ogni combinazione delle regole di produzione, **produce ogni stringa possibile***

Osservazione 2.2.2 *In ogni passo di derivazione, la grammatica pratica **al più una sostituzione***

2.3 Tipi di Linguaggi e grammatiche

Le grammatiche si differenziano fra loro per la struttura delle regole di produzione. **Noam Chomsky** teorizzò per primo la **gerarchia dei linguaggi**:

$$Tipo\ 0 \supseteq Tipo\ 1 \supseteq Tipo\ 2 \supseteq Tipo\ 3$$

I linguaggi quindi si dividono in 4 possibili tipi e si differenziano anche sulla struttura degli automi che accettano i linguaggi di un certo tipo rispetto ad un altro:

- **Tipo 0: Linguaggi ricorsivamente enumerabili**: In questi linguaggi le regole di produzione possono essere solo del tipo

  $$\alpha \to \beta\ con\ \alpha, \beta \in (V \cup T)^*$$

  Gli automi che accettano questo tipo di linguaggi sono le **Macchine di Turing**

- **Tipo 1: Linguaggi contestuali o "dipendenti dal contesto"**: In questi linguaggi le regole di produzione sono del tipo:

  $$\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2\ con\ \alpha_1, \alpha_2, \beta \in (V \cup T)^*, \beta \neq \varepsilon\ e\ A \in V$$

  α_1, α_2 sono il **contesto** della variabile A; cioè la sostituzione avviene solamente se A è tra α_1, α_2. Gli automi che accettano questi linguaggi sono le **Macchine di Turing che lavorano in spazio lineare**

- **Tipo 2: Linguaggi Context-Free o "liberi dal contesto"**: In questi linguaggi, le regole di produzione sono del tipo

  $$A \to \beta\ dove\ A \in V\ e\ \beta \in (V \cup T)^*$$

  Gli auotomi che accettano questi linguaggi sono gli **Automi a pila non deterministici**

  Osservazione 2.3.1 *Le regole di tipo 2 sono un caso particolare delle regole di tipo 1*

- **Tipo 3: Linguaggi regolari**: Le regole di produzione di questi linguaggi sono del tipo:

  $$A \to aB\ oppure A\ \to a\ dove A, B \in V\ e\ a \in T$$

  Gli automi che accettano questi linguaggi sono gli **Automi a stati finiti deterministici, non-deterministici e non-deterministici con ε-transizioni**

Proposizione 2.3.1 *Le **variabili** sono anche dette **categorie sintattiche**; se vi sono 2 regole del tipo:*

$$A \to B$$

$$B \to C|D|...$$

*Con B che non permette di tornare ad A; allora B può essere considerata come una **sottogrammatica***

2.4 Metodo dell'inferenza ricorsiva

Un altro metodo per derivare una stringa da una grammatica è quello dell'**inferenza ricosrsiva**:

1. Si assegna ad ogni regola di produzione un numero
2. Si costruisce una tabella con campi rispettivamente n° passo, stringa derivata, variabile usata, n° della produzione usata, stringhe pregresse usate (identificate con il n° di passo)
3. si riempe la tabella derivando la stringa deisderata

Osservazione 2.4.1 *Utilizzare direttamente una derivazione o usare il metodo dell* ***inferenza ricorsiva*** *è equivalente*

2.5 Derivazioni

Definizione 2.5.1 ($\Rightarrow$) *Sia $G = (V, T, P, S)$ una grammatica libera dal contesto (CFG da questo momento in poi). Sia $\alpha A \beta$ t.c. $\alpha \in (V \cup T)^*, A \in V$. Sia $A \to \gamma$ una regola di produzine con $\gamma \in (V \cup T)^*$. Allora scriviamo: $\alpha A \beta \Rightarrow \alpha \gamma \beta$.*

Definizione 2.5.2 ($\Rightarrow^*$) *La definizione di derivazione in "zero o più passi" è induttiva:*

- ***Base:*** *$\forall \alpha \in (V \cup T)^*, \alpha \Rightarrow^* \alpha$*
- ***Passo induttivo:*** *Se $\alpha \Rightarrow^* \beta$ e $\beta \Rightarrow \gamma$ allora $\alpha \Rightarrow^* \gamma$*

Definizione 2.5.3 ($\Rightarrow^*$ alternativa) *$\alpha \Rightarrow^* \beta$ se $\exists \gamma_1, \gamma_2, ..., \gamma_n$, con $n \geq 1$ t.c. $\alpha = \gamma_1, \beta = \gamma_2$ e $\forall i = 1, 2, ..., n-1$ vale che $\gamma_i \Rightarrow \gamma_{i+1}$*

Notazione 2.5.1 *Se si opera su una grammatica G oppure vi è ambiguità, allora si scrive $\Rightarrow^*_G$*

Definizione 2.5.4 ($\Rightarrow_{lm}$) *Una derivazione è "left-most" quando si sostituisce* ***sempre*** *la variabile più a sinistra*

Definizione 2.5.5 ($\Rightarrow_{rm}$) *Una derivazione è "right-most" quando si sostituisce* ***sempre*** *la variabile più a destra*

Teorema 2.5.1 *$A \Rightarrow^* w$ (con $A \in V$ e $w \in (V \cup T)^*$) $\Leftrightarrow$ si può derivare $A \Rightarrow^*_{lm} w$ e $A \Rightarrow^*_{rm} w$*

2.6 Linguaggio generato da una CFG

Data una CFG $G = (V, T, P, S)$, il linguaggio generato da G è

$$L(G) = \{w \in T^* | S \Rightarrow^* w\}$$

Definizione 2.6.1 (Forma sentenziale) *Data una $G = (V, T, P, S)$, una forma sentenziale è una stringa $\alpha \in (V \cup T)^*$ t.c. $S \Rightarrow^* \alpha$*

2.7 Linguaggi di tipo 1: Linguaggi contestuali

Vediamo alcuni esempi di linguaggi contestuali:
1° Esempio

$$L = \{a^n b^n c^n | n \geq 1\}$$
$$G = \big\{\{S, B, C, X\}, \{a, b, c\}, P, S\big\}$$

P:

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

Supponiamo di avere (segnamo in **grassetto** il contesto di ogni variabile):

$$...C\mathbf{B}... \Rightarrow_3 \mathbf{X}B \Rightarrow_4 X\mathbf{C} \Rightarrow_5 BC$$

cioè $CB \Rightarrow^* BC$ Per ottenere quindi una qualsiasi stringa, in generale:

$$S \Rightarrow_1^{n-1} a^{n-1}S(BC)^{n-1} \Rightarrow_2 a^n(BC)^n \Rightarrow_{3,4,5}^{n(n+1)/2} a^n b^n c^n \Rightarrow_6 a^b B^{n-1} C^n \Rightarrow_7^{n-1} a^n b^n C^n$$
$$\Rightarrow_8 a^n b^n c C^{n-1} \Rightarrow_9^{n-1} a^n b^n c^n$$

Perchè $\Rightarrow_{3,4,5}$ deve essere applicato $n(n+1)/2$ volte?

Dimostrazione 2.7.1 *Procediamo per induzione su n:*

- ***Base (n=1)*** $BC \Rightarrow^{1(1-1)/2} BC$
- ***Passo induttivo****: Supponiamo* $(BN)^n \Rightarrow_{3,4,5}^{n(n+1)/2} B^n C^n$ *vero e dimostriamo che vale anche per n+1:*

  $$(BC)^{n+1} = (BC)^n BC \Rightarrow_{3,4,5}^{n(n-1)/2} B^n C^n BC \Rightarrow_{3,4,5}^{n} B^{n+1} C^{n+1}$$

  Il numero di applicazioni è quindi $n(n-1)/2 + n = n(n+1)/2$

□

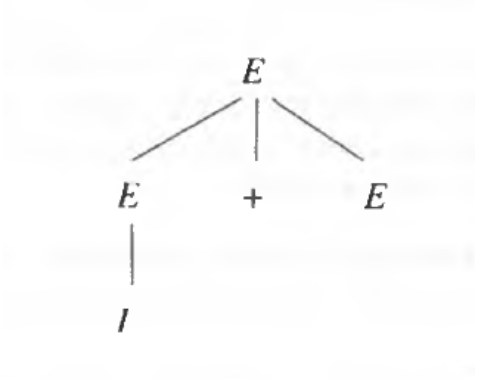
2° esempio:

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

Il linguaggio è evidentemente di tipo 1, visto che presenta un "accavallamento" tra n e m che non permetterebbe di esprimerne una grammatica seguendo solo regole di tipo 2

2.8 Alberi sintattici

Un **albero sintattico** è una struttura ad albero che rappresenta come una certa sequenza di caratteri è stata ottenuta; quest'ultima è indicata come il **prodotto dell'albero**.



Definizione 2.8.1 (Albero sintattico) *Data una grammatica $G = (V, T, P, S)$, allora un albero sintattico per G è tale che:*

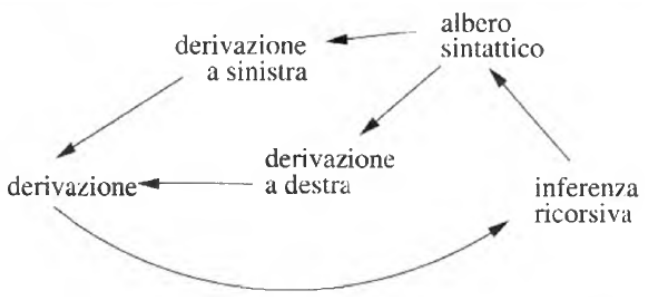
- *Ogni nodo interno è etichettato da una variabile*
- *Ogni foglia è etichettata da una variabile, da un simbolo terminale oppure da ε. Se una foglia è etichettata da ε, deve essere l'unico figlio del suo genitore*
- *Se un nodo interno è eitchettato da A e i suoi figli sono etichettati, a partire da sinistra, con $x_1, x_2, ..., x_k$, allora $A \to x_1x_2...x_k$ è una regola di produzione di P.*
- *Gli alberi sintattici sono **ordinati***

Osservazione 2.8.1 *Se un sottoalbero si trova a sinistra/destra di un altro sottoalbero, allora la formula indicata da esso si trova a sinistra/destra della formula indicata dall'altro sottoalbero*

Teorema 2.8.1 *Data una CFG G e un linguaggio A, i seguenti enunciati si equivalgono:*

- *L'inferenza ricorsiva stabilisce che la stringa terminale w è nel linguaggio delal variabile $A \Leftrightarrow$*
- $A \Rightarrow^* w \Leftrightarrow$
- $A \Rightarrow^*_{lm} w \Leftrightarrow$
- $A \Rightarrow^*_{rm} w \Leftrightarrow$
- *Esiste un albero sintattico con radice A e prodotto w.*

Dimostrazione 2.8.1 *Questa equivalenza è dimostrabile secondo il seguente schema:*



Ogni arco del grafo denota un teorema secondo il quale, se w soddisfa la condizione nella ***coda dell'arco****, la soddisfa anche nella* ***testa dell'arco****. Per esempio, se si deduce che w è nel linguaggio per* ***inferenza ricorsiva****, allora esiste anche un albero sintattico T con radice A e prodotto w.*

2.9 Ambiguità

Una stringa w di terminali può essere **ambigua**; cioe può avere **più di un albero sintattico** da cui è possibile derivarla. L'ugaglianza degli alberi quindi crea problemi ad un potenziale **parser sintattico** per il linguaggio. L'ideale sarebbe quindi che le grammatiche **non fossero ambigue**, tuttavia ci sono delle **cattive notizie**:

- In generale, non esiste un algoritmo che determina se una CFG G è ambigua o meno
- Data una CFG G che **sappiamo** essere ambigua, non esiste un algoritmo che ci consente di trasformarla in una CFG G' **non ambigua**
- Esistono **linguaggi ineritamente ambigui**, cioè per i quali non esiste nessuna CFG **non** ambigua.

2.9.1 Ambiguità e left-most derivation

Teorema 2.9.1 $\forall CFG\ G = (V, T, P, S)$ *e* $\forall w \in T^*$*, abbiamo che w ha 2 alberi sintattici distinti* $\Leftrightarrow$ *ha 2 derivazioni sinistre distinte.*

2.9.2 Ambiguità inerente

Definizione 2.9.1 (Linguaggio inerentemente ambiguo) *Un linguaggio si dice inerentemente ambiguo se esso* ***non ammette grammatiche non ambigue****; cioè tutte le sue grammatiche sono ambigue*

2.10 Grammatiche regolari

Definizione 2.10.1 (Grammatica regolare) *Sia $G = (V, T, P, S)$. Se:*

1. *ε può comparire solo nella regola $S \to \varepsilon$, dove S è lo "starting symbol"*
2. *Le regole di produzione P sono **tutte** lineari a destra oppure **tutte** lineari a sinistra; cioè se*
   - ***lineare a dx:** $A \to aB$ oppure $A \to a$ dove $A, B \in V$ e $a \in T$*
   - ***lineare a sx:** $A \to Ba$ oppure $A \to a$ dove $A, B \in V$ e $a \in T$*

*Allora la grammatica è **regoalre***

Proposizione 2.10.1 *Usando solo le regoli lineari a sx/dx, ottengo **tutti i linguaggi regolari**. In particolare, posso trasformare un linguaggio che usa una delle 2 nell'alternativa*

2.11 Operazioni su linguaggi

Vi sono principalmente tre operazioni possibili sui linguaggi:

- **Unione**: Dati due linguaggi L e M su un certo alfabeto Σ, allora la loro unione $L \cup M$ è l'insieme delle stringhe appartenenti ad L e M, prese una sola volta
- **Concatenazione**: Dati L e M, abbiamo che la loro concatenazione $LM = \{w = w_1 w_2 | w_1 \in L, w_2 \in M\}$. In particolare, l'insieme LM contiene tutte le stringhe che si possono formare prendendo una qualsiasi stringa di L e concatenandola con una qualsiasi stringa di M. La cardinalità di LM è $|LM| = |L| \cdot |M|$; inoltre $|L \cup M| \leq |L| + |M|$
- **Chiusura (o "star") di Kleene**: La chiusura di Kleene di un linguaggio L viene indicata con L^* e rappresenta l'insieme delle stringhe che possono essere formate prendendo il linguaggio L e **concatenandolo** con se stesso infinite volte. Formalmente quindi:

$$L^* = \bigcup_{i=0}^{+\infty} L^i$$

$$L^i = \underbrace{L \cdot L \cdot L \cdot L ... \cdot L}_{i-volte}$$

In particolare:

$$L^1 = L$$

$$L^0 = \{\varepsilon\}$$

Osservazione 2.11.1 *Se $|L| = q$, allora $|L^i| = q^i$*

Proposizione 2.11.1 *La stella di kleene è definibile anche su linguaggi infiniti*

Vi sono ancora due casi particolari:

$$L = \emptyset \Rightarrow L^0 = \emptyset^0 = \{\varepsilon\} \; mentre \; L^i = \underbrace{\emptyset \cdot \emptyset \cdot ... \cdot \emptyset}_{i-volte} = \emptyset$$

$$L = \{\varepsilon\} \Rightarrow L^0 = \{\varepsilon\}^0 = \{\varepsilon\} mentre \; L^i = \underbrace{\{\varepsilon\} \cdot \{\varepsilon\} \cdot ... \cdot \{\varepsilon\}}_{i-volte} = \{\varepsilon\}$$

Quanto vale quindi la stella di kleene in questi due casi?

$$L = \emptyset \Rightarrow L^* = \{\varepsilon\}$$

$$L = \{\varepsilon\} \Rightarrow L^* = \{\varepsilon\}$$

Essi sono gli unici 2 casi in cui la chiusura di Kleene è un insieme finito.

2.12 Espressioni Regolari (ER)

Le espressioni regolari si usano per rappresentare (in termini tecnici, **denotare**) i linguaggi regolari.

Definizione 2.12.1 (Espressioni regolari) *La definizione procede per induzione:*

- ***Base**: ε e $\emptyset$ sono **espressioni regolari**. Questi due simboli, tuttavia non denotano la stringa vuota o l'insieme vuoto, ma **i linguaggi denotati dall'insieme vuoto e da** ε, i quali sono*

$$L(\varepsilon) = \{\varepsilon\}$$

$$L(\emptyset) = \emptyset$$

Se $a \in \Sigma$ (vogliamo denotare un linguaggio $L \subseteq \Sigma^*$, cioè un linguaggio costituito con i simboli di Σ) allora $\boldsymbol{a}$ è un espressione regolare che denota il linguaggio

$$L(\boldsymbol{a}) = \{a\}$$

*Le variabili che rappresentano i linguaggi (es. L) **sono espressioni regolari**.*

- ***Passo induttivo***

  1. ***Unione**: Se E ed F sono espressioni regolari, allora $E + F$ è un espressione regolare.*

  $$L(E + F) = L(E) \cup L(F)$$

  2. ***Concatenazione**: Se E e F sono espressioni regolari, allora EF è un espressione regolare:*

  $$L(EF) = L(E)L(F)$$

  3. ***Chiusura**: Se E è un espressione regolare, allora E^* è una espressione regolare:*

  $$L(E^*) = (L(E))^*$$

  4. ***Parentesi**: Se E è un espressione regolare, allora (E) è un espressione regolare*

  $$L((E)) = L(E)$$

2.12.1 Proprietà algebriche delle ER

- * si applica alla **più piccola sequenza di simboli a sinistra ben formata**.
- **Concatenazione**: Essa ha le seguenti proprietà:
  - **Associativa**: $E \cdot (G \cdot F) = (E \cdot G) \cdot F = E \cdot G \cdot F$
  - **Non** è commutativa: $EF \neq FE$
- **Unione**: Essa ha le seguenti proprietà:
  - **Associativa**: $E + (F + G) = (E + F) + G = E + F + G$
  - **Commutativa**: $E + F = F + E$
- $\emptyset$ è **l'identità** per l'unione: $\emptyset + L = L + \emptyset = L$
- ε è **l'identità** per la concatenazione: $\varepsilon \cdot L = L \cdot \varepsilon = L$
- $\emptyset$ è **annichilatore** per la concatenazione: $\emptyset \cdot L = \emptyset$
- **Distributività sinistra della concatenazione sull'unione**:

$$L(M + N) = LM + LN$$

- **Distributività destra della concatenazione sull'unione**:

$$(M + N)L = ML + NL$$

- **Idempotenza dell'unione**: $L + L = L$

3 Automi a stati finiti

Vi sono 3 forme principali di automa a stati finiti:

- Automi a stati finiti **deterministici** (DFA)
- Automi a stati finiti **non deterministici** (NFA)
- Automi a stati finiti **non deterministici con ε-transizioni**(ε-NFA)

Tutti questi automi accettano **tutti e soli** i linguaggi regolari. Cioè: dato un linguaggio regolare L, esiste un automa a stati finiti A_L che accetta ogni sua stringa.

3.1 Automi a stati finiti deterministici (DFA)

Definizione 3.1.1 (Automa a stati finiti deterministico (DFA)) *Un DFA A è una quintupla:*

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

Dove:

- Q *è l'insieme finito e non vuoto degli stati*
- Σ *è l'alfabeto dei simboli in ingresso*
- δ *è la funzione di transizione degli stati*
- $q_0 \in Q$ *è lo stato iniziale*
- $F \subseteq Q$ *è l'insieme degli stati finali (o accettanti)*

Definizione 3.1.2 (Funzione di transizione per DFA) *La funzione di transizione δ è definita come segue:*

$$\delta : Q \times \Sigma \to Q$$

La funzione ***deve essere totale****. In particolare, se $|Q| = n$ e $|\Sigma| = m$, allora il numero degli elementi di δ è $n \cdot m$.*

3.1.1 Tabella di transizione

Una tabella di transizione è una modalità di rappresentazione della funzione δ. Le righe della tabella corrispondono agli stati, le colonne agli input. La voce all'incrocio della riga corrispondente allo stato q e della colonna corrispondente all'input a è lo stato $\delta(q, a)$

	0	1
$\to q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

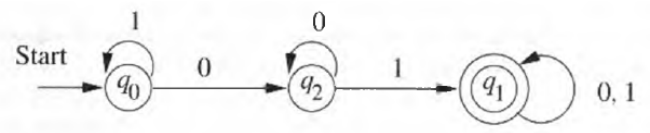
In particolare, si indica con una $\to$ lo stato iniziale; mentre con un * gli stati finali.

3.1.2 Diagramma di transizione

Per un DFA, un diagramma di transizione è composto nel seguente modo:

- Per ogni stato in Q esiste un **nodo**
- Per ogni stato q in Q e ogni simbolo di input $a \in \Sigma$, sia $\delta(q, a) = p$. Allora il grafo ha un arco dal nodo q al nodo p etichettato con a. Se esistono più simboli di input che portano da q a p, allora l'arco può essere etichettato dalla lista di tali simboli

- Una freccia etichettata **START** nello stato iniziale q_0. Tale freccia non proviene da nessuno nodo
- I nodi corrispondenti agli stati finali sono segnati con un doppio circolo.



3.2 Funzione $\hat{\delta}$ per DFA

Precisiamo la nozione di **linguaggio di un DFA**. Per farlo, dobbiamo quindi estendere la definizone di δ per adattarla a quando partiamo da uno stato q ed effettuiamo una serie di input; chiamiamo questa funzione estesa $\hat{\delta}$:

$$\hat{\delta} : Q \times \Sigma^* \to Q$$

Definizione 3.2.1 ($\hat{\delta}$) *La definizione procede per induzione:*

- ***Base:*** *Se* $|w| = 0$, *allora* $\hat{\delta}(q, w) = \hat{\delta}(q, \varepsilon) = q$
- ***Passo induttivo:*** *Se* $|w| > 0$, *allora* $w = ax$, *dove* $a \in \Sigma$ *e* $x \in \Sigma^*$. *Allora* $\hat{\delta}(q, w) = \hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x)$

Definizione 3.2.2 ($\hat{\delta}$ alternativa) *La definizione procede per induzione:*

- ***Base:*** *Se* $|w| = 0$, *allora* $\hat{\delta}(q, w) = \hat{\delta}(q, \varepsilon) = q$
- ***Passo induttivo:*** *Se* $|w| > 0$, *allora* $w = xa$, *dove* $a \in \Sigma$ *e* $x \in \Sigma^*$. *Allora* $\hat{\delta}(q, w) = \hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$

3.3 Linguaggio accettato da un DFA

Definizione 3.3.1 (Linguaggio accettato da un DFA) *Il linguaggio accettato da un DFA A è definito come:*

$$L(A) = \{w \in \Sigma^* | \hat{\delta}(q_0, w) \in F\}$$

3.4 Automa a stati finiti non determinisstico (NFA)

Definizione 3.4.1 (Automa a stati finiti non determinisstico (NFA)) *Un NFA è una quintupla*

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

dove:

- Q *è l'insieme finito e non vuoto degli stati*
- Σ *è l'alfabeto dei simboli in ingresso*
- δ *è la funzione di transizione degli stati*

- $q_0 \in Q$ *è lo stato iniziale*
- $F \subseteq Q$ *è l'insieme degli stati finali (o accettanti)*

Definizione 3.4.2 (Funzione di transizione per NFA) *La funzione di transizione δ per NFA è definita come:*

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

Con 2^Q l'insieme delle parti di Q. In particolare, se $|Q| = n$ allora $|2^Q| = 2^{|Q|}$. La funzione delta ***può anche non essere totale****.*

3.4.1 Tabella degli stati per un NFA

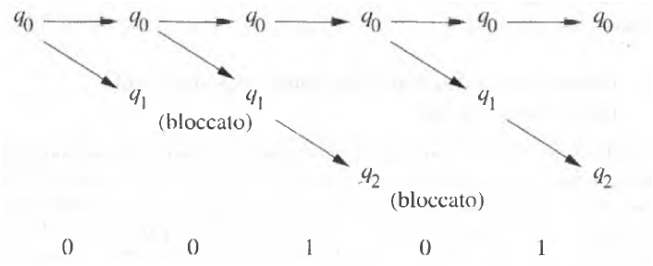
	0	1
$\to q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

La differenza sostanziale con la tabella per DFA è la funzione delta **ritorna un insieme di stati in cui l'NFA potrebbe transitare**.

Osservazione 3.4.1 *Anche con un solo stato, l'output di δ per un NFA è sempre un insieme di stati*

Osservazione 3.4.2 *Ogni DFA può essere visto come caso particolare di NFA*

3.4.2 Interpretazione della computazione di un NFA



Si può immaginare la computazione dell'NFA come una serie di **"processi paralleli"**: Partendo dallo stato iniziale, viene letto il primo carattere, l'automa può passare in una moltitudine di stati; immaginiamo quindi che facce **entrambe le cose contemporaneamente**; un processo quindi **"muore"** quando viene letto un carattere ma la funzione di transizione non è definita per quella particolare coppia Stato-Valore. La stringa è accetatta se almeno un processo è arrivato ad uno stato accettante.

3.5 Linguaggio definito da un NFA

$$L(N) = \{w \in \Sigma^* | \hat{\delta}_n(q_o, w) \cap F \neq \emptyset\}$$

Tuttavia, come fa un NFA a capire quale strada intraprendere o se ha finito la computazione? L'automa NFA può chiedere ad un **oracolo**, il quale "prevede il futuro" e sa qual'è il percorso corretto. Si può simulare questo oracolo? Per gli NFA, esso è simulabile tramie un **DFA** che accetta lo stesso linguaggio dell'NFA. Un problema però, se $|Q| = n$, allora $|2^Q| = 2^{|Q|} = 2^n$; poiché il DFA scandisce il diagramma dell'NFA "a blocchi" (ogni stato del DFA è un insieme di stati dell'NFA), c'è il rischio che si abbia un esplosione nel numero di stati da rappresentare

3.6 $\hat{\delta}$ per NFA

Definizione 3.6.1 ($\hat{\delta}$) *La definizione procede per induzione:*

- ***Base:*** *Se* $|w| = 0$, *allora* $\hat{\delta}(q, w) = \hat{\delta}(q, \varepsilon) = \{q\}$
- ***Passo induttivo:*** *Se* $|w| > 0$, *allora* $w = ax$ *con* $a \in \Sigma$ *e* $w \in \Sigma^*$*:*

$$\delta(q, a) = \{p_1, p_2, ..., p_k\}$$

$$\hat{\delta}(q_i, w) = \bigcup_{i=1}^{k} \hat{\delta}(p_i, x) = \{R_1, ..., R_K\}$$

Definizione 3.6.2 ($\hat{\delta}$ **alternativa**) *La definizione procede per induzione:*

- ***Base:*** *Se* $|w| = 0$, *allora* $\hat{\delta}(q, w) = \hat{\delta}(q, \varepsilon) = \{q\}$
- ***Passo induttivo:*** *Se* $|w| > 0$, *allora* $w = xa$ *con* $a \in \Sigma$ *e* $w \in \Sigma^*$*:*

$$\delta(q, x) = \{p_1, p_2, ..., p_k\}$$

$$\hat{\delta}(q_i, w) = \bigcup_{i=1}^{k} \delta(p_i, a) = \{R_1, ..., R_K\}$$

3.7 Algoritmo di conversione NFA a DFA equivalente

Per costruire il DFA $D = (Q_D, \Sigma, \delta_D\{q_s\}, F_D)$ a partire dall'NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N\}$ t.c $L(N) = L(D)$ partiamo dalle seguenti definizioni e adottiamo una tecnica chiamata "**costruzione per sottoinsiemi**":

- Q_D è l'insieme potenza di Q_N; quindi $Q_D = 2^{Q_N}$. Seppur la cardinalità di $|2^{Q_N}| = 2^{|Q_N|}$, non sempre si riescon a raggiungere tutti gli stati; quindi nel DFA gli stati potrebbero essere anche minori di $2^{|Q_N|}$
- $F_D = \{S \subseteq Q_N | S \cap F_N \neq \emptyset\}$, cioè F_D è formato da tutti i sottoinsiemi di Q_N che contengono almeno uno stato accettante

- Per ogni $S \subseteq Q_N$ e per ogni simbolo di input $a \in \Sigma$:

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

  Cioè, per computare $\delta_D(S, a)$ consideriamo tutti gli stati di S, rileviamo in quali insiemi di stati va a finire l'automa partendo da p e leggendo a e infine prendiamo l'unione di tutti questi sottoinsiemi.

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

Nominiamo infine ogni stato con una lettera e costruiamo il DFA equivalente.

Nota 3.7.1 *Ogni transizione che porta all'insieme vuoto risulta in un* ***nodo pozzo***

3.8 ε-NFA

Gli ε-NFA sono particolari tipi di NFA che presentano le "ε-transizioni", cioè transizioni ad altri stati che **non comportano il consumo di un simbolo**. Quando una ε-transizione viene eseguita dall'automa, si dice che esso ha fatto una ε-mossa

Definizione 3.8.1 (ε-NFA) *Un ε-NFA è una quintupla*

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

Dove:

- *Q è l'insieme finito e non vuoto degli stati*
- *Σ è l'alfabeto dei simboli in ingresso*
- *δ è la funzione di transizione degli stati*
- *$q_0 \in Q$ è lo stato iniziale*
- *$F \subseteq Q$ è l'insieme degli stati finali (o accettanti)*

Definizione 3.8.2 (Funzione di transizione per ε-NFA) *La funzione di transizione δ per ε-NFA è definita nel seguente modo:*

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$$

3.9 Eclose

Sia $q \in Q$. Allora $Eclose(q)$ è **l'insieme** degli stati raggiungibili tramite ε-mosse (o 0 mosse). Essa è quindi una funzione definita come segue:

$$Eclose : Q \to 2^Q$$

Definizione 3.9.1 *La definizione procede per induzione:*

- ***Base:*** $q \in Eclose(q)$
- ***Passo induttivo:*** *Se* $p \in Eclose(q)$ *ed esiste una transizione che porta da* p *ad* r *tramite una* ε*-mossa, allora* $r \in Eclose(q)$

3.10 $\hat{\delta}$ per ε-NFA

Definizione 3.10.1 ($\hat{\delta}$) *La definizione procede per induzione:*

- ***Base:*** *Se* $|w| = 0$*, allora* $w = \varepsilon$*; quindi* $\hat{\delta}(q, w) = \hat{\delta}(q, \varepsilon) = Eclose(q)$
- ***Passo induttivo*** *Se* $|w| > 0$*, allora* $w = ax$ *con* $a \in \Sigma$ *e* $x \in \Sigma^*$*. Posto* $Eclose(q) = \{p_1, ..., p_k\}$*,* $eclose(\bigcup_{i=1}^{k} \delta(p_i, a)) = \{R_1, ..., R_m\}$ *allora vale*

$$\hat{\delta}(q, ax) = \bigcup_{j=1}^{m} \hat{\delta}(R_j, x)$$

Cerchiamo di semplificare la definizione introducendo le seguenti definizioni (sia S l'insieme degli stati):

- $Eclose(S) = \bigcup_{q \in S} Eclose(q)$
- $\delta(S, a) = \bigcup_{q \in S} \delta(q, a)$
- $\hat{\delta}(S, w) = \bigcup_{q \in S} \hat{\delta}(q, w)$

Definiamo anche i casi particolari:

- $Eclose(\emptyset) = \emptyset$
- $\delta(\emptyset, a) = \emptyset; \forall a \in \Sigma$
- $\hat{\delta}(\emptyset, w) = \emptyset \ \forall w \in \Sigma^*$

Definizione 3.10.2 ($\hat{\delta}$ **semplificata**) *La definizione procede per induzione:*

- ***Base:*** *Se* $|w| = 0$*, allora* $w = \varepsilon$*; quindi* $\hat{\delta}(q, w) = \hat{\delta}(q, \varepsilon) = Eclose(q)$
- ***Passo induttivo:*** *Se* $|w| > 0$*, allora* $w = ax$*; con* $a \in \Sigma$ *e* $x \in \Sigma^*$*. Posto:* $P = Eclose(q)$*,* $S = \delta(P, a) = \bigcup_{p \in P} \delta(p, a)$ *e* $R = Eclose(S) = \bigcup_{s \in S} Eclose(s)$*, allora vale*

$$\hat{\delta}(q, ax) = \hat{\delta}(R, x)$$

Osservazione 3.10.1 *Togliendo le Eclose, si ottiene la definizione per la* $\hat{\delta}$ *degli NFA.*

3.11 Algoritmo di trasformazione da ε-NFA a DFA

Dato un ε-NFA $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ costruiamo un DFA $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$ equivalente ad E, cioè $L(E) = L(D)$. Utilizziamo il metodo **della costruzione per sottoinsiemi**

- $Q_D = 2^{Q_E}$, inoltre tutti gli stati raggiungibili del nostro DFA corrispondono a insiemi S di stati dell' ε-NFA che sono **ε-chiusi**, cioè $Eclose(S) = S$.
- $q_D = Eclose(q_0)$
- $F_D = \{S \in Q_D | S \cap F_E \neq \emptyset\}$
- $\forall a \in \Sigma$ e $\forall S = \{p_1, ..., p_k\} \in Q_D$ allora $\delta_D(S, a)$ si ottiene:
  1. Calcolando $\bigcup_{i=1}^{k} \delta_E(P, a) = \{R_1, ..., R_M\}$
  2. Ponendo $\delta_D(S, a) = Eclose(\{R_1, ..., R_M\})$

3.12 Automa prodotto

Consideriamo un DFA $A_L = (Q_L, \Sigma, \delta, q_L, F_L)$ tale che $L(A_L) = L$ e consideriamo un DFA $A_M = (Q_M, \Sigma, \delta, q_M, F_M)$. Costruiamo l'automa prodotto $A_L \otimes A_M = A$. A ha stati del tipo (p, q) dove $p \in Q_L$ e $q \in Q_M$. A si trova in ogni istante in uno stato (p, q) e quando legge un simbolo $a \in \Sigma$ va nello stato $(\delta_L(p, a), \delta_M(q, a))$

Definizione 3.12.1 *L'automa prodotto è una quintupla:*

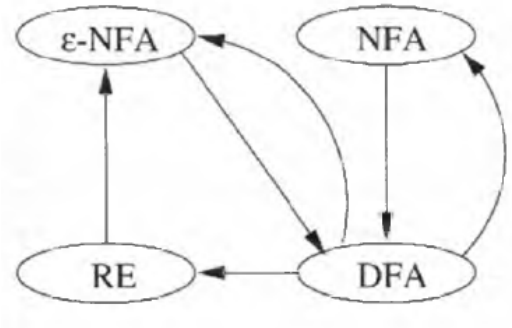
$$A = (\underbrace{Q_L \times Q_M}_{Q}, \Sigma, \delta, \underbrace{(q_L, q_M)}_{q_0}, \underbrace{F_L \times F_M}_{F})$$

Con funzione δ di transizione di questo tipo:

$$\delta((p, q), a) = ((\delta_L(p, a), \delta_M(q, a)))$$

4 Automi a stati finiti ed espressioni regolari

4.1 Da espressione regolare a ε-NFA



Mostriamo ora come passare da un'espressione regolare R ad un ε-NFA E. In particolare, quindi, mostriamo che ogni linguaggio L che sia uguale a $L(R)$ per un'espressione regolare R è anche uguale a $L(E)$ per un ε-NFA E. La dimostrazione avviene per **induzione strutturale sulla struttura di R**:

Teorema 4.1.1 *Ogni linguaggio definito da un'espressione regolare è definito anche da un automa a stati finiti*

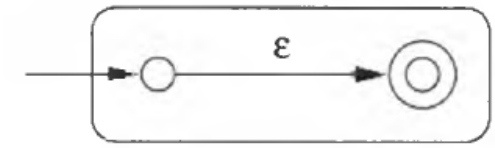
Dimostrazione 4.1.1 *Supponiamo che $L = L(R)$ per un'espressione regolare R. Mostriamo anche che $L = L(E)$ per un ε-NFA E per induzione con dei "moduli" costruiti nel seguente modo:*

1. *Uno stato accetante*
2. *Nessun arco entrante nello stato iniziale*
3. *Nessun arco uscente dallo stato finale*

*Procediamo per induzione sulla struttura di R utilizzando la definizione ricorsiva delle espressioni regolari: (gli archi entranti nel "primo" stato dei moduli **NON** indicano stati iniziali, ma transizioni da stati "precedenti", potenzialmente dallo stato iniziale dell'automa complessivo)*

***Casi base**:*

1. *Se $R = \varepsilon$ (quindi $L(R) = \{\varepsilon\}$), allora il modulo è il seguente:*

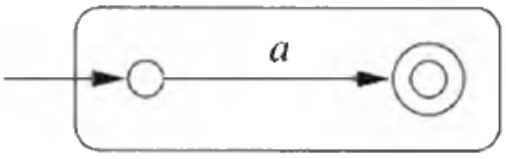


(a)

2. *Se $R = \emptyset$ (quindi $L(\emptyset) = \emptyset$), allora il modulo è il seguente:*


(b)

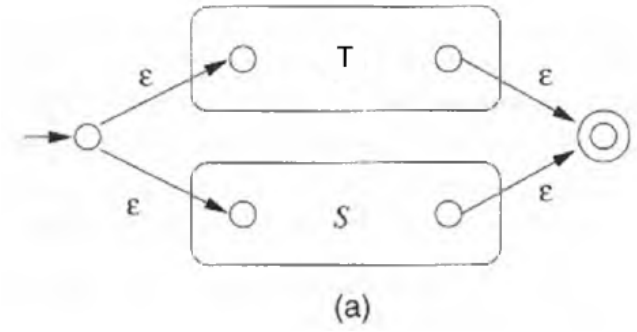
3. *se $R = a$ (e quindi $L(R) = \{a\}$), allora il modulo è il seguente:*



(c)

Passo induttivo:

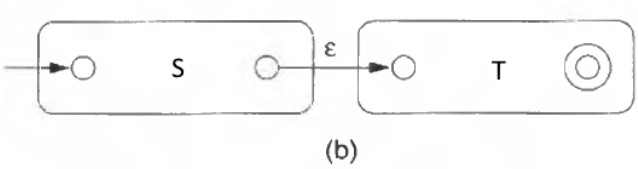
1. *Se $R = S + T$ (e quindi $L(R) = L(S) \cup L(T)$) allora il modulo è il seguente:*



(a)

L'espressione è $S + T$ per due espressioni più piccole S e T. Partendo dal nuovo stato iniziale, possiamo andare nello stato iniziale dell'automa per T o nello stato iniziale dell'automa per S. Possiamo quindi arrivare nello stato accetante di uno dei due atomi seguendo un cammino etichettato da una stringa che si trova o in $L(S)$ o in $L(T)$. Una volta arrivati in uno dei due stati accettanti per $L(S)$ o $L(T)$, possiamo usare uno dei due archi ε per arrivare nello stato accetante del nuovo automa. Il linguaggio riconosciuto è pertanto $L(S) \cup L(T)$

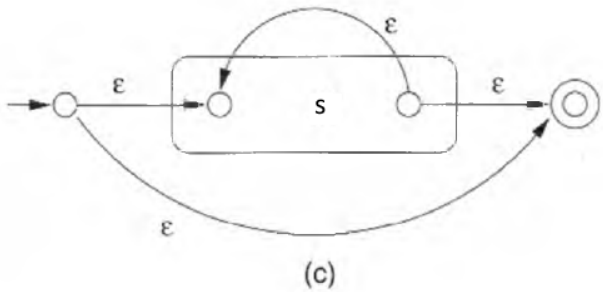
2. *Se $R = ST$ (e quindi $L(R) = L(S) \circ L(T)$), allora il modulo è il seguente:*



(b)

*L'espressione è ST per due espressioni più piccole S e T. Si noti come lo stato accettante del 2° automa diventa lo stato accentate per l'automa complessivo. L'idea è che un cammino dallo stato iniziale a quello accettante deve prima seguire un cammino etichettato da una stringa in $L(S)$ e poi un cammino etichettato da una stringa in $L(T)$; quindi i cammini dell'automa sono **tutti e soli** i cammini etichettati dalle stringhe in $L(S) \circ L(T)$*

3. *$R = S^*$ (quindi $L(R) = (L(S))^*$, allora il modulo è il seguente:*



(c)

L'espressione è S^* per un'espressione più piccola S. Abbiamo quindi due tipi di percorso:

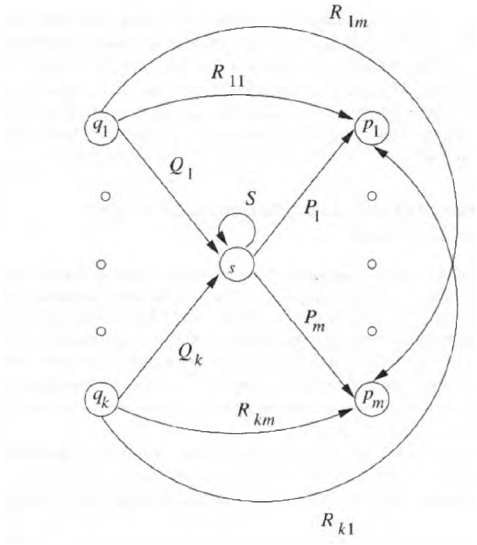
- *Direttamente dal 1° stato del modulo allo stato accettante attraverso una ε-transizione. Tale cammino fa accettare ε, che si trova in $L(S^*)$. a prescindere da S.*
- *Si passa per lo stato iniziale dell'automa per S, attraverso l'automa una o più volte, e poi verso lo stato accetante. Così facendo, copriamo tutte le stringhe di $L(S^*)$ eccetto ε*

4. *$R = (S)$ (quindi $L(R) = L(S)$), allora il modulo per R è uguale al modulo per S.*

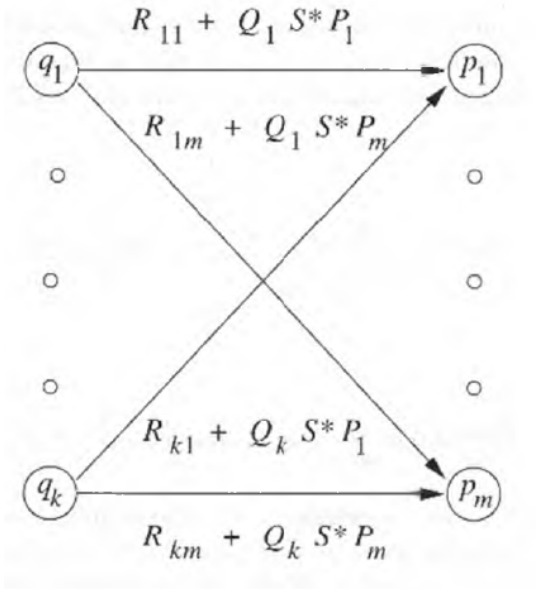
Si osservi quindi che i "moduli" (automi) definiti rispettano le 3 condizioni date nell'ipotesi induttiva.

4.2 Da DFA a Espressione Regolare tramite eliminazione di stati

Mostriamo ora la costruzione di un espressione regolare a partire dal DFA che la accetta mediante l'eliminazione di stati. Quando eliminiamo uno stato s, tutti i cammini che passano per s non esisteranno più nel nuovo automa; per non cambiare il linguaggio da esso accettato dobbiamo includere, su un arco che va direttamente da uno stato q a uno stato p, le etichette dei cammini che vanno da q a p attraverso s. Per farlo, etichettiamo gli archi con le **stringhe** che rappresentano la transizioni. Poiché le transizioni potrebbero essere rappresentate da infinite stringhe, utilizziamo le **espressioni regolari** per rappresentarle. Siamo dunque portati a considerare automi che hanno come etichette degli archi delle espressioni regolari e il **linguaggio dell'automa** è l'unione, su tutti i cammini dallo stato iniziale a uno stato accettante, dei linguaggi formati concatenando i linguaggi delle espressioni regolari lungo un cammino. Si noti che questa regola è coerente con la definizione di linguaggio associato ad un qualunque tipo di automa trattato fin qui. Ogni simbolo a o ε può essere pensato come un'espressione regolare il cui linguaggio è la stringa $\{a\}$ o $\{\varepsilon\}$. Passiamo ora alla descrizione della procedura: consideriamo il seguente automa con uno stato s che vogliamo eliminare:



Supponiamo che gli stati $q_1, ..., q_k$ siano predecessori di s e gli stati $p_1, ..., p_m$ siano successori di s. È possibile che alcuni dei q siano anche fra i p, ma supponiamo che s non si trovi tra i q o i p, anche se esiste un ciclo da s a se stesso. Mostriamo anche un'espressione regolare su ciascun arco da uno dei q ad s; l'espressione Q_i etichetta l'arco uscente da q_i. In modo analogo, mostriamo l'espressione regolare P_i che etichetta l'arco da s a p_i per ogni i. Mostriamo un ciclo su s con etichetta S. Infine, esiste un espressione regolare R_{ij} sull'arco che va da q_i a p_j per tutti i valori di i e j. Si osservi che alcuni di questi archi **potrebbero non esistere nell'automa**, quindi li possiamo pensare come archi etichettati con $\emptyset$. Eliminiamo ora lo stato, portando l'automa in questa forma:



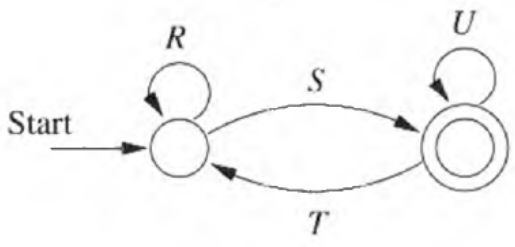
Per compensare la cancellazione di s, introduciamo per ogni predecessore q_i e ogni successore p_j di s un'espressione regolare che rappresenta tutti i cammini che partono da q_i, vanno a s, ciclano per 0 o più volte su s e infine vanno a p_j. L'espressione per questi cammini è:

$$Q_i S^* P_j$$

Tale espressione viene aggiunta, attraverso l'operatore di unione(+), all'arco da q_i a p_j. Se non esiste un arco tra q_i e p_j, ne introduciamo prima uno, etichettandolo con l'espressione regolare $\emptyset$.
Descriviamo quindi la procedura per ottenere un'espressione regolare partendo da un'automa a stati finiti:

1. Per ogni stato accettante q applichiamo il processo di riduzione visto sopra per produrre un'automa equivalente con le etichette sugli archi costituite da **espressioni regolari**. **Eliminiamo tutti gli stati eccetto q e lo stato iniziale q_0.**

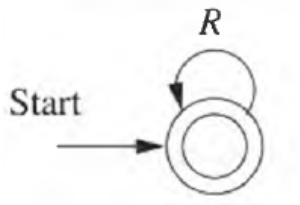
2. Se $q \neq q_0$, allora il risultato è un automa a due stati simile al seguente:



L'espressione regolare per le stringhe accettate si può descrivere in vari modi, uno dei quali è:

$$E_i = (R + SU^*T)^*SU^*$$

3. Se lo stato iniziale è anche accettante, dobbiamo eliminare **tutti gli stati dell'automa originale**, eccetto quello iniziale. Così facendo otteniamo un automa a uno stato, simile al seguente:



L'espressione regolare che indica le stringhe accettate dall'automa è la seguente:

$$E_i = R^*$$

4. L'espressione regolare desiderata è la somma (unione) di tutte le espressioni regolari derivate dagli automi ridotti **per ogni stato accettante**, in virtù delle regole (2) e (3).

4.2.1 Ordine di eliminazione degli stati

Uno stato viene eliminato in tutti gli automi derivati quando non è né iniziale, né accettante. Perciò, un vantaggio del processo di eliminazione degli stati è che possiamo iniziare eliminando ogni stato che non è né iniziale né accettante. Lo sforzo si duplica se dobbiamo eliminare degli stati accettanti; tuttavia possiamo procedere in questa maniera: supponiamo di avere 3 stati accettanti p, q, r. È possibile eliminare p, e poi eliminare separatamente q ed r, producendo gli automi ridotti rispettivamente per r e q. Poi ricominciamo con tutti e 3 gli stati accettanti ed eliminiamo sia q ed r, ottenendo l'automa per p

5 Proprietà dei linguaggi regolari

5.1 Chiusura dei linguaggi regolari

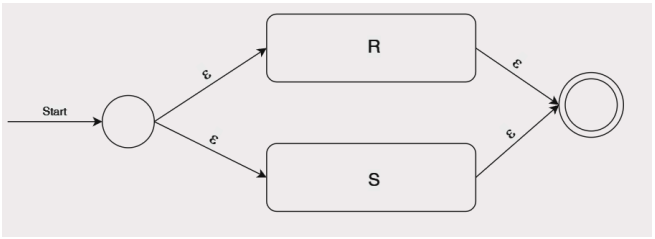
Indichiamo con REG la **classe dei linguaggi regolari** e dimostriamo la loro **chiusura** rispetto a varie operazioni.

5.1.1 Chiusura rispetto all'unione, concatenazione e chiusura di Kleene

Teorema 5.1.1 *Se $L, M \in REG$, allora $L \cup M \in REG$*

Dimostrazione 5.1.1 *Se L, M sono regolari allora esistono due espressioni regolari R, S t.c. $L(R) = L, L(S) = M$. Allora $L \cup M = L(R + S)$*

Dimostrazione 5.1.2 (Alternativa) *Se $L, M \in REG$, allora esistono due automi a stati finiti R ed S che riconoscono rispettivamente L ed M. Il seguente ε-NFA riconosce $L \cup M$, quindi esso è regolare.*



Inoltre, se $L \subseteq \Sigma_1^*$ e $M \subseteq \Sigma_2^*$ il teorema vale ancora? Si poiché $\Sigma = \Sigma_1 \cup \Sigma_2$ quindi $L, M \subseteq \Sigma$

Teorema 5.1.2 *Se $L, M \in REG$, allora $LM \in REG$*

Teorema 5.1.3 *Se $L \in REG$, allora $L^* \in REG$*

5.1.2 Chiusura rispetto alla complementazione

Teorema 5.1.4 *Se $L \in REG$, su Σ ($L \subseteq \Sigma^*$), allora anche il complemento di L, $\overline{L} = \Sigma^* \backslash L$ è regolare.*

Dimostrazione 5.1.3 *Se $L \in REG$, allora $\exists DFA\ A = (Q, \Sigma, \delta, q_0, F)$ t.c. L è il linguaggio accettato da A. Allora $\overline{L} = L(B)$ dove B è il DFA:*

$$B = (Q, \Sigma, \delta, q_0, Q \backslash F)$$

Osservazione 5.1.1 *$\overline{L}$ dipende dall'alfabeto Σ. Quindi $\overline{L} = \Sigma^* \backslash L$. Se definisco L su $\Sigma \subset \Gamma$, allora il complemento sarà $\overline{L} = \Gamma^* \backslash L$*

Proposizione 5.1.1 *La cardinalità di Σ e Γ è isomorfa alla cardinalità dei numeri naturali:*

$$|\mathbb{N}| = \aleph_0$$

Come costruiamo una ER per $\overline{L}$ partendo da una ER per L? Possiamo questa procedura:

1. Costruiamo l'ε-NFA per L
2. Convertiamo l'ε-NFA per L in DFA
3. Trasformiamo gli stati finali del DFA in stati non finali e viceversa, ottenendo il DFA per $\overline{L}$
4. Ricostruiamo l'ER per $\overline{L}$ attraverso la procedura di eliminazione degli stati.

5.1.3 Chiusura rispetto all'intersezione

Teorema 5.1.5 *Se $L, M \in REG$, allora $L \cap M \in REG$ (su Σ).*

Dimostrazione 5.1.4 *Per le leggi di De Morgan:*

$$L \cap M = \overline{\overline{L} \cup \overline{M}}, \overline{L} \in REG, \overline{M} \in REG, \overline{L} \cup \overline{M} \in REG$$

$$\overline{\overline{L} \cup \overline{M}} \in REG$$

5.1.4 Chiusura rispetto alla differenza insiemistica

Teorema 5.1.6 *Se $L, M \in REG$, allora $L \backslash M \in REG$*

Dimostrazione 5.1.5 *Vale $L \backslash M = L \cap \overline{M}$. Poiché $M \in REG$ allora $\overline{M} \in REG$, quindi anche $L \cap \overline{M} \in REG$*

5.2 Equivalenza e minimizzazione di automi

Prima di presentare il processo di minimizzazione di un DFA, diamo prima qualche nozione preliminare:

5.2.1 Relazioni di equivalenza tra stati

Definizione 5.2.1 *Dati due stati p e q, essi sono **equivalenti** se per ogni stringa $w \in \Sigma^*$, $\hat{\delta}(p, w) \in F \Leftrightarrow \hat{\delta}(q, w) \in F$ (quindi entrambi finali o non finali)*

Se p e q sono equivalenti, allora lo indicheremo con $p \approx q$. In caso non lo siano, lo indicheremo con $p \not\approx q$.

Teorema 5.2.1 *$\approx$ è una relazione di equivalenza su Q.*

Dimostrazione 5.2.1 *La relazione è:*

- ***Riflessiva**: $p \approx p \; \forall p \in Q$*
- ***Simmetrica**: Se $p \approx q$ allora $q \approx p \; \forall p, q \in Q$*
- ***Transitiva**: Se $p \approx q$ e $q \approx r$ allora $p \approx r \; \forall p, q, r \in Q$*

Quindi $\approx$ determina delle **classi di equivalenza** su Q. Le classi di equivalenza sono tra loro **disgiunte**, cioè non vi è nessun elemento che può appartenere a due classi di equivalenza distinte.
Tutti gli elementi appartengono ad una classe di equivalenza; l'insieme di tutte le classi di equivalenza forma una **partizione di Q**.

Nota 5.2.1 *Se vogliamo dimostrare che $p \not\approx q$, basta trovare una stringa $w \in \Sigma^*$ t.c. $\hat{\delta}(p, w) \in F$ ma $\hat{\delta}(q, w) \notin F$. Se esiste, tale stringa è detta **testimone** del fatto che $p \not\approx q$. Se due stati non sono equivalenti si dicono **distinguibili***

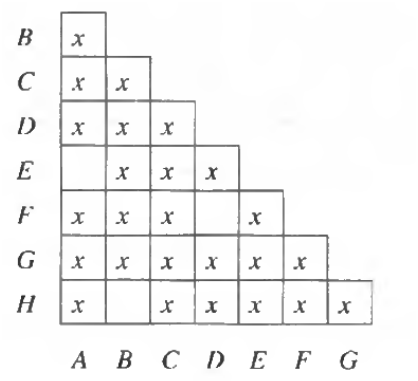
Per trovare gli stati equivalenti dobbiamo prima conoscere gli stati **distinguibili**; per trovarli applichiamo **l'algoritmo riempi-tabella**:

Proposizione 5.2.1

- ***Base:*** *Se p è uno stato accettante e q non è uno stato accettante, allora la coppia $\{p, q\}$ è distinguibile.*
- ***Induzione:*** *Siano p e q due stati tali che, per un simbolo di input a, $r = \delta(p, a)$ e $s = \delta(q, a)$ sono stati che sappiamo essere distinguibili. Allora $\{p, q\}$ sono distinguibili. La regola vale perché dev'esserci una stringa w che distingue r ed s; in altre parole solo uno tra $\hat{\delta}(r, w)$ e $\hat{\delta}(s, w)$ è accettante. Ma allora la stringa ax distingue p da q perché $\hat{\delta}(p, aw)$ e $\hat{\delta}(q, aw)$ coincidono rispettivamente con $\hat{\delta}(r, w)$ e $\hat{\delta}(s, w)$*

O in altre parole:

1. Creo una tabella con tutte le combinazioni tra gli stati dell'automa; evitando i controlli duplicati
2. Metto un segno nella tabella per ogni coppia di stati "immediatamente" distinguibile; cioè per ogni coppia per la quale il testimone ha lunghezza 1.
3. Aumento la lunghezza del testimone cercato e itero il punto 2; se metto almeno un segno, itero di nuovo l'algoritmo; altrimenti termino



Teorema 5.2.2 *Se due stati non sono distinti dall'algoritmo riempi-tabella, allora* ***sono equivalenti***

Possiamo ora costruire l'automa minimo, i quali stati sono le **classi di equivalenza**

Osservazione 5.2.1 *Non ci può essere una classe di equivalenza che contiene stati finali e non finali*

Proposizione 5.2.2 *Per costruire l'automa minimo e applicare l'algoritmo riempi-tabella, è necessario eliminare gli* ***stati non raggiungibili*** *prima di procedere.*

Proposizione 5.2.3 *La relazione di equivaelnza $\approx$ è* ***stabile*** *rispetto alla δ, cioè prendendo un rappresentate di classe qualsiasi e calcolandone la δ si finisce sempre in un elemento appartenente ad una classe di equivalenza*

Proposizione 5.2.4 *L'automa minimo è* ***unico***

5.3 Equivalenza di linguaggi regolari

Siano $L, M \in REG$. Essendo regolari, esistono due DFA A_L e A_M:

$$A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$$

$$A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$$

Costruiamo un nuovo automa A:

$$A = (Q_L \cup Q_M, \Sigma, \delta, q_L, F_L \cup F_M)$$

Con δ che si comporta come δ_L sul "diagramma degli stati" di A_L e come δ_M sul "diagramma degli stati" di A_M. Quindi:

$$L = M \Leftrightarrow q_L \approx q_M$$

cioe'

$$\Leftrightarrow \forall w \in \Sigma^*, \hat{\delta}(q_L, w) \in F_L \Leftrightarrow \hat{\delta}(q_M, w) \in F_M$$

cioe'
$\Leftrightarrow w$ è accettata sia da A_L che da A_M oppure è rifiutata da entrambi. Per scoprire se i due automi sono uguali posso quindi applicare l'algoritmo riempi-tabella su A.

Nota 5.3.1 *Se, con una stringa w, il segno finisce in una coppia con già un segno per lo stesso simbolo, allora sono* ***distinguibili***

5.4 Tempi di calcolo dell'algoritmo riempi-tabella

Sia Q l'insieme di stati di un DFA. Allora $|Q| = n$ e $|Q \times Q| = n^2$. Durante l'esecuzione dell'algoritmo riempi-tabella analizziamo un numero di coppie distinte (p, q) uguale a n^2. Ciò vuol dire che:
Complessità temporale ad ogni passo intermedio: $O(n^2)$
Complessità temporale totale: $O(n^4)$

Nota 5.4.1 *In verità, con una struttura dati che permetta l'accesso diretto alle celle vuote della tabella, si hanno tempi di calcolo leggermente ridotti; ma sempre nell'ordine di quanto riportato sopra.*

5.5 Perché il DFA minimo non può essere migliorato

Supponiamo di minimizzare un DFA A con l'algoritmo riempi-tabella e di ottenere come risultato un DFA M. Può esistere un DFA N che accetta lo stesso linguaggio di A e di M ma con un numero di stati minore di M? La risposta è no:

Dimostrazione 5.5.1 *Cominciamo eseguendo la procedura per riconoscere gli stati* ***distinguibili*** *sugli stati di M ed N, come se fossero un solo DFA. Possiamo suppore che i nomi degli stati di M siano distinti da quelli di N; in questo modo, la funzione di transizione δ dell'automa composto è* ***l'unione delle due funzioni****, senza interferenze. Un stato è accettante nel DFA composto se e solo se è accettante nel DFA di provenienza. Gli stati iniziali di M ed N sono* ***indistinguibili*** *poiché $L(M) = L(N)$;*

inoltre, se p e q sono indistinguibili, lo sono anche i loro successori per qualsiasi simbolo di input: infatti, se potessimo distinguere i loro successori, potremmo anche distinguere p e q. Né M né N hanno stati ***inaccessibili****; in caso contrario, possiamo eliminarli ed ottenere un DFA più piccolo per lo stesso linguaggio.*
Perciò, ogni stato di M è ***indistinguibile*** *da almeno uno stato di N. Per convincersene, sia p uno stato di M. Deve esistere una stringa $a_1a_2...a_k$ che porta dallo stato iniziale di M a p. La stessa stringa porta anche dallo stato iniziale di N a uno stato q. Sappiamo che gli stati iniziali sono* ***indistinguibili****, quindi anche i loro successori rispetto ad a_1 sono* ***indistinguibili****; lo stesso vale per i successori di questi stati rispetto ad a_2 e così via, fino a concludere che p e q sono indistinguibili. Poiché N ha* ***meno stati*** *di M, ci sono due stati di M* ***indistinguibili*** *dallo stesso stato di N, e quindi* ***indistinguibili fra loro****. Ma M è stato costruito come essere l'automa minimo, quindi esso deve avere tutti i suoi stati* ***distinguibili fra loro a due a due****; abbiamo quindi un assurdo! Quindi* ***non può esistere*** *N ed M ha un numero di stati* ***non superiore*** *a quello di un qualsiasi DFA equivalente ad A*

In termini formali, abbiamo dimostrato il seguente teorema:

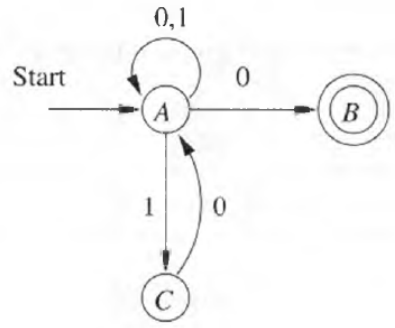
Teorema 5.5.1 *Sia A un DFA ed M il DFA costruito a partire da A applicando l'algoritmo riempi-tabella. Allora il numero di stati di M non è superiore a quello di un qualsiasi DFA equivalente ad A.*

Possiamo inoltre affermare che, con la dimostrazione precedente, abbiamo provato che esiste una corrispondenza **uno-a-uno** fra gli stati di due automi minimi N ed M. Poiché abbiamo dimostrato che ogni stato di M deve essere equivalente ad un solo stato di N e che non può essere equivalente a due o più stati di N; allora possiamo facilmente dimostrare, in maniera simile, il contrario. In conclusione, quindi, vale il seguente teorema:

Teorema 5.5.2 *Il DFA minimo equivalente ad A è* ***unico****, a meno di rinominare gli stati*

5.5.1 Perché l'algoritmo riempi-tabella non funziona sugli NFA

Si potrebbe pensare che l'algoritmo riempi-tabella possa essere applicato anche per la ricerca di NFA minimi a DFA assegnati. Invece, anche se è possibile trovare un NFA con numero minimo di stati che accetta un determinato linguaggio, non è sufficiente ripartire gli stati di un NFA per quel linguaggio. Facciamo un esempio:



In questo NFA, **non ci sono coppie di stati equivalenti**; B è di sicuro distinguibile da A e C; A e C sono distinguibili per l'input 0; A è l'unico successore di C e non è accettante, mentre l'insieme dei successori di A è $\{A, B\}$, che contiene uno stato accettante. Perciò, non si può ridurre il numero di stati raggruppando quelli equivalenti. Eppure, si può trovare un NFA più piccolo per questo linguaggio **rimuovendo** C; infatti, i soli stati A e B accettano tutte le stringhe del linguaggio e l'aggiunta di C non permette di accettare altre stringhe.

5.6 Pumping Lemma

Il Pumping Lemma è un teorema(Lemma) che permette di **dimostrare**, procedendo per assurdo, che un certo linguaggio L non è regolare.

Nota 5.6.1 *Le **prefisse** di una stringa sono le stringhe ottenute rimuovendo simboli dalla fine della stringa.*

Consideriamo il linugaggio:

$$L_{01} = \{0^n 1^n | n \geq 1\}$$

Vogliamo dimostrare che è regolare. Se per assurdo, L_{01} fosse regolare, allora esisterebbe un DFA che lo accetta. Sia quindi $A = (Q, \Sigma, \delta, q_0, F)$ e supponiamo che $|Q| = k$. Diamo in input ad A la stringa 0^k; essa ha $k+1$ prefisse $(\varepsilon, 0, 00, 000...)$ Poiché ci sono k stati, devono esistere due prefisse **diverse**, 0^i e 0^j, con $i \neq j$, che portano allo stesso stato, **quindi non sono distinguibili**. A deve accettare $0^i 1^i$ poiché appartiene al linguaggio, tuttavia accetterebbe anche $0^j 1^i$, e quindi **sbaglia**; quindi A non può esistere e L_{01} non è regolare.

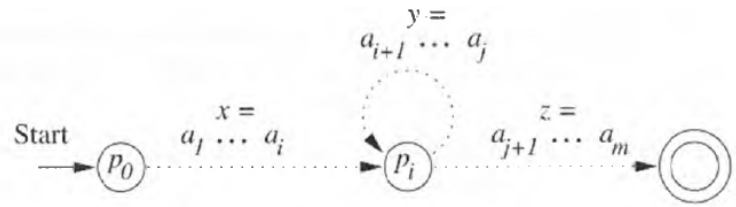
Teorema 5.6.1 *Sia L un linguaggio regolare, allora esiste una costante n (dipendente da L) t.c. $\forall w \in L$ t.c. $|w| \geq n$, allora w può essere scomposta come $w = xyz$ in modo che valgano:*

1. $y \neq \varepsilon$
2. $|xy| \leq n$
3. $\forall k \geq 0$, *anche* $xy^k z \in L$

Dimostrazione 5.6.1 *Se $L \in REG$, allora esiste un DFA A che lo riconosce. Supponiamo che A abbia n stati e consideriamo una stringa $w = a_1 a_2 ... a_m$ t.c. $m \geq n$ (quindi $|w| \geq n$) con $a_i \in \Sigma \; \forall i = 1, ..., m$. Per ogni $i = 0, ..., n$, sia $p_i = \hat{\delta}(q_0, a_1, ..., a_i)$. Avremo che $p_0 = q_0$. Dato che A ha n stati, gli $n+1$ stati $p_0, p_1, ..., p_n$ non possono essere tutti distinti. Allora $\exists i, j$ con $0 \leq i < j \leq n$ tali che $p_i = p_j$. Scomponiamo $w = xyz$, allora:*

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

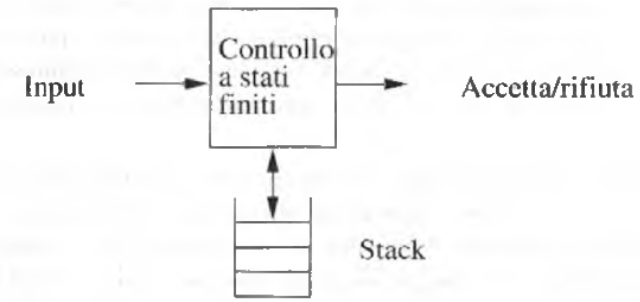
In altre parole, x porta fino a p_i, y porta fino a p_j, il quale coincide con p_i e infine z conclude w. Le relazioni fra stati e stringhe sono rappresentabili con il seguente automa: (il cappio indica una qualsiasi cammino che ritorna a p_i):



Consideriamo ora cosa accade se l'automa riceve in input la stringa xy^kz per $k \geq 0$. Se $k = 0$, l'automa passa dallo stato iniziale allo stato p_i sull'input x. Poiché p_i coincide con p_j, l'automa deve passare allo stato accettante sull'input z, quindi l'automa accetta xz.
Se $k > 0$, A va da p_0 in p_i su input x, cicla su p_i per k volte e poi passa allo stato finale su input z, quindi l'automa accetta la stringa xy^kz. La stringa w quindi soddisfa le proprietà enunciate nel teorema.

6 Automi a Pila

Un automa a Pila o PDA è essenzialmente un automa **non deterministico** dotato di uno **stack** nel quale memorizzare una stringa di simboli. A differenza degli automi a stati finiti, grazie allo stack l'automa può ricordare una sequenza potenzialmente **infinita** di caratteri, tuttavia esso può accedervi solo in ordine **LIFO**. In generale, gli automi a pila riconoscono **tutti e soli i linguaggi liberi dal contesto** (Context-Free)



Dotiamo questo stack di 3 operazioni fondamentali:

1. **Push**: Immette un simbolo nella pila
2. **Pop**: Estrae un simbolo dalla pila
3. **Top**: Controlla cosa c'è in cima alla pila

Definizione 6.0.1 *Un PDA è una 7-upla:*

$$P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$$

Dove:

- Q *è l'insieme finito degli stati*
- Σ *è l'insieme dei simboli in ingresso*
- Γ *è l'insieme dei simboli della pila*
- δ *è la funzione di transizione degli stati*
- $q_0 \in Q$ *è lo stato iniziale*
- $z_0 \in \Gamma \backslash \Sigma$ *è un simbolo che indica che la pila è* ***logicamente vuota****. Esso viene introdotto poiché la funzione* δ *ha bisogno che ci sia* ***almeno un simbolo in cima alla pila*** *per effettuare le sue operazioni. Se vogliamo terminare la computazione,* ***rimuoviamo*** z_0 *dalla pila.*
- F *è l'insieme degli stati finali*

Definiamo ora la funzione di transizione δ:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}$$

Dove:

- Q è lo stato attuale dell'automa
- $(\Sigma \cup \{\varepsilon\})$ è il simbolo in ingresso, appartenente a Σ oppure ε
- Γ è il simbolo **correntemente in cima alla pila**

La funzione di transizione restituisce un **insieme di coppie** (q, γ) dove q è il nuovo stato dell'automa e γ è la stringa di simboli che vanno a rimpiazzare la cima dello stack (elencati da sinistra verso destra). Per rimuovere un elemento dalla pila, lo si sostituisce con ε.

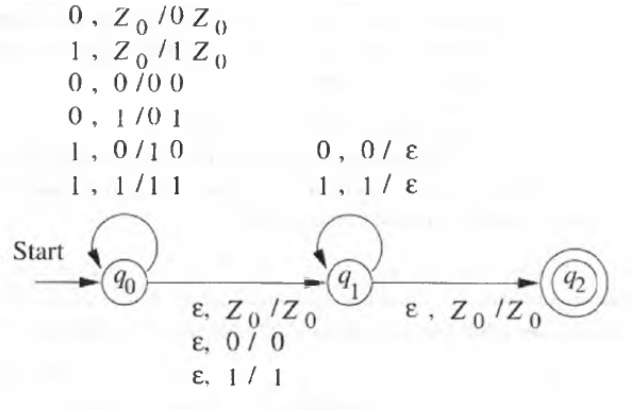
Nota 6.0.1 *Possiamo aggiungere più stringhe alla pila, ma possiamo togliere un carattere alla volta.*

6.1 Notazione grafica per i PDA

Per costruire il diagramma di transizione per un PDA, si seguono le seguenti regole:

1. I nodi corrispondono agli stati del PDA
2. Una freccia etichettata **START** indica lo stato iniziale, mentre gli stati contrassegnati da un doppio cerchio sono finali

3. Gli archi corrispondono alle transizioni del PDA. In particolare, le etichette degli archi sono della forma $a, X/\alpha$, dove a è il simbolo in input, X è il simbolo **correntemente in cima alla pila** e α è la stringa che verrà immessa in cima alla pila effettuando quella transizione.



6.2 Configurazioni di un PDA

Notazione 6.2.1 *Configurazione di un PDA = Descrizione istantantea di un PDA (ID)*

Finora abbiamo dato solo una vaga nozione di come un PDA computa. Intuitivamente, il PDA passa da una configurazione all'altra reagendo ai vari simboli di input ma, diversamente dagli automi a stati finiti, la configurazione di un PDA comprende anche **il contenuto dello stack**. Essendo di grandezza arbitraria, lo stack è spesso la parte più importante della configurazione. Rappresentiamo quindi la configurazione di un PDA come una **tripla** (q, w, γ) con q lo stato attuale del PDA, w "l'input residuo" e γ il contenuto della pila. Per convenzione, mostriamo la sommità dello stack all'estremo sinistro e il fondo all'estremo destro della stringa γ.

6.2.1 Passo di computazione

Per indicare il passo di computazione (o "mossa") di un PDA e il suo cambio di ID definiamo un nuovo simbolo: $\vdash_P$ (se il PDA ha nome P)

Definizione 6.2.1 *Dato un PDA* $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$*, supponiamo che* $\delta(q, a, x)$ *contenga* (p, α)*; allora* $\forall w \in \Sigma^*$ *e* $\forall \beta \in \Gamma^*$

$$(q, aw, x\beta) \vdash_P (p, w, \alpha\beta)$$

Nota 6.2.1 *L'insieme di coppie di output di* δ ***deve*** *essere finito; ciò è importante specificarlo poiché seppur il dominio è* ***infinito****, l'insieme* $2^{Q \times \Gamma^*}$ *è* ***potenzialmente infinito***

Nota 6.2.2 *La definizione di computazione mette in relazione due ID* $\Leftrightarrow$ *sono legati da un passo di computazione di un caso della* δ*. Quindi* $\vdash$ *è una* ***relazione binaria*** *tra due configurazioni legate da un* ***passo di computazione***

Definizione 6.2.2 *Dato un PDA* $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ *e date due sue configurazioni* I *e* J*, se* $I \vdash_P^* J$*, si dice allora che* J *è raggiungibile da* I ***in zero o più passi di computazione****(o mosse). La definizione procede per induzione:*

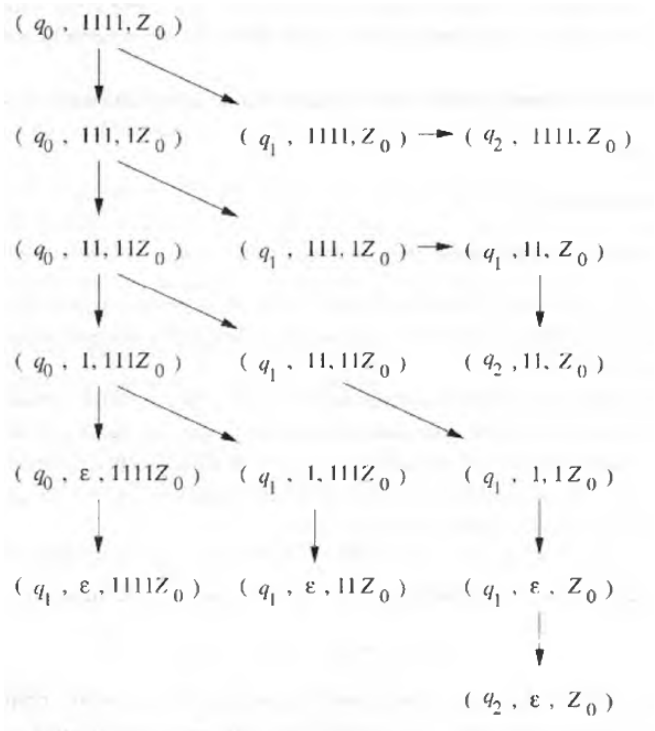
- ***Caso base****:* $I \vdash_P^* I$ *per ogni ID* I
- ***Passo induttivo****:* $I \vdash_P^* J$ *se* $\exists\, ID\ K$ *t.c.* $I \vdash_P K$ *e* $K \vdash_P^* J$

La definizione può essere anche data senza utilizzare l'induzione:
$I \vdash_P^* J$ *se esiste una sequenza di ID* $K_1, K_2, ..., K_n$ *t.c.:*

- $I = K_1$
- $J = K_n$
- $\forall i = 1, ..., n-1,\ K_i \vdash_P K_{i+1}$

Quindi $I = K_1 \vdash K_2 \vdash ... \vdash K_{n-1} \vdash K_n = J$

Tramite le ID possiamo quindi costruire anche un **albero di computazione**, così da mostrare ogni possible computazione e ogni possibile scelta non deterministica che l'automa può fare in ogni suo possibile stato:



Teorema 6.2.1 *Se* $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ *è un PDA e* $(q, x, \alpha) \vdash_P^* (p, y, \beta)$ *allora* $\forall w \in \Sigma^*$ *e* $\forall \gamma \in \Gamma^*$ *vale che:*

$$(q, xw, \alpha\gamma) \vdash_P^* (p, yw, \beta\gamma)$$

Teorema 6.2.2 *Se* $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ *e* $(q, xw, \alpha) \vdash_P^* (p, yw, \beta)$ *allora vale anche:*

$$(q, x, \alpha) \vdash_P^* (p, y, \beta)$$

6.3 Accettazione da parte di un PDA

6.3.1 Accettazione per stati finali

Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ un PDA. Il linguaggio accettato da P per **stato finale** è:

$$L(P) = \{w \in \Sigma^* | (q_0, w, z_0) \vdash_P^* (q, \varepsilon, \alpha) \; con \; q \in F \; e \; \alpha \in \Gamma^*\}$$

6.3.2 Accettazione per pila vuota

Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ un PDA. Il linguaggio accettato da P per **pila vuota** è:

$$N(P) = \{w \in \Sigma^* | (q_0, w, z_0) \vdash_P^* (q, \varepsilon, \varepsilon) \; con \; q \in Q\}$$

Se stiamo considerando un PDA che accetta per pila vuota, allora possiamo **omettere** l'insieme degli stati finali F; portando il PDA ad essere una sestupla del tipo:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, z_0)$$

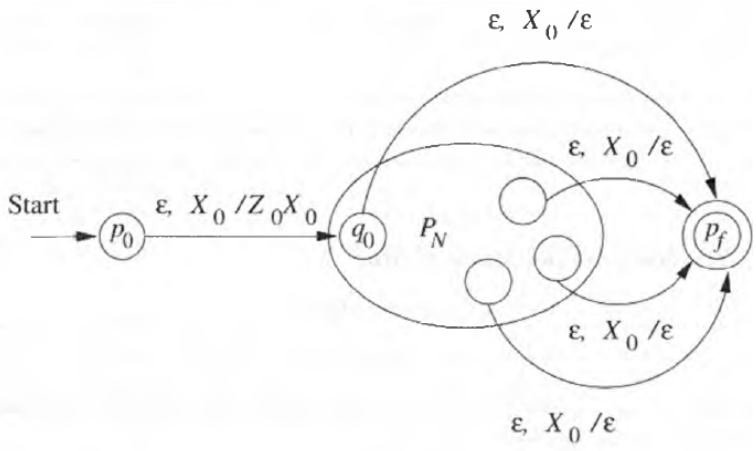
6.3.3 Da accettazione per pila vuota a per stati finali

Teorema 6.3.1 *Se L è un linguaggio t.c $L = N(P_N)$ per un PDA $P_N = (Q, \Sigma, \Gamma, \delta, q_0, z_0)$ allora esiste un PDA P_F t.c $L = L(P_F)$*

Dimostrazione 6.3.1 *L'idea su cui poggia la dimostrazione è l'utilizzo di un nuovo simbolo x_0 che non appartiene a Γ; x_0 è sia il simbolo iniziale di P_F sia il simbolo che indica quando P_N ha svuotato lo stack sullo stesso input. Quindi, P_F avrà la seguente struttura:*

- *Un nuovo stato iniziale p_0 che, tramite una ε-transizione dia il controllo al controllo finito di P_N e che metta sullo stack il simbolo iniziale di P_N z_0*
- *Il controllo finito di P_N viene modificato, tracciando un arco ad un nuovo stato finale p_f in ogni stato dove, su un certo input, lo stack verrebbe svuotato.*

In generale, quindi, P_F simula P_N fino a quando lo stack di P_N è vuoto, condizione che porta al nuovo stato accettante p_f



La definizione formale del nuovo automa P_F è quindi la seguente:

$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{x_0\}, \delta_F, p_0, x_0, \{p_F\})$$

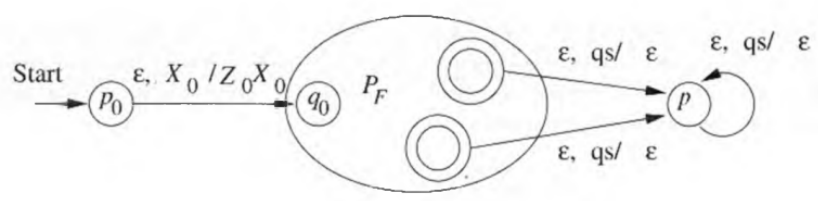
Con δ_F definita nel seguente modo:

- $\delta_F(p_0, \varepsilon, x_0) = \{(q_0, z_0, x_0)\}$
- $\forall q \in Q,\ \forall a \in \Sigma (o\ a = \varepsilon)\ e\ \forall y \in \Gamma\ \delta_F(q, a, y)$ contiene tutte le coppie di $\delta(q, a, y)$. Inoltre, $\delta_F(q, \varepsilon, x_0)$ contiene (p_f, ε)

6.3.4 Da accettazione per stati finali a per pila vuota

Teorema 6.3.2 *Sia $L = L(P_F)$ per un PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, z_0, F)$. Allora esiste un automa P_N t.c. $L = N(P_N)$*

Dimostrazione 6.3.2 *Costruiamo il nuovo automa P_N come suggerito nella seguente figura:*



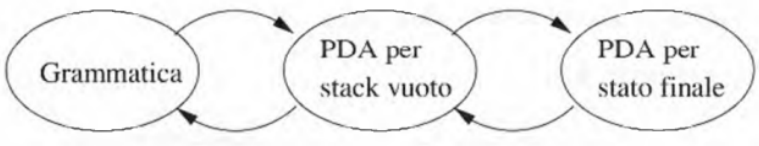
Quindi sia $P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{x_0\}, \delta_N, p_0, x_0)$
Con δ_N definita come segue:

- $\delta_N(p_0, \varepsilon, x_0) = \{(q_0, z_0, x_0)\}$
- $\forall q \in Q,\ \forall a \in \Sigma\ (o\ a = \varepsilon),\ \forall y \in \Gamma,\ \delta_N(q, a, y)$ *contiene tutte le coppie di* $\delta_F(q, a, y)$
- $\forall q \in F,\ \forall y \in \Gamma\ (oppure\ y = x_0), \delta_N(q, \varepsilon, y)$ *contiene* (p, ε)
- $\forall y \in \Gamma\ (oppure\ y = x_0), \delta_N(p, \varepsilon, y) = (p, \varepsilon)$

La dimostrazione quindi segue lo stesso ragionamento della ***dimostrazione 6.3.1****.*

6.4 Equivalenza tra PDA e CFG

Dimostriamo ora che i linguaggi accettati dai PDA sono proprio i **linguaggi liberi dal contesto**; la via da seguire è indicata nella seguente figura:



Abbiamo già dimostrato che i linguaggi che vengono accettati da PDA per **stack vuoto** sono gli stessi accettati dai PDA **per stato finale**; quindi dimostriamo che le **grammatiche libere dal contesto** definiscono una classe di linguaggi equivalente alla classe dei linguaggi accettati dai PDA per stack vuoto.

6.4.1 Da PDA a CFG

Dato un PDA P che accetta per pila vuota, si costruisce una CFG in cui **le produzioni corrispondono al passaggio tra una configurazione alla successiva** in modo tale che le derivazioni della grammatica:

$$S \Rightarrow ... \Rightarrow w$$

Corrispondano ad un **passo di computazione dell'automa** e si arrivi a produrre la stringa $w \Leftrightarrow w \in N(P)$.

6.4.2 Da CFG a PDA che accetta per pila vuota

Data una CFG $G = (V, T, Q, S)$ costruiamo il PDA P che accetta il linguaggio $L(G)$ per pila vuota:

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

Con δ definita come segue:

- $\forall A \in V, \delta(q, \varepsilon, A) = \{(q, \beta) | A \to \beta \text{ è una regola di produzione}\}$
- $\forall a \in T, \delta(q, a, a) = \{(q, \varepsilon)\}$

Osservazione 6.4.1 *Per costruire un PDA per stati finali che accetta $L(G)$, lo si deve costruire a partire dall'automa a pila vuota.*

6.5 PDA Deterministici (DPDA)

Definizione 6.5.1 *Una PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ è **determnistico** (cioè è un DPDA) se:*

1. $|\delta(q, a, x)| \leq 1 \; \forall q \in Q, \; \forall a \in \Sigma \cup \{\varepsilon\}, \; \forall x \in \Gamma$
2. *Se $|\delta(q, a, x)| \neq 0$ per qualche $a \in \Sigma$, allora $|\delta(q, \varepsilon, x)| = 0$*

Notazione 6.5.1 *Quindi nei DPDA **le ε-transizioni esistono**, ma vengono semplicemente gestite in maniera diversa.*

6.5.1 Linguaggi accettati dai DPDA

I DPDA accettano una classe di linguaggi che si pone tra **i linguaggi context-free** e i **linguaggi regolari**. Iniziamo a dimostrare che i DPDA accettano tutti i linguaggi regolari:

Teorema 6.5.1 *Se L è un linguaggio regolare, allora $L = L(P)$ per un DPDA P*

Dimostrazione 6.5.1 *Il DPDA è formato da un **controllo a stati finiti** e una pila. Se immaginiamo di **non utilizzare la pila**, cioè di leggere, togliere e rimettere z_0 costantemente, allora il DPDA sta a tutti gli effetti simulando un **DFA**.*

Tuttavia, i DPDA **non accettano tutti i linguaggi context-free** ma solo una loro sottocategoria; cioè quei linguaggi che non hanno **grammatiche ambigue**. Inoltre i DPDA possono accettare per **pila vuota** solo se il linguaggio è **prefix-free**; diamo quindi qualche definizione:

Definizione 6.5.2 *Una **prefissa** di una stringa $w \in L$ è una stringa w' ottenuta togliendo 1 o più simboli dal fondo di w*

Definizione 6.5.3 *Un linguaggio L ha la **proprietà di prefisso**(cioè è **prefix-free**) se $\nexists x, y \in L$ t.c. $x \neq y$ e x è **prefissa** di y*

Osservazione 6.5.1 *ε è **prefissa di ogni stringa** di L, il linguaggio quindi non è prefix-free se $\varepsilon \in L$*

Quindi possiamo enunciare il seguente teorema:

Teorema 6.5.2 *Un linguaggio L è $N(P)$ per un DPDA $P \Leftrightarrow L$ ha la proprietà di prefisso (cioè è **prefix-free**); inoltre è $L(P')$ per un DPDA P'*

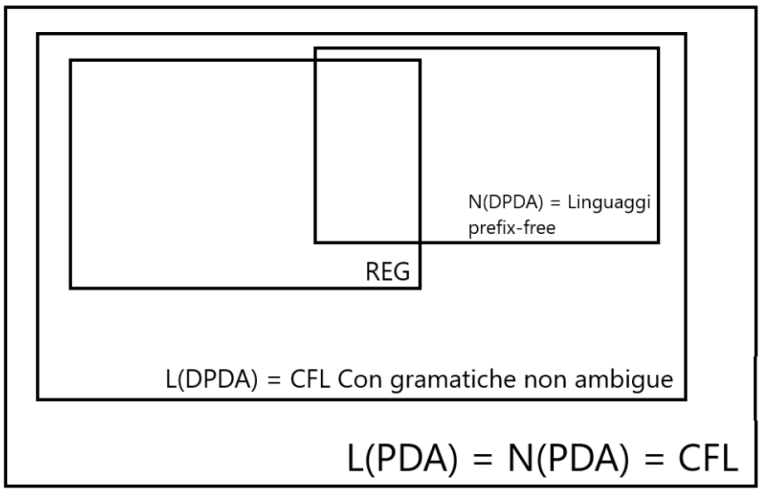
Dimostrazione 6.5.2 *Immaginiamo di avere L **NON** prefix-free, allora $\exists x, y \in L$ t.c. x è **prefissa** di y. Allora, scomponiamo y in $y = xw$ con $w \neq \varepsilon$ (poiché $x \neq y$). Assumiamo esista un DPDA P t.c. $N(P) = L$. Se esso prende in input x, allora lo accetta, perché $x \in L$; quindi la pila è vuota. Tuttavia se gli passiamo $y = xw$, l'automa si ferma **prima che tutta la stringa venga consumata**, poiché **toglie** z_0 **dopo aver analizzato x**. Quindi è assurdo che esista un DPDA del genere.*

Inoltre, possiamo enunciare i seguenti teoremi:

Teorema 6.5.3 *Se L è $N(P)$ per un DPDA P allora L ha una CFG non ambigua*

Teorema 6.5.4 *Se L è $L(P)$ per un DPDA P allora L ha una CFG non ambigua*

Riassumiamo ciò che abbiamo enunciato nel seguente schema:



7 Macchine di Turing

Alla fine dell' '800, inizio '900 ci fu il tentativo da parte di alcuni matematici, tra cui **Hilbert**, di formalizzare la matematica. Tra i problemi proposti da Hilbert vi era anche il cosidetto **Entscheidungsproblem** (problema della decisione), il quale chiedeve se esistesse una **procedura del tutto meccanica** in grado di stabilire, per ogni formula espressa nel linguaggio formale della logica del primo ordine, se tale formula è o meno un teorema della logica del primo ordine. È quindi possibile che i teoremi matematici siano **dimostrati da una macchina**? La risposta a questo problema fu data tramite i lavori di **Gödel**, **Turing** e **Church**. In particolare, Turing immaginò un "**matematico meccanico**" il cui "cervello" si trova in ogni istante in una configurazione diversa appartenente ad un insieme **finito di configurazioni**; esso dispone di un **nastro infinito** dove effettuare tutte le operazioni di lettura/scrittura e di una testina che punta, ad ogni istante, ad una cella del nastro. Ciò che quindi Turing immaginò è simile agli automi che abbiamo presentato finora; possiamo quindi immaginarne la funzione δ di transizione degli stati:

$$\delta(q, a) = (p, b, L)$$

$$\delta(q, B) = (...)$$

Il modello di calcolo teorizzato da Turing prende il nome di **macchina di Turing**(MdT), di cui adesso diamo una definizione formale dopo due brevi considerazioni:

Osservazione 7.0.1 *Durante una computazione di una MdT,* ***la stringa non viene consumata***

Nota 7.0.1 *Turing aveva inizialmente definito l'accettazione di una MdT di una stringa w come l'evento in cui la macchina si ferma e non va in loop infinito. Per renderla più conforme alla teoria degli automi, si è preferito ridefinirla con* ***stati accentanti****.*

Definizione 7.0.1 *Una MdT è una settupla:*

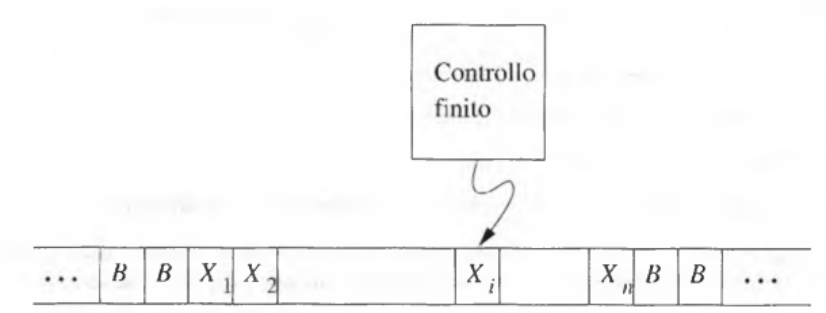
$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

Dove:

- Q *è l'insieme degli stati della macchina*
- Σ *è l'alfabeto dei simboli di input*
- Γ *è l'alfabeto dei simboli del nastro, inoltre* $\Sigma \subseteq \Gamma$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *è la funzione di transizione degli stati. Essa prende in input lo stato attuale della macchine e il simbolo* $a \in \Gamma$ *che la testina di lettura/scrittura punta e restituisce una tripla* (p, γ, D) *dove:*
  - $p \in Q$
  - $\gamma \in \Gamma$ *ed è il simbolo che andrà a sostituire* a
  - $D \in \{L, R\}$ *che è la direzione in cui si muove la testina di lettura e scrittura. Essa si può quindi spostare di una cella a sinistra o a destra.*

Osservazione 7.0.2 *Così definita,* ***una computazione porta sempre allo spostamento della testina***

- $q_0 \in Q$ *è lo stato iniziale*
- $B \in \Gamma \backslash \Sigma$ *è il simbolo "Blank", cioè un simbolo che indica una cella* ***logicamente vuota***
- $F \subseteq Q$ *sono gli stati finali*



7.1 Configurazioni (o ID) di una MdT

Un ID per una MdT ha la forma:

$$x_1 x_2 ... x_{i-1} q x_i x_{i+1} ... x_n$$

Supponiamo che $Q \cap \Sigma = \emptyset$.
La testina è posizionata **sulla cella immediatamente a destra di q**. Prima di x_1 e dopo x_n sono presenti solo simboli di blank.

7.2 Passi di computazione di una MdT

Descriviamo i passi di computazione di una MdT $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ mediante la notazione $\vdash_M$ già usata per i PDA. La relazione binaria $\vdash_M$ è assolutamente speculare a quella dei PDA, compreso il simbolo $\vdash_M^*$. Supponiamo che la M si trovi nella configurazione:

$$x_1 x_2 ... x_{i-1} q x_i x_{i+1} ... x_n$$

e che la funzione di transizione abbia il seguente caso:

$$\delta(q_i, x_i) = (p, y, L)$$

Allora la nuova configurazione è:

$$x_1 x_2 ... x_{i-2} p x_{i-1} y x_{i+1} ... x_n$$

Esistono due **casi particolari** per quanto riguarda uno spostamento a sinistra:

1. Se $i = 1$ allora: $q x_1 x_2 ... x_n \vdash_M pBy x_2 ... x_n$
2. Se $i = n$ e $y = B$, allora: $x_1 x_2 ... x_{n-1} q x_n \vdash_M x_1 x_2 ... x_{n-2} p x_{n-1}$

Quindi segnamo B solo se viene puntato dalla testina della MdT. Se invece la funzione di transizione avesse questo caso:

$$\delta(q_i, x_i) = (p, y, R)$$

allora la nuova configurazione è:

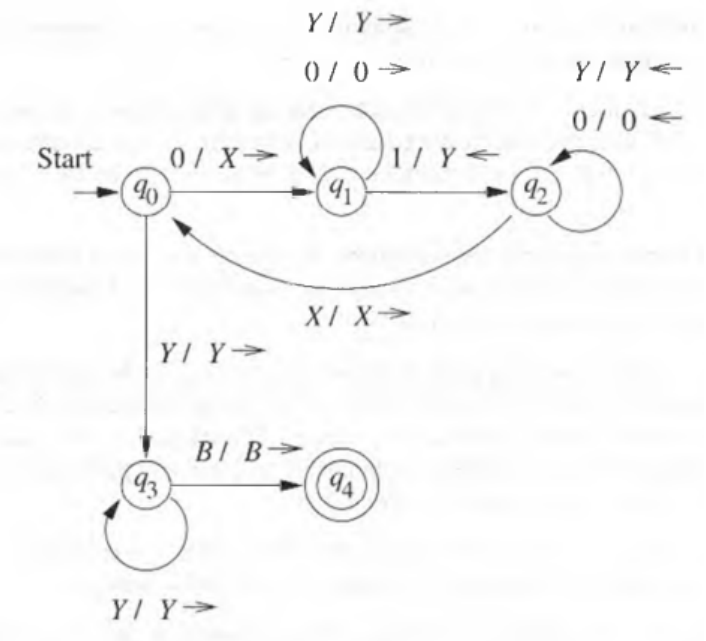
$$x_1 x_2 ... x_{i-1} y p x_{i+1} ... x_n$$

Esistono due **casi particolari** per quanto riguarda uno spostamento a destra:

1. Se $i = n$, allora: $x_1 x_2 ... q x_n \vdash_M x_1 x_2 ... x_{n-1} y p B$
2. Se $x = 1$ e $y = B$ allora: $q x_1 x_2 ... x_n \vdash_M q x_2 ... x_n$

7.3 Diagramma di transizione per una MdT

Possiamo costruire anche diagrammi di transizione per una MdT, utilizzando una notazione simile a quella dei PDA. Ogni nodo è uno **stato della MdT**. Un arco da uno stato q a uno stato p rappresenta una **possibile computazione** che porta la macchina dallo stato q allo stato p e ogni arco è etichettato da uno o più oggetti della forma X/YD dove $X \in \Gamma$ è il simbolo presente sul nastro, $Y \in \Gamma$ è il simbolo che andrà a rimpiazzare X e D è la direzione in cui si sposta la testina. Uno stato accettante è contrassegnato da un doppio cerchio, mentre lo stato iniziale è contrassegnato da un arco entrante etichettato con **START**.



7.4 Linguaggi accettati da una MdT: Linguaggi Ricorsivamente Enumerabili

Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ una MdT. Allora il linguaggio accettato da M è:

$$L(M) = \{w \in \Sigma^* | q_o w \vdash_M^* \alpha p \beta \; con \; p \in F \; e \; \alpha, \beta \in \Gamma^*\}$$

I linguaggi accettati dalle MdT si chiamano **linguaggi ricorsivamene enumerabili**, abbreviati in RE. Essi sono caratterizzabili in vari modi, tra i quali la seguente: "I linguaggi ricorsivamente enumerabili sono quei linguaggi per i quali esiste una procedura **"effettiva"**(quindi calcolabile) che stampa tutte le stringhe del linguaggio". Quindi $w \in L \Leftrightarrow$ appare in "output" dalla procedura, altrimenti **non posso affermare che non sia parte del linguaggio**. Inoltre la procedura potrebbe andare avanti all'infinito. Concludiamo presentando il seguente enunciato, tuttavia non in modo formale, chiamato **Tesi di Church-Turing**: "Tutto quello che è **computabile** con qualunque metodo di calcolo "ragionevole" è computabile da una MdT". Oltre alla MdT, esistono anche altri modelli di calcolo:

- Macchine a registri/contatori
- Sistemi di riscrittura di Post: essi riscrivono pezzi di stringhe; la computazione avviene quindi su operazioni sulle stringhe
- ecc...

Tutti i modelli di calcolo classici sono **Turing-equivalenti**, cioè hanno la stessa potenza di una macchina di Turing. Esistono modelli più potenti? Si, per esempio modelli che effettuano computazioni sui **numeri reali** oppure le **macchine quantistiche**, le quali sono uguali alle MdT tranne per la loro capacità di **generare numeri randomici**.

7.5 Estensioni della MdT

Supponiamo di avere due modelli di calcolo M_1 e M_2. Se ogni computazione di M_1 è **simulabile** da M_2, allora **le macchine della classe** M_1 **NON potranno essere più potenti delle macchine della classe** M_2, cioè non possono riconoscere linguaggi "in più". Ciò si indica con:

$$M_1 \leq M_2$$

Se vale anche

$$M_2 \leq M_1$$

allora

$$M_1 = M_2$$

Vediamo ora delle estensioni equivalenti della MdT:

7.5.1 MdT con la simbolo "Stay"

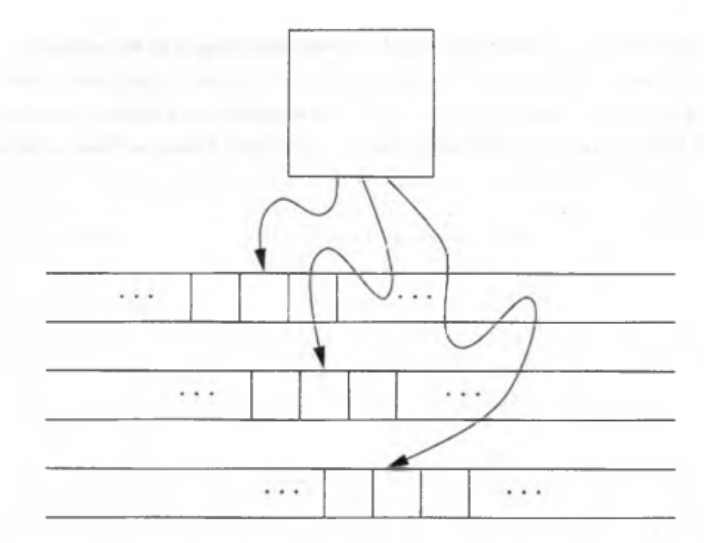
Questa estensione modifica la funzione δ di transizione nel seguente modo:

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

S indica che la testina della MdT non si andrà a spostare dopo aver effettuato un passo di computazione. Questa MdT è facilmente simulabile da una MdT "standard" poiché ogni passo che indica alla MdT di non muovere la testina può essere simulato effettuando due mosse, una avanti e una indietro, dalla MdT standard.

7.5.2 Macchine multinastro (con num. finito k di nastri)

Una MdT multinastro si presenta nel seguente modo:



quindi ha un controllo finito e un insieme finito di nastri. Ogni nastro è diviso in celle e ogni cella può contenere un simbolo dell'alfabeto finito di nastro; la quale definizione è identica a quella della MdT mononastro. L'insieme degli stati comprende uno stato iniziale degli stati accettanti. All'inizio della computazione, valgono i seguenti criteri:

1. L'input w, una sequenza finita di simboli, si trova sul 1° nastro
2. Tutte le altre cellule di ogni nastro contengono B
3. Il controllo si trova nello **stato iniziale**
4. La testina del 1° nastro si trova all'estremo sinistro dell'input w
5. Tutte le altre testine si trovano in posizione arbitraria; la loro posizione iniziale non importa poiché tutti i nastri eccetto il 1° partono "logicamente vuoti"

Una mossa di una MdT multinastro dipende dallo stato del controllo finito e dai simboli $x_1, x_2, ..., x_k$ letti su ogni nastro. Quindi possiamo definire le operazioni eseguite dalla macchina come segue:

1. Il controllo entra in un nuovo stato, che può coincidere con il corrente
2. Su ogni cella della testina viene scritto un nuovo simbolo appartenente all'alfabeto finito dei simboli di nastro; il quale può coincidere con il corrente
3. Ogni testina si muove a destra o a sinistra, indipendentemente dal movimento delle altre testine.

Quindi una funzione δ ha questa forma:

$$\delta(q, x_1, x_2, ...x_k) = (p, y_1, y_2, ..., y_k, D_1, D_2, ...D_k)$$

Questa estensione è **Turing-equivalente**; infatti è banale mostrare come una MdT multinastro possa simulare una MdT mononastro; tuttavia, è più difficile mostrare la validità del seguente teorema:

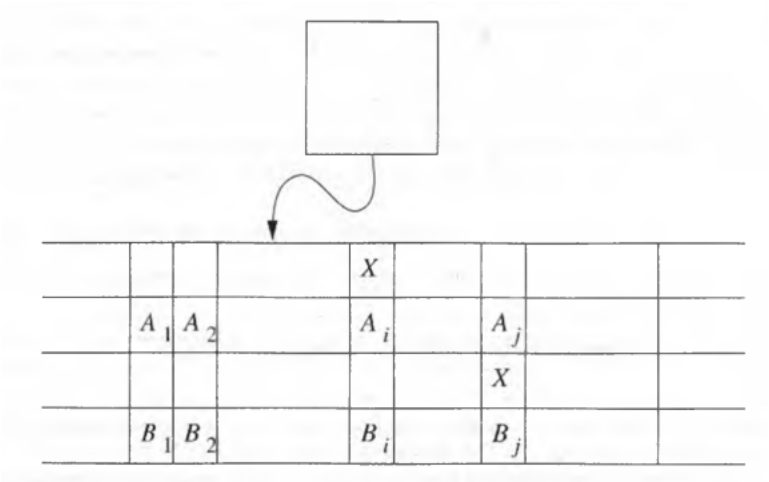
Teorema 7.5.1 *Ogni linguaggio accettato da una MdT multinastro è ricorsivamente enumerabile*

Dimostrazione 7.5.1 *Possiamo simulare una MdT multinastro M con una MdT mononastro N definendo quest'ultima nel seguente modo:*

- *$x_1x_2...x_k$ diventa un nuovo **simbolo di nastro***
- *Definisco quindi N con un alfabeto di simboli di nastro che **contiene le k-uple** dell'alfabeto di simboli di nastro di M (guardo quindi la funzione δ)*
- *La riscrittura di una determinata k-upla sarà un altro simbolo che corrisponderà a $y_1y_2...y_k$*

Questa dimostrazione è applicabile anche se le testine vanno in direzioni diverse.

Dimostrazione 7.5.2 (Libro) *La dimostrazione è mostrata dalla seguente figura:*



*Simuliamo una macchina multinastro M attraverso una MdT mononastro N, il cui nastro è diviso in $2k$ **tracce**. Metà delle tracce sono usate per **replicare i nastri di M**; ognuna delle restanti ospita un **marcatore** che indica la posizione corrente della testina del corrispondente nastro di M. Chiamiamo X questo marcatore (nella figura ipotizziamo $k = 2$).*
*Per simulare una mossa di M, la testina di N deve visitare i k marcatori. Per non smarrisi, N deve tenere conto di **quanti marcatori stanno alla sinistra della testina in ogni istante**; il contatore è **conservato come componente del controllo finito di N**. Dopo aver visitato ogni marcatore e memorizzato, in una componente del controllo, il simbolo nella cella corrispondente, N sa **quali sono i simboli che vengono guardati dalle testine di M**; inoltre N conosce lo stato di M, memorizzato nel suo controllo. Perciò N può dedurre la **prossima mossa di** M.*
*Gli stati accettanti di N sono quelli in cui è **registrato uno stato accettante di** M; in questo modo N accetta solo quando M accetta.*

7.5.3 MdT non deterministiche

Una MdT non deterministica si distingue dalla MdT deterministica per la sua funzione δ:

$$\delta : Q \times \Gamma \to 2^{(Q \times \Gamma \times \{L,R\})}$$

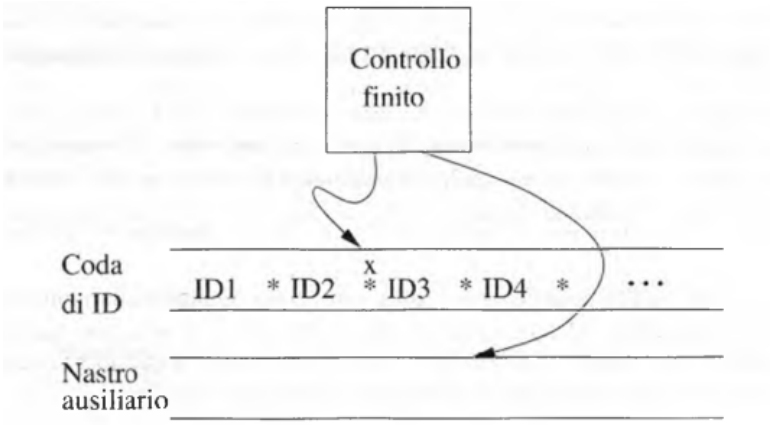
Ogni computazione quindi genera un **albero di computazione**, i cui nodi sono formati dalle **varie configurazioni che la MdT non determinstica assume nei vari casi della** δ.
Come facciamo a scoprire se c'è una configurazione accettante? Come per tutti i modelli di calcolo non deterministici che abbiamo visto, esistono due possibilità:

- Ad ogni scelta non deterministica, la macchina "**si moltiplica**", tuttavia questo metodo è molto inefficiente
- Esiste **un oracolo** che guila la computazione. Possiamo quindi simulare questo oracolo?

Per gli NFA e ε-NFA era possibile convertendoli in DFA. I DPDA **riconscono meno linguaggi** dei PDA. Le MdT non deterministiche **sono Turing-equivalenti**, cioè riconoscono la stessa classe di linguaggi delle MdT deterministiche; infatti una MdT deterministica può essere vista come un **caso particolare di MdT non deterministica** con albero di computazione che presenta un solo nodo figlio per la radice e per tutti i suoi nodi intermedi e una sola foglia. Mostrare il contrario è più complesso:

Teorema 7.5.2 *Se M_N è una MdT non deterministica, esiste una MdT di Turing deterministica M_D t.c. $L(M_N) = L(M_D)$*

Dimostrazione 7.5.3 *Costruiamo la macchina multinastro M_D come segue:*



*La macchina multinastro M_D ha un nastro contenente la **configurazione correntemente considerata** della macchina e un nastro "**coda**" in cui vengono immesse, ad ogni passo di computazione, le possibili prossime configurazioni da analizare. Il 2° nastro quindi funge da **coda** delle prossime configurazioni da analizzare e permette di analizzare l'albero di computazione **in ampiezza**. M_D accetta solo se la configurazione (e quindi il nodo dell'albero di computazione) è accettante.*

7.6 Versioni ristrette delle MdT

Una restrizione di una MdT può essere utile per rendere più semplice la dimostrazione di vari teoremi, purché essa rimanga comunque **Turing-equivalente**. Vediamo quindi ora delle restrizioni della MdT che comunque mantengono questa proprietà:

7.6.1 MdT con nastri semi-finiti

Teorema 7.6.1 *Ogni linguaggio accetato da una MdT M_2 è accettato anche da una MdT M_1 con le seguenti restrizioni:*

1. *La testina di M_1 non va **mai a sinistra** della posizioni iniziale*
2. *M_1 non scrive mai B sul nastro*

Dimostrazione 7.6.1 *È possibile simulare le MdT standard con questa restrizione. Per farlo costruiamo M_1 con un nastro infinito **solo a destra**. Per simulare una MdT standard, prendiamo tutto il nastro a sinistra della posizione iniziale e lo "giriamo" in modo da ottenere un nastro formato nel seguente modo:*

X_0	X_1	X_2	...
*	X_{-1}	X_{-2}	...

*Con $X_0, X_1, ...$ i simboli **a destra della pos. iniziale** e $X_{-1}, X_{-2}...$ i simboli **a sinistra della pos. iniziale**. Il simbolo $*$ indica la fine della traccia e impedisce che la testina "cada" accidentalmente fuori dal bordo del nastro. Definiamo quindi un nuovo alfabeto di simboli del nastro i cui simboli **rappresentano coppie dei simboli del nastro** formate prendendo i primi due simboli delle due "tracce" dall'altro verso il basso; considerando i due simboli come se fossero nella stessa cella. Il nuovo alfabeto sarà quindi formato da simboli rappresentanti coppie (x_i, s). Per esempio, se una cella contiene la coppia (a, b) possiamo assegnarle il simbolo α; se per esempio voglio scrivere c al posto di b, allora la coppia diverrà (a, c) e possiamo assegnare a questa nuova configurazione il simbolo β. Quindi, ad esempio, un possibile nuovo alfabeto di nastro può avere questa forma:*

$$\Gamma_{M_1} = \{(a, *) = a_1, (b, c) = a_2, (i, b) = a_3...\}$$

La 2° condizione si risolve sostituendo ogni scrittura di B con un simbolo B'

7.6.2 Macchine multi-stack

Consideriamo di avere un DPDA con k pile formato nel seguente modo (nell'imagine si ipotizza $k = 3$):

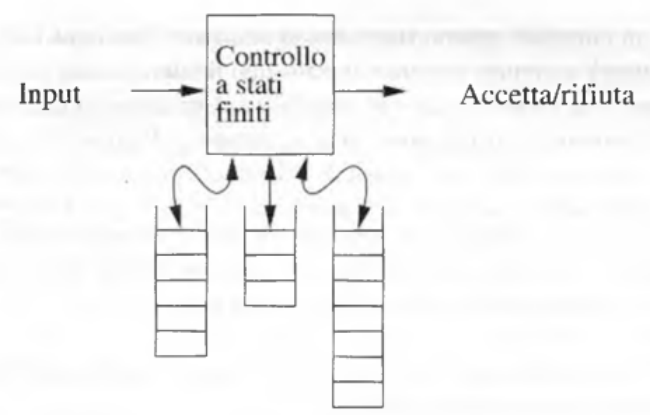


Figura 8.20 Una macchina con tre stack.

Com'è fatta la δ di questo automa? Possiamo immaginarla come segue:

$$\delta(q, a(oppure\ \varepsilon), x_1, x_2, ..., x_k) = (p, \gamma_1, \gamma_2, ..., \gamma_k)$$

Cosa può fare questo tipo di automa? Vediamo il seguente teorema:

Teorema 7.6.2 *Se un linguaggio L è accettato da una MdT, allora L è accettato anche da una macchina con almeno due stack.*

Dimostrazione 7.6.2 *Si sfrutta il fatto che due stack possono* ***simulare il nastro di una MdT****: uno stack conserva ciò che sta a sinistra della testina e uno ciò che sta a destra della testina (sono escluse le infinte sequenze di B). In dettaglio, sia $L = L(M)$ per una MdT M con un nastro. Descriviamo le operazioni che vengono compiute dalla macchina a due stack S:*

1. *S comincia con i due stack "logicamente vuoti", quindi con solo il simbolo iniziale degli stack presente.*

2. *S legge la stringa w di input e la* ***spinge*** *su una pila e poi lo travasa sull'altra: otteniamo una pila vuota e* ***una pila con in cima il carattere puntato dalla testina*** *della MdT all'inizio della computazione.*

3. *L'automa simula un passo della MdT come segue:*

   (a) *S conosce lo stato di M poiché lo simula nel proprio controllo*

   (b) *S controlla cosa c'è in cima alla pila ed effettua un passo di computazione della MdT. In particolare:*

   - *Se M sostituisce X con Y e va a destra, allora S pone Y sul primo stack, a rappresentare che Y ora è a sinistra della testina; X viene tolto dal 2° stack di S. Ci sono due eccezioni:*

   i. *Se contiene solo l'indicatore di fondo,* ***il secondo stack non viene modificato****; poiché vuol dire che la testina si è spostata su un simbolo B*
   ii. *Se $Y = B$ e il 1° stack è vuoto, allora esso rimane vuoto, poiché a sinistra rimangono comunque solo blank.*

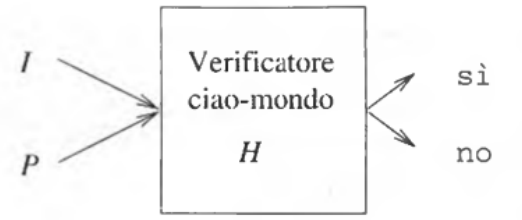
- *Se M sostituisce X con Y e va a sinistra, S toglie il simbolo in cima al primo stack (diciamo Z) e sostituisce X con ZY nel secondo stack. In questo modo si tiene conto del fatto che il simbolo che era immediatamente a sinistra della testina è ora sotto di essa. Si ha una sola eccezione quando Z è il simbolo di fondo: S deve allora porre BY sul secondo stack senza togliere nulla dal primo.*

4. *S accetta solo se si trova in uno stato di M accettante.*

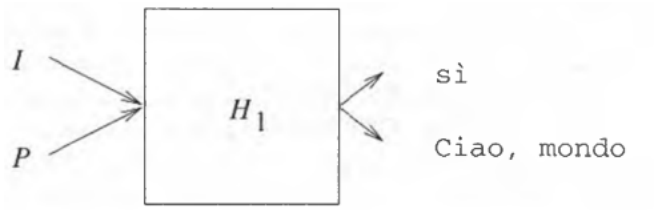
8 Computabilità

8.1 Indecidibilità

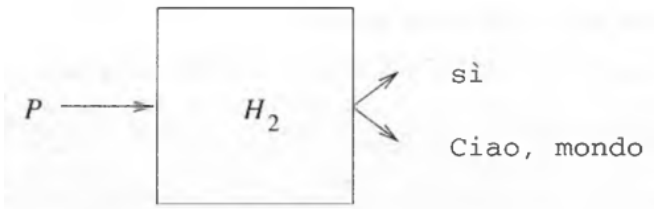
Supponiamo esista un programma H formato in questo modo:



Con P un altro programma e I un input da dare P. Quindi il programma H cerca di **prevedere il comportamento di** P quando riceve un input I e risponde "si" solo se i primi 11 caratteri stampati da P con input I sono "Ciao, mondo" . Se esiste però H allora di sicuro esisterà anche un programma H_1 così costruito:



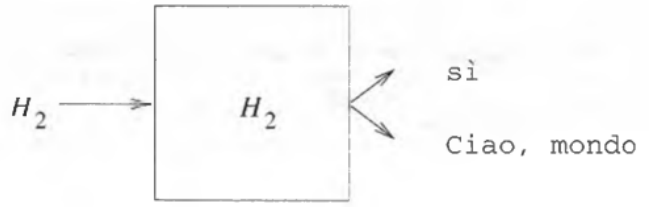
Il quale è una semplice modifica di H; ma se esiste H_1 allora sicuramente esiste H_2 così costruito:



Possiamo immaginare la struttura interna di H_2 così formata:

- Una funzione "copy" che copia il programma di input e lo immagazina in I
- Una componente H_1 che prende in input P e I

Poiché anche H_2 è un programma, allora possiamo fare:



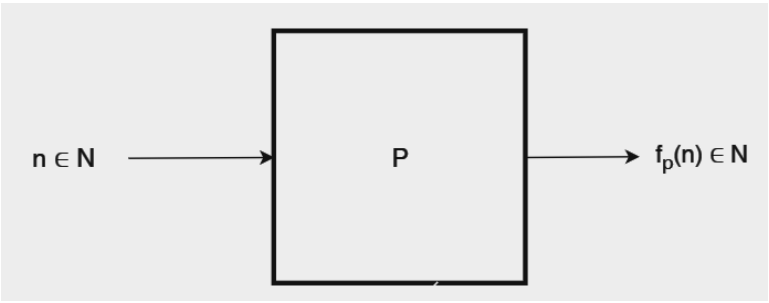
Cosa quindi deve fare H_2? Deve analizzare se stesso e dare una risposta. Tuttavia, se stampa "si", allora **i primi 11 caratteri stampati non sono la stringa "Ciao, mondo"**. Allora dovrebbe stampare "Ciao, mondo", tuttavia allora **essi SONO gli 11 primi caratteri stampati dal programma**. Ci troviamo quindi davanti ad un **paradosso**, e per questo H_2 **non può esistere**. Ma allora neanche H_1 può esistere e allora neanche H esiste.
Quindi, in generale, **non esiste un programma che ne analizza un altro e ne prevede il comportamento**. Le "**prorpietà dinamiche**" (cioè del comportamento dei programmi) sono **indecidibili**. Esistono quindi dei problemi **indecidibili**.

Nota 8.1.1 *Nella versione esposta, il problema contiene* ***autoreferenzialità****: H_2 analizza se stesso. Questa caratteristica è comune anche ad altri paradossi, come, per esempio, il* ***paradosso di Russel***

Un altro paradosso può essere:

"Questa frase è falsa"

Esaminiamo ora un programma P:



Quindi che calcola $f_p : \mathbb{N} \to \mathbb{N}$. Quante sono queste funzioni? Se $|\mathbb{N}| = \aleph_0$ e $|\mathbb{R}| = \aleph_1$ allora le funzioni sono $|\mathbb{N}^{\mathbb{N}}| = \aleph_0^{\aleph_0}$. Quanti sono i programmi P? Poiché possiamo immaginarli come **sequenza di bit**, allora $|Progs| = \aleph_0 = |\mathbb{N}|$. Seppur **non sia mai stato dimostrato**, si pensa che:

$$\aleph_0 << \aleph_0^{\aleph_0}$$

Poiché le funzioni sono quindi **molte di più dei programmi**, allora **esistono funzioni non calcolabili** poiché non vi è un programma P che le calcola. Queste funzioni sono **la maggioranza di tutte le funzioni esistenti**.

8.2 Enumerazione delle stringhe binarie

"Enumerare" significa dare una regola per il quale si possa elencare **in ordine** un insieme di oggetti. La regola che diamo per le stringhe binarie è la seguente:
se $w \in \{0,1\}^*$, consideriamo $1w$ come il numero intero i (scritto in binario), e chiamo w la i-esima stringa binaria e la indico con w_i. L'enumerazione quindi **elenca le stringhe in ordine di lunghezza**:

$$\varepsilon = 1\varepsilon = 1 \leftrightarrow w_1$$

$$0 = 10 = 2 \leftrightarrow w_2$$

$$1 = 11 = 3 \leftrightarrow w_3$$

$$00 = 100 = 4 \leftrightarrow w_4$$

$$...$$

La regola di enumerazione è quindi una **biezione**.

8.2.1 Conversione da MdT a stringa binaria

Codifichiamo una MdT come una **stringa binaria**: tuttavia, la vincoliamo per renderne la gestione più semplice:
Consideriamo una MdT $M = (Q, \underbrace{\{0,1\}}_{\Sigma}, \Gamma, \delta, q_1, B, \{q_2\})$
con:

- $Q = \{q_1, q_2, ..., q_r\}$
- $\Gamma = \{X_1, X_2, ..., X_S\}$ dove $X_1 = 0, X_2 = 1, X_3 = B$

Le direzioni di spostamento non compaiono in questa definizione, ma le codifichiamo nella δ come:

$$L = D_1$$

$$L = D_2$$

Codifichiamo la δ:

$$\delta(q_i, X_j) = (q_k, X_l, D_m)$$

Possiamo quindi codificare un generico case della δ usando questa notazione:

$$(i, j, k, l, m)$$

Con i, j, k, l, m gli indici che appaiono nel generico caso della δ visto sopra. Rappresentiamo quindi la 5-upla ottenuta in binario nel seguente modo:

$$(i, j, k, l, m) \to 0^i 10^j 10^k 10^l 10^m = C_1$$

(quindi, in effetti, gli indici sono **rappresentati in unario**). Cosa succede se abbiamo più casi della δ $C_1, C_2, ..., C_n$? Allora li rappresentiamo come segue:

$$C_1 11 C_2 11 .. 11 C_N \in \{0,1\}^*$$

Questa stringa **rappresenta tutta la δ della MdT** e **rappresenta una MdT**. Poiché abbiamo enumerato le stringhe binarie, in generale una stringa binaria **rappresenta un MdT**? In generale no; possiamo però risolvere questo problema nel seguente modo: definisco una **macchina di default** che **contiene un solo stato e la sua δ è vuota**. La macchina quindi non effettua alcuna operazione e **non accetta mai**, quindi accetta il **linguaggio vuoto**. Ad ogni stringa che non può essere decodificata in una MdT ben formata associo questa macchina.
Viceversa, ogni macchina di Turing è **rappresentabile come una stringa binaria**?

Osservazione 8.2.1 *In generale una MdT può essere rappresentata da più stringhe binarie.*

Effettuando questi accorgimenti, partendo dall'enumerazione delle stringhe binarie, si **costruisce l'enumerazione di tutte le possibili MdT**.
Supponiamo di avere una macchina M (scritta come visto prima) e una stringa binaria w. Se chiamo:

$$cod(M)$$

la stringa che rappresenta la MdT, allora posso costruire:

$$cod(M)111w$$

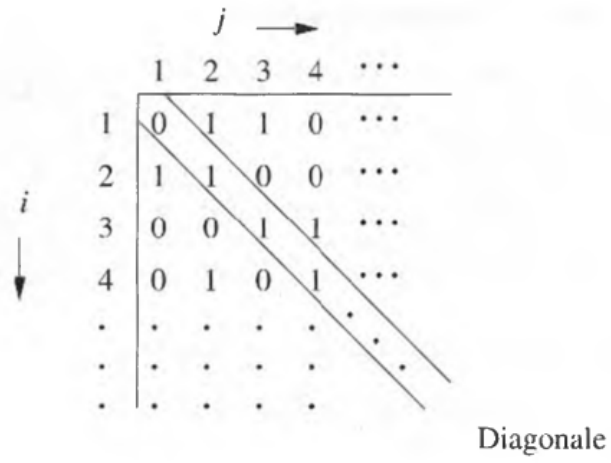
Ottenendo quindi una stringa binaria che **rappresenta la macchina e la sua stringa di input**; questo perché $cod(M)$ **non contiene mai 111**.

8.3 Linguaggio di diagonalizzazione

Indichiamo il **linguaggio di diagonalizzazione** con:

$$L_d = \{w_i \in \{0,1\}^* | w_i \notin L(M_i)\}$$

Le MdT sono in **corrispondenza biunivoca** con le stringhe binarie. Abbiamo quindi che la stringa w_i rappresenta la **macchina di Turing** M_i(se non è una MdT valida, allora M_i è la macchina di default). Costruiamo quindi la seguente tabella infinita:



Con le macchine M_i associate alle righe della tabella e le stringhe di input w_j associate alle colonne della tabella. Se in una cella (i, j) è presente un 1, allora la macchina M_i accetta la stringa w_j, altrimenti la cella contiene 0 (non accetta/va in loop infinito).

Osservazione 8.3.1 *Osservando tutte le colonne j in cui vi è un 1 sulla riga i si ha una **rappresentazione di** $L(M_i)$*

Perché si chiama "di diagonalizzazione"?
Consideriamo la diagonale principale, cioè quindi consideriamo i valori della tabella per cui $i = j$. Consideriamo il **complemento** della stringa formata dai valori della diagonale:

$$L_d = (1, 0, 0, 0...)$$

L_d è il **vettore che rappresenta il linguaggio** L_d poiché contiene un 1 per tutte le $w_i \notin L(M_i)$. Quindi, nella matrice costruita, si ha 1 nella posizione $(i, i) \Leftrightarrow$ **la macchina di Turing M_i accetta se stessa come input**.
Il linguaggio di diagonalizzazione è quindi **l'insieme di tutte le stringhe binarie che non vengono accettate dalle macchine di Turing rappresentate da quelle stringhe**. È quindi formato da tutte le stringhe binarie che **codificano MdT che non accettano se stesse**. Il vettore riportato sopra è quindi **il vettore caratteristico del linguaggio diagonale**, poiché contiene un 1 in ogni posizione dove $w_i \notin L(M_i)$.

Questo linguaggio è Ricorsivamente Enumerabile?

Teorema 8.3.1 *L_d **NON** è un linguaggio ricorsivamente enumerabile. Cioè non esiste nessuna MdT che lo accetta.*

Dimostrazione 8.3.1 *Abbiamo detto che, per definizione, un linguaggio RE è un linguaggio **accettato da una MdT**. Ogni riga della tabella costruita sopra **contiene il vettore caratteristico del linguaggio accettato dalla MdT corrispondente**. Tuttavia, L_d non può comparire nella tabella, poiché ogni suo elemento x_i è **complemento** dell'elemento $k_{i,i}$ della tabella. Quindi L_d **non è ricorsivamente enumerabile***

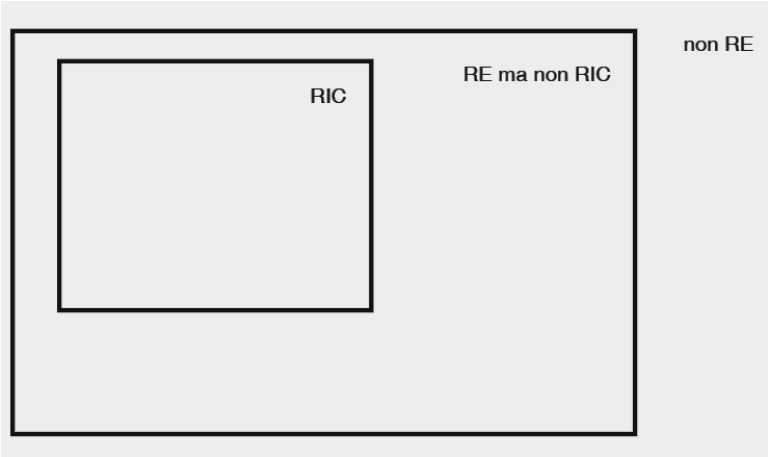
Dimostrazione 8.3.2 (Libro) *Supponiamo $L_d = L(M)$ per una MdT M. Poiché L_d è un linguaggio sull'alfabeto $\{0, 1\}$, M rientra nell'elenco delle MdT che abbiamo costruito, dato che questo elenco include tutte le MdT con alfabeto di input $\{0, 1\}$. Di conseguenza, esiste almeno un codice per M, diciamo i, cioè $M = M_i$. Chiediamoci ora se $w_i \in L_d$:*

- *Se $w_i \in L_d$, allora M_i accetta w_i. Pertanto, per definizione di L_d, w_i non è in L_d, poiché esso contiene solamente le stringhe w_j tali che $w_j \notin L(M_j)$*
- *Se $w_i \notin L_d$, allora M_i non accetta w_i. Pertanto, per definizione di L_d, w_i appartiene a L_d.*

Poiché w_i può non essere ed essere in L_d, abbiamo una contraddizione, quindi una macchina M che accetta L_d non può esistere. Quindi $L_d \notin RE$

8.4 Classificazione dei linguaggi

Presentiamo il seguente schema:



E presentiamo la nuova classe dei linguaggi ricorsivi:

8.4.1 Linguaggi ricorsivi

Supponiamo di avere un linguaggio $L \in RE \backslash RIC$. Allora esiste una MdT M_L t.c. $L = L(M_L)$. Per una stringa di input $w \in \Sigma^*$, la macchina M_L può accettare e fermarsi, non accettare e fermarsi oppure **andare in loop infinito**. Capire se M_L non accetta oppure va in loop infinito è un **problema indecidibile**.
Supponiamo allora di avere $L_1 \in RIC$, allora esiste una MdT M_{L_1} t.c. $L = L(M_{L_1})$. In particolare:

Definizione 8.4.1 *Diremo **ricorsivo** (RIC) un linguaggio L se $L = L(M)$ per una MdT M t.c.:*

1. *se $w \in L$, allora la MdT accetta (e dunque si arresta)*
2. *se $w \notin L$ allora la MdT rifiuta e si arresta.*

I linguaggi ricorsivi sono i linguaggi più vicini alla nozione di algoritmo. In particolare i linguaggi ricorsivi si dicono **decidibili**, quelli RE ma non RIC si dicono **semi-decidibili** e quelli non RE si dicono **indecidibili**.

8.5 Complemento di un linguaggio

8.5.1 Linguaggi ricorsivi

Prendiamo $L_2 \in RIC$. Il complemento di L_2:

$$\overline{L_2} = \{w \in \Sigma^* \backslash L_2\}$$

Inoltre:

Teorema 8.5.1 *Se $L \in RIC$ allora $\overline{L} \in RIC$*

Dimostrazione 8.5.1 *Il teorema si dimostra considerando la macchina M_L, che accetta L e costruendo, basandosi su di essa, la macchina $M_{\overline{L}}$ che accetta il linguaggio $\overline{L}$. In particolare, si scambiano **gli stati finali con quelli non finali**. La nuova macchina costruita accetta il linguaggio $M_{\overline{L}}$ e si fermerà **sempre**, quindi $M_{\overline{L}} \in RIC$*

8.5.2 Ricorsivamente enumerabili ma non ricorsivi

Prendiamo $L_1 \in RE \backslash RIC$. Consideriamo il complemento:

$$\overline{L_1} = \{w \in \Sigma^* \backslash L_1\}$$

Allora $\overline{L_1} \in RE \backslash RIC$? Se fosse così, allora posso costruire una MdT $M_{\overline{L_1}}$ tale che accetti $\overline{L_1}$ ma che abbia la possiblità **di non accettare o di andare in loop infinito**. Ma allora **posso costruire una MdT che simula** sia la macchina $M_{\overline{L_1}}$ sia la macchina M_{L_1} che accetta il linguaggio L_1. In particolare, la nuova macchina simula un passo di computazione delle 2 macchine **in maniera alternata**(cioè prima fa un passo di M_{L_1} poi un passo di $M_{\overline{L_1}}$ e così via). Chiamo questa macchina M_L. Tuttavia, questa macchina **non presenta il rischio di non arrestarsi mai**, poiché simula le 2 macchine in maniera alternata e una delle 2 prima o poi si fermerà (una qualsiasi stringa w appartiene o a L_1 oppure a $\overline{L_1}$, non vi è una terza opzione); quindi M_L è una **macchina che si ferma sempre** e che accetta L_1, quindi $L_1 \in RIC$, quindi, per il **teorema 8.5.1**, anche $\overline{L_1} \in RIC$.
Quindi **non è possibile** che $L_1 \in RE \backslash RIC$ e $\overline{L_1} \in RE \backslash RIC$, poiché altrimenti sarebbero entrambi ricorsivi. Ma allora $\overline{L_1}$ a che famiglia appartiene? $\overline{L_1} \in non\ RE$, quindi è **indecidibile**.

9 Macchine di Turing universali e altre definizioni

Poiché possiamo codificare una MdT in stringa binaria, allora possiamo **costruire una macchina di Turing** che prende in input la descrizione (stringa) di un'altra MdT e una stringa di input per quella macchina e **simuli** la macchina in input sulla stringa in input. Questa macchina quindi può accettare la stringa in input, rifiutare la stringa in input oppure **andare in loop infinito**. Il linguaggio accettato da questa macchina è chiamatao **linguaggio universale** L_U, che possiamo definire come:

$$L_U = \{cod(M)111w \in \{0,1\}^* | w \in L(M)\}$$

Per il comportamento della macchina universale si evince che:

$$L_U \in RE \backslash RIC$$

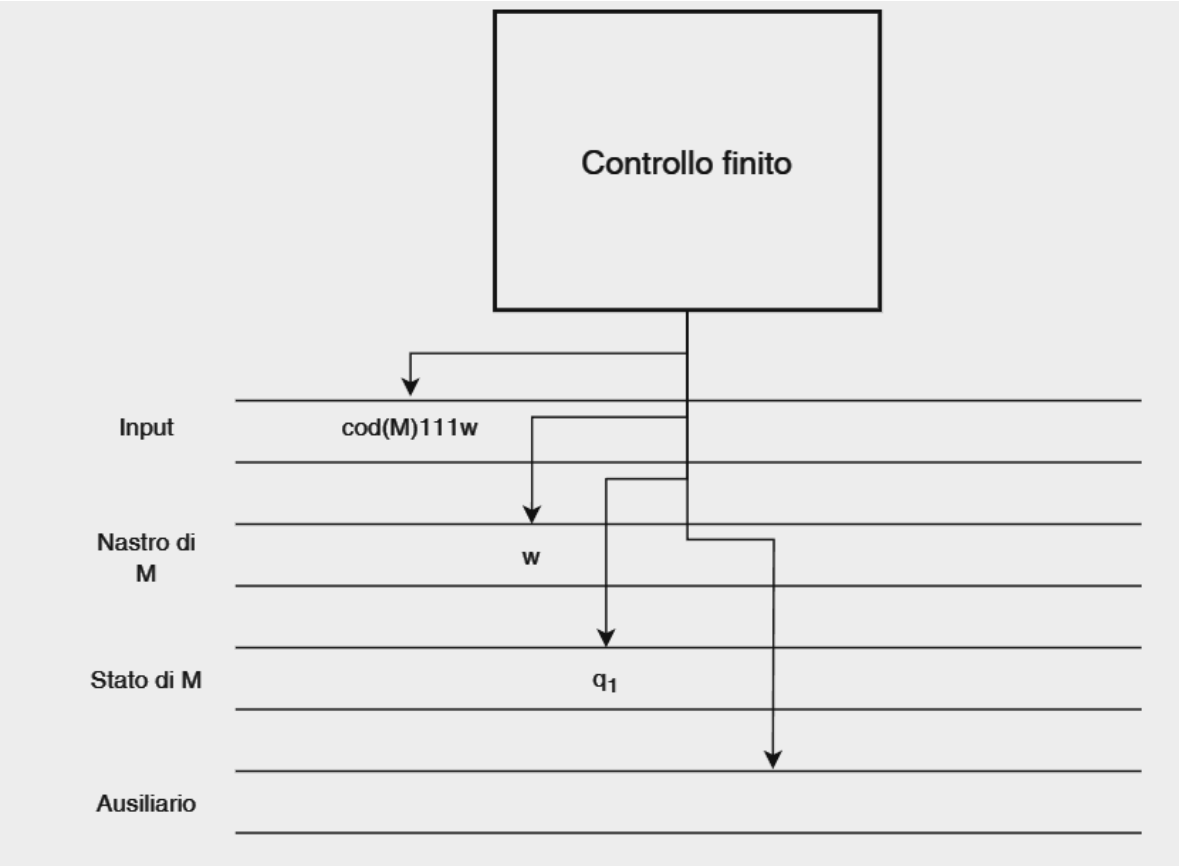
Quindi $\overline{L_U} \in non\ RE$ Presentiamo quindi la struttura e il comportamento della macchina di Turing universale U:

1. All'inizio della computazione, 1° testina è posizionata sul 1° bit di $cod(M)$ e tutti gli altri nastri sono riempiti di B

2. Un passo di computazione **utilizza il 2° nastro** come se fosse il nastro della macchina simulata; quindi cerca w nel 1°nastro e la **copia** nel 2°, quindi torna indietro con la 2° testina, posizionandola sul 1° bit di w.

3. Dopodiché, per definizione, scrive lo stato iniziale q_1 sul 3° nastro.

4. La macchina universale effettua una **scansione** per verificare che **vi sia un caso della δ associato al 1° bit di w e a q_1** e, in caso esista, lo **simula** andando a modificare w, lo stato della macchina simulata M e la posizione della testina del 2°nastro. Questo passo si itera fino a quando la macchina si ferma e accetta, si ferma e non accetta oppure **va in loop infinito**. Quindi simula **perfettamente** M



La macchina universale accetta **il linguaggio universale**.

9.1 Macchine di Turing che accettano il linguaggio vuoto

Consideriamo un alfabeto $\Sigma = \{0, 1\}$, un linguaggio è quindi $L \subseteq \Sigma^*$. Prendiamo ora **l'enumerazione delle macchine di Turing**:

$$M_1 \leftrightarrow w_1$$

$$M_2 \leftrightarrow w_2$$

$$M_3 \leftrightarrow w_3$$

$$M_4 \leftrightarrow w_4$$

...

Quindi $cod(M_i) = w_i$. Formiamo un linguaggio $L = \{w_2, w_5, ..., w_1 0, ...\} \subseteq \{0, 1\}^* = \{M_2, M_5, ..., M_1 0, ...\}$. Allora possiamo formare un linguaggio nel modo seguente:

$$L_E = \{M | L(M) = \emptyset\}$$

Cioè l'insieme delle MdT che **accettano il linguaggio vuoto**. Considero anche il suo complemento:

$$L_{NE} = \{M | L(M) \neq \emptyset\}$$

In particolare, L_E contiene **tutte le copie della macchina di default**. Dove si "trovano" questi linguaggi?

Teorema 9.1.1 *Il linguaggio $L_{NE} \in RE$*

Teorema 9.1.2 *Il linguaggio $L_{NE} \notin RIC$*

Teorema 9.1.3 *Il linguaggio $L_E \notin RE$*

9.2 Proprietà di un linguaggio RE e teorema di Rice

Definizione 9.2.1 *Una proprietà dei linguaggi RE è un insieme di linguaggi RE*

Definizione 9.2.2 *Una proprietà dei linguaggi RE è **banale** se è l'insieme vuoto oppure se è l'insieme che contiene tutti i linguaggi RE*

Definizione 9.2.3 *Una proprietà dei linguaggi RE è **non banale** se non è banale*

Sappiamo che $L \in RE \Leftrightarrow \exists M \text{ t.c. } L = L(M)$. Tuttavia, non possiamo riconoscere un insieme di linguaggi come i linguaggi stessi. La ragione è che il tipico linguaggio è **infinito** e quindi non può essere espresso con una stringa di lunghezza finita che possa essere l'input di una MdT. Dobbiamo piuttosto **riconoscere le MdT che accettano quei linguaggi**: il codice di una MdT è **finito** anche se il linguaggio che accetta è infinito. Di conseguenza, se P è una proprietà dei linguaggi RE, il linguaggio L_P è **l'insieme dei codici delle MdT M_i tali che $L(M_i) \in P$**. Quando quindi parliamo di **decidibilità di una proprietà** P, intendiamo quindi la decidibilità di L_P. Quindi enunciamo il seguente teorema:

Teorema 9.2.1 (Teorema di Rice) *Ogni proprietà non banale dei linguaggi RE è **indecidibile***

10 Ringraziamenti e note

Si ringraziano tutte le persone che hanno contribuito a correggere e stilare questo documento.