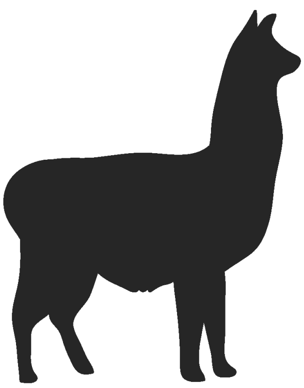
Algoritmi e Principi dell'Informatica

Lorenzo Rossi e tutti coloro che mi hanno aiutato!

AA 2021/2022

Ultimo aggiornamento: 2022-04-19



nessun alpaca è stato ferito nella scrittura di questi appunti

Indice

Introduzione

Questi appunti si riferiscono al corso di *Algoritmi e principi dell'Informatica*, tenuto nell'anno accademico *2021/2022* dal professor *Marco Martinenghi.* La versione più aggiornata può essere trovata sulla repo di GitHub: github.com/lorossi/appunti-di-algoritmi-e-principi-dell-informatica.

Il contenuto di questo documento è tratto parzialmente dalle slide mostrate in aula e dai libri indicati nel manifesto degli studi, (*Informatica Teorica. Mandrioli, Spoletini* e *Introduzione agli algoritmi e strutture dati. Cormen, Leiserson, Rivest, Stein*). Spesso, le illustrazioni e le tabelle saranno uguali *(a meno di palesi errori da parte mia).*

Alcuni argomenti, paragrafi e sezioni potrebbero talora seguire le lezioni, talora seguire i libri, perché mi sono preso la libertà di riordinare il corso a piacimento, in modo che seguisse il mio schema mentale.

Il lettore si senta libero di avvisarmi qualora trovasse un errore! Vi ringrazio in anticipo.

Lorenzo Rossi

1 Linguaggi Formali

Con **linguaggi formali** si intende un'insieme di **stringhe** costruito su un **alfabeto**, secondo uno specifico insieme di regole.
Formalmente:

- **Alfabeto**, o *vocabolario*
  - Insieme **finito** di simboli di base
- **Stringa** su un alfabeto A
  - Sequenza **finita** di simboli dell'alfabeto A
    $\rightarrow$ c'è un ordine tra gli elementi
    $\rightarrow$ non c'è un limite superiore alla lunghezza
  - Sono consentite le **ripetizioni**

1.1 Stringhe

Una **stringa**, è caratterizzata dalla sua **lunghezza**:

- Equivale al numero di simboli che contiene
- La lunghezza della generica stringa x si indica con $|x|$

La stringa **vuota** è una stringa:

- Indicata come ϵ
- Per **definizione**, $|\epsilon| = 0$
- Un insieme formato dalla stringa vuota non corrisponde all'insieme vuoto
  $\rightarrow$ *in simboli:* $\{\epsilon\} \neq \emptyset$

La stringa vuota è definita su qualsiasi alfabeto.

1.1.1 Confronto di stringhe

Date due stringhe

$$x = x_1 x_2 \dots x_n$$
$$y = y_1 y_2 \dots y_m$$

esse si dicono **uguali** se e solo se sono valide le seguenti due proposizioni:

1. Le due stringhe hanno la stessa **lunghezza**

$$|x| = |y| \Leftrightarrow n = m$$

2. Gli elementi in posizioni corrispondenti sono **uguali**

$$x_i = y_i \; \forall \, 1 \leq i \leq n$$

1.1.2 Concatenazione di stringhe

Date due stringhe x e y, la loro **concatenazione** (detta anche *prodotto*) è una stringa xy (*oppure* $x \cdot y$) dove x è seguita da y.

Proprietà della concatenazione:

1. Una stringa x concatenata con ϵ è ancora x *(non viene alterata)*
   $\rightarrow$ $\{\epsilon\} \cdot x = x$
2. La concatenazione è **associativa** e **non commutativa**

$\rightarrow\ a \cdot (x \cdot y) = (a \cdot x) \cdot y$
$\rightarrow\ x \cdot y \neq x \cdot y$

3. Le ripetizioni di un carattere all'interno di una stringa vengono abbreviate tramite elevazione a potenza
   $\rightarrow\ xx \rightarrow x^2$
   $\rightarrow\ yyy \rightarrow y^3$
   $\rightarrow\ yyyyxx \rightarrow y^4x^2$

1.1.3 Sottostringhe

Una stringa x è una **sottostringa** (detta anche *fattore*) di una stringa s se esistono due stringhe y, z

$$y = y_1y_2 \dots y_m$$
$$z = z_1z_2 \dots z_m$$

tali che:

$$s = yxz$$

Proprietà delle sottostringhe:

- sia y che z possono essere ϵ
  - se $y = \epsilon \Rightarrow x$ è detta **prefisso** e s inizia con i caratteri di x
  - se $z = \epsilon \Rightarrow x$ è detta **suffisso** e s finisce con i caratteri di x
- Se $y = \epsilon, z = \epsilon$ allora x è *uguale* a s

1.2 Stella di Kleene

La **stella di Kleene** è un operatore unario che si applica a un insieme di simboli o a un insieme di stringhe. Si indica con il simbolo $*$ e si pronuncia *"A star"*.
Se A è un alfabeto, allora A^* è l'insieme di tutte le stringhe su simboli di A, inclusa la stringa vuota ϵ *a patto che essa faccia parte dell'alfabeto.* Non è imposto un limite superiore alla lunghezza delle stringhe prodotte.

1.2.1 Definizione formale della Stella di Kleene

È possibile definire la stella di Kleene tramite una trattazione più algebrica:

- Un **semigruppo** è una coppia $\langle S, \circ \rangle$ in cui:
  - S è un **insieme** ed è **chiuso** rispetto a $\circ$
  - $\circ$ è un'operazione **associativa** su S
    $\rightarrow$ le operazioni possono essere associate a piacere
    $\rightarrow$ è *distributiva* e *commutativa*
- Un **monoide** è un semigruppo tale per cui:
  - $\exists\, u\, \forall\, x \,|\, x \circ u = u \circ x = x$
  - u è *l'elemento neutro* rispetto all'operazione $\circ$
- Un **gruppo** è un monoide che tale per cui:
  - $\forall\, x\, \exists\, x^{-1} \,|\, x \circ x^{-1} = x^{-1} \circ x = u$
  - x^{-1} è *l'elemento inverso* di x rispetto all'operazione $\circ$

Tra i 3 insiemi è valida la relazione:

$$\textit{semigruppo} \subseteq \textit{monoide} \subseteq \textit{gruppo}$$

A valle di queste definizioni, è possibile inoltre affermare che:

- Dato un semigruppo $\langle S, \circ \rangle$ e un sottoinsieme X di S:
  - X^+ *(detto "più di Kleene")* denota il sottoinsieme di S generato da X, cioè tutte le sequenza della forma
$$x_1 \circ \ldots \circ x_n \quad x_i \in X, n \geq 1$$
  - Per un monoide $\langle S, \circ \rangle$ con unità u:
    - $X^* = X^+ \cup \{u\}$
    - X^* è detto il **monoide libero** generato da X

1.3 Linguaggi

È detto **linguaggio** un qualsiasi insieme di stringhe definite su un alfabeto. Sono linguaggi:

- Italiano, Inglese, Francese, ...
- `C`, `Java`, `Pascal`, ...
- Linguaggi grafici, Musica, Multimedia, ...

Inoltre:

- Una lingua come l'Italiano è *infinita*, perché è possibile scrivere frasi di lunghezza infinita
- Analogamente, un linguaggio come il `C` è un insieme *potenzialmente infinito* poiché l'insieme di programmi corretti è infinito

Formalmente, un linguaggio L definito su un alfabeto A è un *sottoinsieme* di A^*.
I linguaggi formali, contrariamente a quanto possa sembrare, *non sono solo rappresentazioni matematiche astratte.* Essi infatti sono metodi utili a *rappresentare* o *comunicare* una informazione, quindi non solo stringhe senza significato. Usando le operazioni descritte nella Sezione 1.3.1, in base ai vari contesti, è possibile interpretare un linguaggio in modi consoni. Anche i *calcoli* possono essere rappresentati tramite linguaggi formali.
Esistono molti tipi di linguaggi, tra i quali si riconoscono soprattutto:

- Linguaggi *naturali*
- Linguaggi *di programmazione*
- Linguaggi *logici*

Il significato di queste definizioni verrà affrontato in seguito.

1.3.1 Operazioni sui linguaggi

Poiché un linguaggio non è altro che un insieme di stringhe, su di loro si applicano le operazioni insiemistiche. Esse sono:

1. **Unione** - $\cup$
2. **Intersezione** - $\cap$
3. **Differenza** - $\setminus$ *oppure* $-$
4. **Complemento** - L^c
5. **Concatenazione** - $\cdot$
6. **Potenza n-esima** - L^n
7. **Chiusura di Kleene** - L^*

Le operazioni sui linguaggi creano nuove classi di linguaggi. Esempi e le loro rispettive proprietà sono descritte nelle Sezioni seguenti (1.3.1.1 - 1.3.1.6).

1.3.1.1 Unione

Siano L_1, L_2 due linguaggi:

$$L_1 \cup L_2 = \{w \,|\, w \in L_1 \lor w \in L_2\}$$

Esempio: siano L_1, L_2 due linguaggi:

$$L_1 = \{\epsilon, a, b, c, bc, ca\} \qquad L_2 = \{ba, bb, bc, ca, cb, cc\}$$

La loro unione sarà:

$$L_1 \cup L_2 = \{\epsilon, a, b, c, ba, bb, bc, ca, cb, cc\}$$

1.3.1.2 Intersezione

Siano L_1, L_2 due linguaggi:

$$L_1 \cup L_2 = \{w \,|\, w \in L_1 \wedge w \in L_2\}$$

Esempio: siano L_1, L_2 due linguaggi:

$$L_1 = \{\epsilon, a, b, c, bc, ca\} \qquad L_2 = \{ba, bb, bc, ca, cb, cc\}$$

La loro intersezione sarà:

$$L_1 \cap L_2 = \{bc, ca\}$$

1.3.1.3 Differenza

Siano L_1, L_2 due linguaggi:

$$L_1 \setminus L_2 = L_1 - L_2 = \{w \,|\, w \in L_1, w \notin L_2\}$$

È un'operazione che viene generalmente usata quando $L_2 \subseteq L_1$.

Esempio: siano L_1, L_2 due linguaggi:

$$L_1 = \{ba, bb, bc, ca, cb, cc\} \qquad L_2 = \{bc, ca\}$$

La loro differenza sarà:

$$L_1 \backslash L_2 = \{ba, bb, cb, cc\}$$

1.3.1.4 Complemento

Sia L_1 un linguaggio:

$$\neg L_1 = \{w \,|\, w \notin L_1\}$$

Sia L un linguaggio definito su un alfabeto A. Allora la sua operazione di complementazione sarà definita come:

$$L^c = A^* \backslash L$$

1.3.1.5 Concatenazione

Siano L_1, L_2 due linguaggi:

$$L_1 \cdot L_2 = \{wz \,|\, w \in L_1, z \in L_2\}$$

- L'operazione non è commutativa: $L_1 \cdot L_2 \neq L_2 \cdot L_1$
- Si fa scorrere ogni elemento di L_1 associandolo a ogni elemento di L_2
- La concatenazione di un linguaggio con un linguaggio vuoto dà origine al linguaggio stesso, mentre la concatenazione di un linguaggio con un insieme vuoto dà un insieme vuoto:

$$L \cdot \{\epsilon\} = L \quad L \cdot \emptyset = \emptyset$$

- Il numero delle stringhe in $L_1 \cdot L_2$ sarà pari al prodotto del numero di stringhe in ciascun linguaggio:

$$|L_1 \cdot L_2| = |L_1| \cdot |L_2|$$

Esempio: siano L_1, L_2 due linguaggi:

$$L_1 = \{\epsilon, a, b, c, bc, ca\} \qquad L_2 = \{ba, bb, bc, ca, cb, cc\}$$

La loro concatenazione sarà:

$$\begin{aligned} L_1 \cdot L_2 = \{\ & ba, bb, bc, ca, cb, cc, aba, abb, abc, aca, acb, acc, bba, bbb, bbc, bca, \\ & bcb, bcc, cba, cbb, cbc, cca, ccb, ccc, bcba, bcbb, bcbc, bcca, bccb, bccc, \\ & caba, cabb, cabc, caca, cacb, cacc\ \} \end{aligned}$$

1.3.1.6 Potenze

Sia L un linguaggio. La sua potenza L^n sarà ottenuta *concatenando* L con se stesso per n volte.

$$L^n = \underbrace{L \cdot L \cdot \ldots \cdot L}_{n \text{ volte}}$$

Definizione induttiva:

1. $L^0 = \{\epsilon\}$
2. $L^i = L^{i-1} \cdot L$

Si noti che il punto 1 prende il nome di **caso base**, mentre il punto 2 si chiama **passo induttivo**.
Il *passo induttivo* implica che almeno una parte del problema sia stato risolto (tramite **ipotesi induttiva**). In questo caso, l'*ipotesi induttiva* è data dalla relazione che riguarda L^{i-1}.

Esempi:

- $L^2 = L \cdot L$
- $L^3 = L \cdot L \cdot L$
- $L^4 = L \cdot L \cdot L \cdot L$

1.3.1.7 Chiusura di Kleene

Sia L un linguaggio. Allora l'operatore *stella di Kleene* su L è definito come:

$$L^* = \bigcup_{n=0}^{\infty} L^n$$

Analogamente, è possibile definire l'operatore *più di Kleene*:

$$L^+ = \bigcup_{n=1}^{\infty} L^n$$

Proprietà:

1. $L^* = L^+ \cup L^0 = L^+ \cup \{\epsilon\}$
2. $L^+ = L \cdot L^*$
3. $L^+ = L^* \Leftrightarrow \epsilon \in L$

Poiché la concatenazione **non è commutativa**, la chiusura di Kleene **non è riflessiva**.

2 Automi a stati finiti - *FSA*

Un **automa a stati finiti** (o *FSA*, dall'Inglese *Finite State Automaton*) è il più semplice modello di astrazione. Esso rappresenta un sistema che ammette un *insieme finito* di stati (e di conseguenza un numero limitato di configurazioni). In seguito ad un determinato ingresso (a sua volta formato da un insieme finito di valori), potrà avvenire una transizione tra due stati distinti.
Gli *FSA* rappresentano il più semplice modello di computazione: molti dispositivi possono essere modellati come tali, seppure con alcune limitazioni.
Per poter usare gli *FSA* per riconoscere linguaggi, è importante identificare:

1. Le condizioni *iniziali* del sistema
2. Gli stati *finali* del sistema

Affinché sia possibile costruire un modello adatto, è necessario riconoscere gli elementi al suo interno:

- Gli **stati**, tra i quali si identificano:
  - Stato **iniziale**
  - Stati **finali**
- Le **transizioni**
- L'**ingresso**

2.1 Stati, Transizioni e Ingressi

Gli stati sono rappresentati come dei cerchi con all'interno la loro etichetta di riferimento. La loro rappresentazione è illustrata in Figura 1.

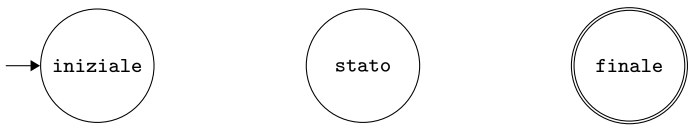


Figura 1: Stati iniziali e finali di un *FSA*

Un *FSA* è definito su un alfabeto. I simboli dell'alfabeto rappresentano l'ingresso del sistema.

Quando il sistema riceve un ingresso, cambia il proprio stato interno. Il passaggio tra stati diversi avviene tramite **transizioni**. Una **transizione** può avere un'etichetta che denomina l'azione che viene intrapresa nel momento della sua attivazione.
Una transizione è rappresentata mediante frecce, come in Figura 2.

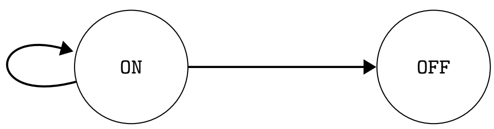


Figura 2: Transizioni in un *FSA*

2.2 Definizione formale *FSA*

Formalmente, un *FSA* è una tupla di 5 elementi $\langle Q, A, \delta, q_0, F \rangle$ dove:

- Q è un insieme di **stati**, finito

- A è l'**alfabeto** di **ingresso**
- δ è la **funzione** di **transizione**:
  - $\delta : Q \times A \to Q$
  - la funzione di transizione è detta **parziale** se non tutte le transizione da tutti i possibili stati per tutti i possibili elementi dell'alfabeto sono definite
  - un *FSA* con una funzione di transizione totale è detto **completo**
- $q_0 \in Q$ è lo **stato iniziale**
- $F \subseteq Q$ è l'insieme di **stati finali**

Si noti che nonostante sia definire **un solo** stato iniziale, un *FSA* ammette **più** stati finali. Inoltre, uno stato può essere contemporaneamente iniziale e finale.

2.2.1 *FSA* con transizione totale

Come già enunciato, una funzione di transizione completa implica che essa sia definita per tutti i possibili stati per ogni possibile elemento dell'alfabeto di ingresso. Il *FSA* in Figura 3 ha una funzione di transizione *totale*, mentre quello in Figura 4 presenta una funzione di transizione *parziale*.

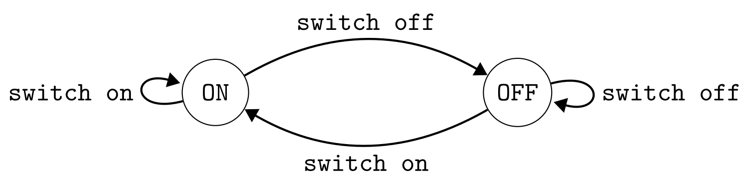


Figura 3: *FSA* con funzione di transizione totale

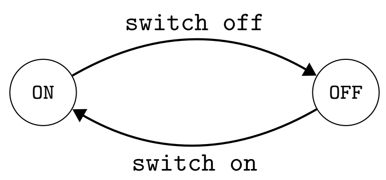


Figura 4: *FSA* con funzione di transizione parziale

Il *FSA* in Figura 4 non mostra la transizione effettuata in corrispondenza dell'azione di `switch off` in corrispondenza di stato `off` e `switch on` sullo stato `on`

2.2.2 Sequenza di mosse

Una **sequenza di mosse** inizia da uno stato iniziale ed è di **accettazione** se raggiunge uno degli stati *finali*.

Formalmente:

- **Sequenza di mosse**:
  - $\delta^* : Q \times A^* \to Q$
  - Il **dominio** è definito come il prodotto cartesiano tra stati e stella di Kleene di sequenze di simboli
  - Il **codominio** coincide con l'insieme degli stati
- δ^* è definita induttivamente a partire da δ:
  1. $\delta^*(q, \epsilon) = q$
  2. $\delta^*(q, yi) = \delta\left(\delta^*(q, y), i\right)$
- Stato **iniziale** $q_0 \in Q$
- Stati **finali** $F \subseteq Q$

2.2.3 Condizione di accettazione di un *FSA*

Il linguaggio relativo al *FSA* è costituito dalle stringhe x che appartengono a δ^*. *Formalmente*:

$$\forall\, x \in L \Leftrightarrow \delta^*(q_0, x) \in F$$

I linguaggi riconosciuti dagli *FSA*, come verrà analizzato più in dettaglio in seguito *(per esempio nella Sezione 6.2)*, prendono il nome di **regolari**.

2.3 Trasduttori a stati finiti - *FST*

Idea: usare gli automi come traduttori di linguaggi. Nasce il **FST** *(dall'Inglese finite state transducer)*, cioè un *FSA* che lavora su due nastri. È una specie di macchina traduttrice.
La rappresentazione più semplificata di un *FST* è rappresentata in Figura 5.

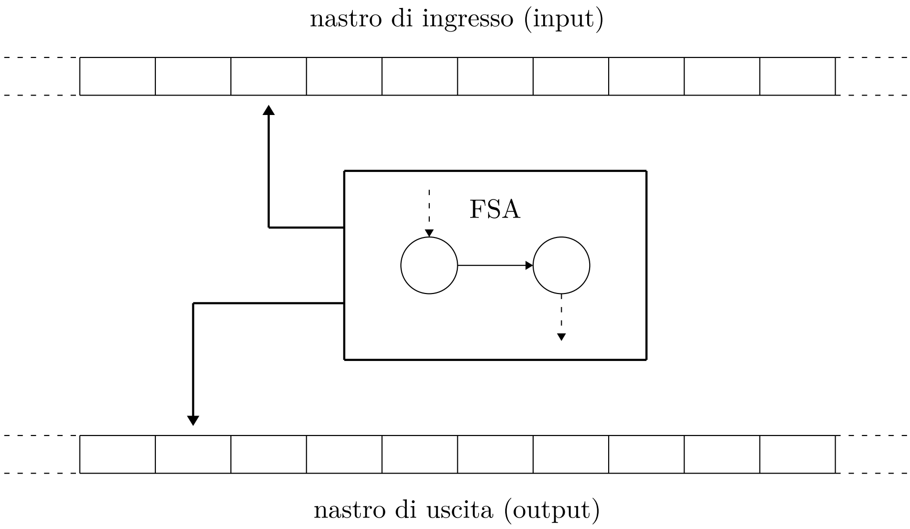


Figura 5: Diagramma semplificato di un FST

Un *FST* è composto da:

1. Una stringa di **ingresso** x
2. Una stringa di **uscita** y
3. Una funzione $\tau : L_1 \to L_2$ tale che $y = \tau(x)$

2.3.1 Definizione formale *FST*

Un trasduttore a stati finiti (*FST*) è una tupla di 7 elementi $\langle Q, A, \delta, q_0, F, O, \eta \rangle$:

- Q è un insieme finito di **stati**
- A è l'**alfabeto** di **ingresso**
- δ è la **funzione** di **transizione**:
  - $\delta : Q \times A \to Q$
- $q_0 \in Q$ è lo **stato iniziale**
- $F \subseteq Q$ è l'insieme di **stati finali**
- O è l'**alfabeto** di **uscita**
- η è la **funzione** di **uscita**:
  - $\eta : Q \times I \to O^*$
  - il **dominio** è il prodotto cartesiano degli stati per l'alfabeto di ingresso
  - il **codominio** è la stella di Kleene dell'alfabeto di uscita

La definizione di Q, A, δ, q_0, F è analoga a quella avvenuta nella Sezione 2.2 riguardante gli *FSA*.
La condizione di accettazione resta la stessa degli accettori. Di conseguenza, **la traduzione avviene solo su stringhe accettate**.

2.3.2 Traduzione di una stringa

Analogamente a quanto già illustrato nella definizione della sequenza di mosse δ (Sezione 2.2.2), η^* verrà definito induttivamente. Infatti, ricordando che $\eta^* : Q \times I^* \to O^*$:

1. $\eta^*(q, \epsilon) = \epsilon$
2. $\eta^*(q, y \cdot i) = \eta^*(q, y) \cdot \eta\left(\delta^*(q, y), i\right)$

sarà quindi valida la relazione:

$$\forall\, x, \tau(x) = \eta^*(q_0, x) \Leftrightarrow \delta^*(q_0, x) \in F$$

2.3.3 Pumping lemma

Si consideri il *FSA* in Figura 6.

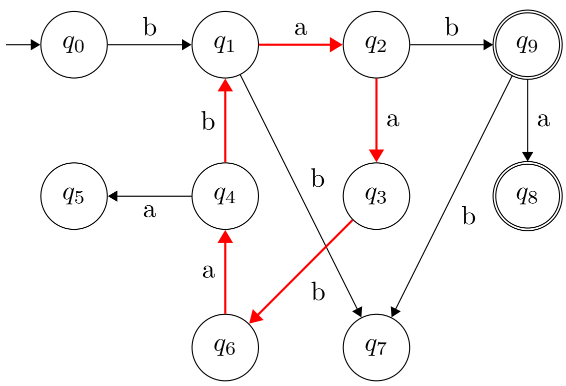


Figura 6: Pumping lemma

Se si ammette la possibilità che il ciclo $q_1 \to q_2 \to q_3 \to q_6 \to q_4 \to q_1 \to \dots$ venga attraversato una volta, allora è possibile che esso venga attraversato un numero indefinito *(potenzialmente infinito)* di volte.
Più formalmente:

- Se $x \in L, |x| \geq |Q|$, allora esistono uno stato $q \in Q$ a una stringa $w \in I^+$ tali che:
  - $x = ywz$
  - $\delta^*(q, w) = q$
- Perciò varrà anche $\forall\, n \geq 0\ yw^n z \in L$

Questo fenomeno prende il nome di **pumping lemma**, e porta a due possibili *conseguenze*:

1. $L = \emptyset$
   - $\exists\, x \in L \Leftrightarrow \exists\, y \in L,\ |y| < |Q|$
   - È sufficiente rimuovere tutti i *cicli* dall'*FSA* che accetta x
2. $L = \infty$
   - Il *FSA* deve verificare in modo analogo se $\exists\, x \in L,\ |Q| \leq |x| < 2 \cdot |Q|$
   - *La lunghezza della stringa x è tale da compiere meno di due cicli*

Nella pratica, ciò porta alle seguenti *implicazioni*:

- Un linguaggio di programmazione che ammette 0 programmi corretti **non è rilevante**
- Analogamente, un linguaggio di programmazione che ammette *un numero finito* di programmi corretti **non è rilevante**

⇒ Il *pumping lemma*, **quindi, non unicamente è un aspetto negativo** poiché permette di creare linguaggi **infiniti**

2.3.3.1 Conseguenza negativa del *pumping lemma*

Si consideri il linguaggio
$$L = \{a^n b^n \mid n > 0\}$$
e si supponga che sia riconosciuto da un qualsiasi *FSA*.

Si consideri ora la stringa
$$x = a^m b^m, \ m > |Q|$$
e si applichi il *pumping lemma*. Come conseguenza dello stesso, poiché la lunghezza della stringa è superiore al numero di stati, dovrebbe esistere una costante k per la quale, se $x \in L, |x| \geq k$, la stessa x può essere scritta come ywz con $1 \leq |w| \leq k, yw^i z \in L \,\forall\, i \geq 0$.
La dimostrazione avviene *per assurdo*. Scomponendo x, si possono identificare 3 possibili forme di essa che verranno accettate:

- $x = ywz, w = a^r, r > 0$, quindi anche $a^{s+k\cdot r} b^n \,\forall\, k \geq 0$ **dovrebbe essere accettato**
- $x = ywz, w = b^r, r > 0$, quindi anche $a^n b^{s+k\cdot r} \,\forall\, k \geq 0$ **dovrebbe essere accettato**
- $x = ywz, w = a^r b^s, r > 0, s > 0$, quindi anche $a^{n-r}(a^r b^s)^k b^{n-s} \,\forall\, k \geq 0$ **dovrebbe essere accettato**

Tutte e tre queste forma, tuttavia, violano la forma di x indicata nell'ipotesi. Quindi si può affermare che esistono linguaggi non riconoscibili tramite *FSA*.
Più precisamente, esistono dei *linguaggi infiniti* che non possono essere riconosciuti dagli *FSA*. Questa affermazione può essere giustificata informalmente affermando che *"gli FSA non possono contare"*.
La famiglia dei linguaggi che sono riconosciuti dagli *FSA* prende il nome di **regolare**.

2.3.3.2 Provare che un linguaggio sia infinito

Per poter provare che:

- Il linguaggio riconosciuto dal *FSA* sia infinito, bisogna provare le infinite stringhe che appartengono al linguaggio su cui esso è definito
- Il linguaggio riconosciuto dal *FSA* sia vuoto, bisogna provare le infinite stringhe che appartengono al linguaggio su cui esso è definito

Non è possibile effettuare un test con un numero infinito di stringhe in un tempo finito. Quindi, per effettuare una ricerca esaustiva e trovare una risposta a questa domanda, si provano le stringhe strettamente inferiori al numero di stati:

⇒ Se una stringa **viene accettata**, allora **tutte** le stringhe verranno accettate
⇒ Se una stringa **non viene accettata**, allora **nessuna** stringa verrà accettata

Il numero di queste stringhe, e quindi il numero di test da effettuare, è limitato.

2.3.3.3 Provare la presenza di un ciclo

Se un automa accetta un certo linguaggio, allora una stringa di lunghezza superiore al numero di stati verrà accettata se e solo se è presente un ciclo all'interno del corrispondente *FSA*.

2.4 Operazioni sugli *FSA*

Prima di poter definire le operazioni sugli *FSA*, è necessario definire il concetto di **chiusura**:
Chiusura in *matematica*: un insieme S è **chiuso** rispetto a una operazione OP se, quando OP è applicata agli elementi di S, il risultato è ancora un elemento di S
Leggermente diverso è il caso dei *linguaggi*, che verrà esaminato più nel seguente paragrafo.

2.4.1 Chiusura per i linguaggi

Siano:

- $\mathcal{L} = \{L_i\}$ una *famiglia* di **linguaggi**
  - $\mathcal{L}$ è chiuso rispetto all'operazione OP se e solo se, $\forall\, L_1, L_2 \in \mathcal{L},\ L_1\ OP\ L_2 \in \mathcal{L}$
- $\mathcal{R}$ la *famiglia* dei **linguaggi regolari**
  - $\mathcal{R}$ è chiuso rispetto alle operazioni insiemistiche, alla concatenazione e all'operatore $*$ *(stella di Kleene)*

2.4.1.1 Intersezione

Siano A_1 e A_2 due generici *FSA*, caratterizzati dalle loro espressioni

$$A_1 = \langle Q_1, I_1, \delta_1, q_{01}, F_1 \rangle$$
$$A_2 = \langle Q_2, I_2, \delta_2, q_{02}, F_2 \rangle$$

Si supponga che $I_2 = I_1 = I$ e che $Q_1 \cap Q_2 = \emptyset$. Entrambe le supposizioni non comportano alcuna perdita di generalità perché:

1. Se I_1 ed I_2, fossero diversi, posto che la funzione di transizione non deve essere necessariamente uguale, i due atomi possono essere considerati definiti su $I_1 \cup I_2$
2. Se Q_1 e Q_2 non fossero disgiunti, i nomi degli stati comuni possono essere banalmente cambiati

FSA che riconosce entrambi i linguaggi (e quindi $I_1 \cap I_2$) è costruito come:

$$C = \langle A_1, A_2 \rangle = \langle Q_1 \times Q_2, I, \delta, \langle q_{01}, q_{02} \rangle, F_1 \times F_2 \rangle$$
$$\delta(\langle q_1, q_2 \rangle, i) = \langle \delta_1(q_1, i), \delta_2(q_2, i) \rangle$$

La dimostrazione che

$$L\left(\langle A_1, A_2 \rangle\right) = L\left(A_1\right) \cap L\left(A_2\right)$$

può avvenire un ragionamento per induzione alla funzione δ.

Intuitivamente, l'*esecuzione parallela* di due *FSA* può essere simulata *accoppiandoli* tramite prodotto cartesiano. L'intersezione di due *FSA* può essere vuota, non riconoscendo alcun linguaggio (ricordando che $\emptyset \neq \{\epsilon\}$).

Esempio: siano A e B due *FSA* rappresentati come segue:

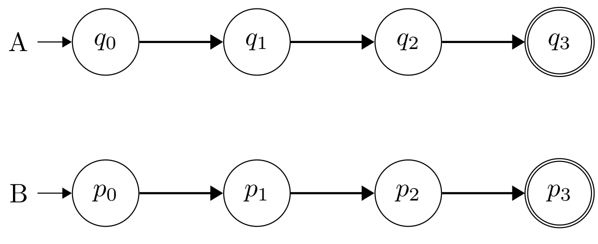


Figura 7: Intersezione tra FSA

La loro *intersezione* $C = A \cap B$ può essere rappresentata come segue:

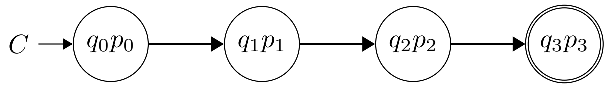


Figura 8: Intersezione tra FSA

2.4.1.2 Unione

Analogamente a quanto avviene nell'intersezione (Sezione 2.4.1.1), con le stesse ipotesi, l'unione di due *FSA*:

$$A_1 = \langle Q_1, I_1, \delta_1, q_{01}, F_1 \rangle$$
$$A_2 = \langle Q_2, I_2, \delta_2, q_{02}, F_2 \rangle$$

La loro unione sarà data da:

$$C = \langle A_1, A_2 \rangle = \langle Q_1 \times Q_2, I, \delta, \langle q_{01}, q_{02} \rangle, F_1 \times Q_2 \cup Q_1 \times F_2 \rangle$$
$$\delta(\langle q_1, q_2 \rangle, i) = \langle \delta_1(q_1, i), \delta_2(q_2, i) \rangle$$

2.4.1.3 Complemento

Si consideri il *FSA*:

$$A = \langle Q, I, \delta, q_0, F \rangle$$

Per poterlo complementare, è prima necessario costruire un *FSA* A' aggiungendo un nuovo stato $\overline{q}$ ad A in modo che la funzione di transizione di A' conduca a $\overline{q}$ ogni qualvolta che è indefinita in A. Si imponga inoltre che l'automa, una volta in $\overline{q}$, ci rimanga per qualsiasi simbolo di ingresso.
In questo modo, la funzione di transizione di A (e quindi la sua corrispondente δ^*) è totale.
Proprietà del complemento:

- Se l'intera stringa di ingresso viene scandita, allora per complementare il risultato basta *scambiare il* `si` *con il* `no`
- Se la fine della stringa non viene raggiunta, allora lo scambio di F con $Q - F$ non funziona
- Nel caso degli *FSA* il *"trucco"* consiste nel completare la funzione di transizione δ
- In generale, *non è possibile considerare equivalenti la risposta negativa a una domanda e la risposta positiva della domanda opposta*

Grazie a questa operazione, è possibile definire l'unione di due *FSA* (Sezione 2.4.1.2) tramite le leggi di *De Morgan*:

$$L(A_1) \cup L(A_2) = \overline{\overline{L(A_1)} \cap \overline{L(A_2)}} = \left(L(A_1)^C \cup L(A_2)^C\right)^C$$

con il medesimo risultato.

3 Pushdown automaton - *PDA*

Come mostrato prima nell'applicazione del *pumping lemma* al linguaggio $L = \{a^n b^n \mid n > 0\}$ (Sezione 2.3.3.1), dimostra l'impossibilità di riconoscere alcuni linguaggi tramite *FSA*, a causa della loro inabilità nel contare una quantità di simboli sconosciuta a priori.
A causa di questa mancanza, essi sono inadatti al riconoscimento di molti linguaggi di interesse pratico. Per ovviare a questo limite, vengono introdotti i *PDA* (in Italiano *automi a pila* or *AP*), il cui diagramma semplificato è mostrato in Figura 9.

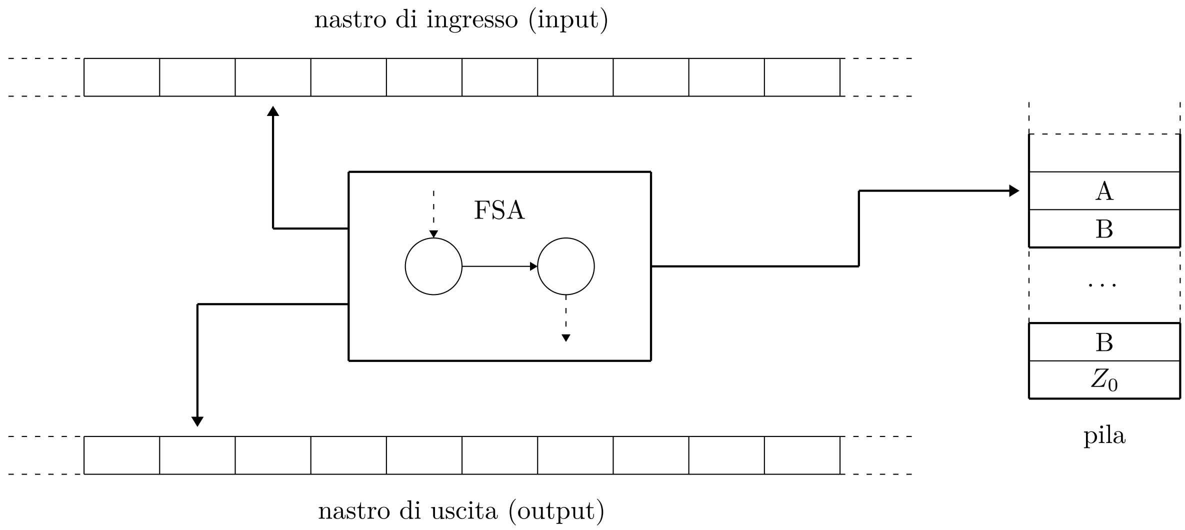


Figura 9: Diagramma semplificato di un PDA

La **pila** è una forma di memoria in cui:

- I nuovi simboli sono **inseriti in cima**
- La pila viene **letta dalla cima**
- Un simbolo letto viene **estratto dalla cima**

  $\rightarrow$ si attua una politica `FIFO`
- L'ultimo elemento in basso della pila è occupato da un particolare simbolo Z_0

I *PDA* differiscono dagli automi a stati finiti in due modi:

1. Possono usare la cima della pila per decidere quale transizione effettuare
2. Possono manipolare la pila durante una transizione

3.1 Definizione formale *PDA*

Formalmente, un *PDA* é una tupla di 7 elementi $\langle Q, A, \Gamma, \delta, q_0, Z_0, F \rangle$, dove:

- Q è un **insieme di stati**, finito
- A è l'**alfabeto** di ingresso
- Γ è l'**alfabeto** di pila
- δ è la funzione di **transizione**:
  - $\delta : Q \times (I \cup \{\epsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$
  - Il **dominio** è definito come il prodotto cartesiano tra stati, alfabeto di ingresso unito all'elemento nullo e alfabeto di pila
  - Il **codominio** è definito come il prodotto cartesiano tra stati e *stella di Kleene* dell'alfabeto di pila
- $Z_0 \in \Gamma$ è il **simbolo iniziale** (il primo in *basso*) di pila
- $q_0 \in Q$ è lo **stato iniziale**

- $F \subseteq Q$ è l'insieme di **stati finali**, finito

Si noti che la definizione di Q, A, δ, q_0, F sono analoghi a quanto illustrato in Sezione 2.2 riguardo gli *FSA*.

3.2 Mosse di un *PDA*

In base a:

- Il simbolo **letto** dall'ingresso *(opzionalmente)*
- Il simbolo **letto** dalla cima della pila
- Lo **stato** del dispositivo di controllo

il *PDA*:

- Cambia il proprio **stato**
- Sposta in avanti la **stato**
- Sostituisce il simbolo letto dalla **pila** con una stringa α *eventualmente vuota*

Sono anche ammesse delle delle mosse *spontanee* (dette ϵ-mosse) che avvengono anche senza che venga letto il simbolo dalla stringa, ignorandolo. Per queste mosse la funzione di transizione è definita come:

$$\delta(q, \epsilon, A) = \langle p, \alpha \rangle$$

Se una ϵ-mossa è stata definita per una coppia $\langle \overline{Q}, \overline{A} \rangle$ *(stato e simbolo in cima alla pila)*, non è possibile definire un'altra mossa non-ϵ per lo stesso simbolo $\overline{A}$ dallo stesso stato $\overline{Q}$.
Gli *FSA* non hanno ϵ-mosse, in quanto esse sono una caratteristica dei *PDA*. Essi non implicano che il carattere in cima alla stringa sia ϵ ma che la stringa non viene letta.

3.3 Accettazione di una stringa

Alla fine della computazione, la stringa di ingresso x è *riconosciuta* (o *accettata*) se valgono entrambe le condizioni:

1. Il *PDA* la legge **completamente**
2. Il *PDA* si trova in uno stato di accettazione (**finale**) quando la fine di x è stata raggiunta

Non ci sono condizioni sulla pila sia vuota affinché la stringa x sia accettata. In particolare, **non è necessario che la pila sia vuota** alla fine della computazione *(anche se è una condizione che si può verificare)*.

3.4 Configurazione di un *PDA*

Una **configurazione** è una generalizzazione della nozione di stato. Essa mostra:

- Lo **stato corrente** del dispositivo di controllo
- La **porzione di stringa** di ingresso a destra dalla testina
  - indica la parte di essa ancora non letta
  - ciò chè è già stato letto non è rilevante poiché è già stato consumato
- Il **contenuto** della pila

Può essere vista come una *istantanea* del *PDA*, mostrando nel tempo il suo stato interno.

Formalmente, la configurazione c è una tripla $\langle q, x, \gamma \rangle$:

- $q \in Q$ è lo **stato corrente** del dispositivo di controllo
- $x \in I^*$ è la **porzione non letta** della stringa di ingresso
- $\gamma \in \Gamma^*$ è la **stringa di simboli** nella pila.

3.5 Transizioni di un *PDA*

Le **transizioni** tra configurazioni dipendono dalla funzione di transizione e si indicano con il simbolo $\vdash$. Una sequenza di transizioni è indicata col simbolo $\vdash^*$.
Esse illustrano come commutare tra una *istantanea* e la sua successiva. Per un dato *PDA* A, la transizione $\vdash_A$ nello spazio di tutte le possibili configurazioni di A è definita da

$$c = \langle q, x, \gamma \rangle \vdash_A c' = \langle q', x', \gamma' \rangle$$

solo se vale una delle due condizioni:

1. La funzione di transizione è definita per un simbolo di ingresso

$$\begin{aligned} x = ay &\mapsto \; x' = y \\ \gamma = A\beta &\mapsto \gamma' = \alpha\beta \\ \delta(q, a, A) &= \langle q', \alpha \rangle \end{aligned}$$

2. La funzione di transizione è definita per una ϵ-mossa

$$\begin{aligned} x &\mapsto x', \\ \gamma = A\beta &\mapsto \gamma' = \alpha\beta \\ \delta(q, \epsilon, A) &= \langle q', \alpha \rangle \end{aligned}$$

3.5.1 Nota sulle transizioni

Si consideri un generico *PDA* con le caratteristiche già indicate, sia δ la sua funzione di transizione.
Si indica con il simbolo $\perp$ il risultato di una transizione che **non è definita** per i parametri indicati.
Allora:

- Una ϵ-mossa è una mossa **spontanea**:
  - se $\delta(q, \epsilon, A) \neq \perp$ *(δ non è indefinita)* e A è il simbolo in cima alla pila, la transizione **può sempre essere eseguita**
- Se $\delta(q, \epsilon, A) \neq \perp$, allora $\forall\, i \in I, \delta(q, i, A) = \perp$
  - se questa proprietà non fosse soddisfatta, entrambe le transizioni **sarebbero consentite**
  - questa condizione **garantisce il determinismo** poiché non sarebbe possibile definire unicamente quale delle più transizioni scegliere partendo dallo stato
  - più avanti verrà mostrato il funzionamento degli automi in assenza di determinismo (Sezione 5)

3.5.2 Notazione grafica delle transizioni di un *PDA*

Il diagramma mostrato in Figura 10 mostra una transizione di un *PDA* da uno stato q_0 a uno stato p_0.

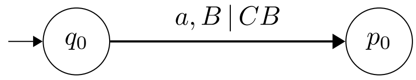


Figura 10: Notazione grafica dei *PDA*, *push*

La notazione implica che se:

1. Il *PDA* si trova nello **stato** q_0
2. Il **carattere** a viene letto dalla stringa di input
3. Il **carattere** B è in cima alla pila

allora succederanno i seguenti eventi:

1. Il carattere C viene **aggiunto** in cima alla pila *(operazione di push)*

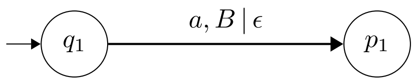


Figura 11: Notazione grafica dei *PDA*, *pop*

2. Lo stato attuale diventa p_0

Il diagramma mostrato in Figura 11 mostra una transizione di un *PDA* da uno stato q_1 a uno stato p_1.
La notazione implica che se:

1. Il *PDA* si trova nello **stato** q_1
2. Il **carattere** a viene letto dalla stringa di input
3. Il **carattere** B è in cima alla pila

allora succederanno i seguenti eventi:

1. Il carattere B viene **rimosso** dalla pila *(operazione di pop)*
2. Lo stato attuale diventa p_1

3.6 Condizione di accettazione di un *PDA*

Sia $\vdash^*$ la *chiusura riflessiva* e *transitiva* della relazione $\vdash$. La condizione di accettazione è:

$$\forall\, x \in I^* \mid x \in L \Leftrightarrow c_0 = \langle q_0, x, Z_0 \rangle \vdash^* c_F = \langle q, \epsilon, \gamma \rangle,\ q \in F$$

in cui:

1. La configurazione iniziale parte dallo **stato iniziale** con la **stringa ancora interamente da leggere** e la **pila vuota**
2. Viene applicato un certo numero di **transizioni**
3. La configurazione finale ha la **stringa completamente letta** e lo **stato attuale è nell'insieme degli stati finali** mentre non ci sono condizioni sullo stato della pila

Informalmente, una stringa è accettata da un *PDA* se esiste un cammino coerente con x su di esso che va dallo stato iniziale allo stato finale. La stringa deve essere letta in tutta la sua interezza.

3.7 *PDA* vs *FSA*

Alcuni linguaggi, a causa del *pumping lemma*, non possono essere riconosciuti da un *FSA*. Essi tuttavia possono essere riconosciuti da un *PDA* e prendono il nome di **linguaggi regolari**.
I *PDA* sono quindi **più espressivi** degli *FSA*.
Infatti, dato un *FSA* $A = \langle Q, I, \delta, q_0, F \rangle$ è immediato costruire un *PDA* $A' = \langle Q', i', \Gamma', q_0', Z_0', F' \rangle$ tale che $L(A) = L(A')$.
Informalmente, i *PDA*, rispetto agli *FSA*, hanno la capacità di invertire le stringhe di lunghezza indefinita e contare, proprio grazie alla loro memoria a pila.

3.8 *PDA* ciclici e aciclici

A differenza degli *FSA*, i *PDA* potrebbero non fermarsi dopo un numero finito di mosse: sono possibili **cicli** di ϵ-mosse. Tuttavia, i *PDA* ciclici non aggiungono potere espressivo alla classe dei *PDA* e questa loro caratteristica potrà essere rimossa.
Formalmente, si consideri un *PDA* A. La notazione

$$\langle q, x, \alpha \rangle \vdash_d^* \langle q', y, \beta \rangle$$

indica che:

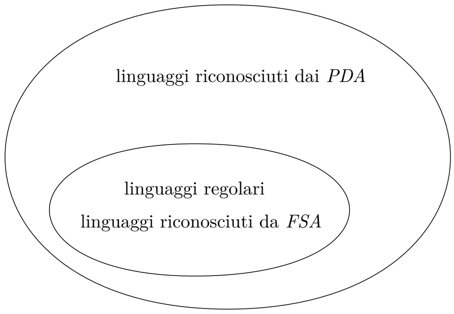


Figura 12: Linguaggi riconosciuti da *PDA* e *FSA*

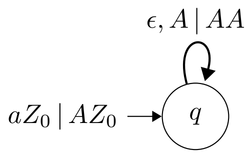


Figura 13: *PDA* ciclico

1. Il *PDA* evolve dalla *configurazione iniziale* $\langle q, x, \alpha \rangle$ alla configurazione $\langle q', y, \beta \rangle$ tramite operatore $\vdash^*$
2. Per $\beta = Z\beta'$, $\delta(q', \epsilon, Z) = \perp$ (è quindi *indefinita*)
    - $\vdash^*_d$ è una sequenza di mosse che porta a una configurazione da cui **non è possibile procedere** con una ϵ-mossa
    - per evolvere da questa configurazione è necessario leggere un simbolo in ingresso

Ciò detto, un *PDA* è **aciclico** se e solo se

$$\forall\, x \in I^* \; \exists\, (q, y) \mid \langle q_0, x, Z_0 \rangle \vdash^*_d \langle q, \epsilon, \gamma \rangle$$

e che quindi:

1. Legge sempre l'intera stringa di ingresso e, al termine di essa, si ferma
2. Non può ripetere un ciclo indefinitamente con ϵ-mosse
3. Si ferma dopo un numero finito di mosse

Ogni *PDA* ciclico può (e deve) essere trasformato nel proprio equivalente aciclico. Un *PDA* che presenta cicli, infatti, potrebbe non raggiungere mai la fine della stringa e il suo corrispondente stato di accettazione, rimanendo per sempre in un ciclo di ϵ-mosse.

3.9 Operazioni sui *PDA*

Nelle successive Sezioni (3.9.1-3.9.4) ci si occuperà di trattare la chiusura dei linguaggi riconosciuti da *PDA* rispetto a varie operazioni.

3.9.1 Complemento

La classe dei linguaggi riconosciuti da *PDA* **è chiusa** rispetto alla complementazione. Il complemento può essere costruito:

- Eliminando i cicli
- Aggiungendo stati di errore
- Scambiando stati finali con non finali
- Occupandosi di ϵ-mosse che collegano stati finali e non finali:
    - Un *PDA* potrebbe imporre l'accettazione solo alla fine di una sequenza *(finita)* di ϵ-mosse
    - Se così fosse, scambiare gli stati finali e iniziali non porterebbe a una complementazione corretta

3.9.2 Unione

La classe dei linguaggi riconosciuti dai *PDA* **non è chiusa** rispetto all'unione.

Per esempio, non esiste alcun *PDA* che riconosca $\{a^n b^n \mid n \geq 1\} \cup \left\{a^n b^{2n} \mid n \geq 1\right\}$. Tuttavia:

- $A = \{a^n b^n \mid n \geq 1\}$ è riconoscibile via *PDA*
- $B = \left\{a^n b^{2n} \mid n \geq 1\right\}$ è riconoscibile via *PDA*

Esempio intuitivo della dimostrazione:

- Se si procede riconoscendo una stringa di A ma trovando una stringa di B, pur sapendo che rimangono n caratteri b da leggere si ha perso l'informazione sul valore stesso di n
- Analogamente, riconoscendo una stringa di B ma trovando una stringa di A, pur sapendo che sono rimasti almeno n simboli nella pila non sarà possibile verificare se essi sono effettivamente n

3.9.3 Intersezione

La classe dei linguaggi riconosciuti da *PDA* **non è chiusa** rispetto all'intersezione.
Applicando la legge di *De Morgan* si verifica che:

$$\cup B = \overline{\left(\overline{A} \cap \overline{B}\right)} = \left(A^C \cap B^C\right)^C$$

Poiché i linguaggi dei *PDA* sono chiusi rispetto al complemento, se fossero chiusi rispetto all'intersezione dovrebbero essere chiusi anche rispetto all'unione, in contrario a quanto affermato nella Sezione 3.9.4

3.9.4 Differenza

La classe dei linguaggi riconosciuti da *PDA* **non è chiusa** rispetto alla differenza.
Applicando le leggi insiemistiche, si verifica che:

$$A \cap B = A - \overline{B} = A - B^C$$

Poiché i linguaggi dei *PDA* sono chiusi rispetto al complemento, se fossero chiusi rispetto alla differenza dovrebbero essere chiusi anche rispetto all'intersezione, in contrario a quanto affermato nella Sezione 3.9.3

3.10 Trasduttori a pila - *PDT*

3.10.1 Definizione formale *PDT*

Un **trasduttore a pila** (*pushdown transducer* o *PDT*) è una tupla $\langle Q, I, \Gamma.\delta, q_0, Z_0, F, O, \eta \rangle$, dove:

- Q è un **insieme di stati**, finito
- I è l'**alfabeto di ingresso**
- Γ è l'**alfabeto di pila**
  - $Z_0 \in \Gamma$ è il simbolo iniziale (il primo in *basso*) di pila
- δ è la **funzione di transizione**
- $q_0 \in Q$ è lo **stato iniziale**
- $F \subseteq Q$ è l'**insieme di stati finali**
- O è l'**alfabeto di uscita**
- η è la **funzione di uscita**:
  - $\eta : Q \times (I \cup \{\epsilon\}) \times \Gamma \to O^*$
  - **dominio**: prodotto cartesiano dell'insieme di stati, alfabeto di ingresso (compreso ϵ) e alfabeto di pila
  - **codominio**: stella di Kleene dell'alfabeto di uscita

Si noti che:

- $Q, I, \Gamma, \delta, q_0, Z_0, F$ sono definiti in modo analogo ai *PDA* nella sezione 3.1.

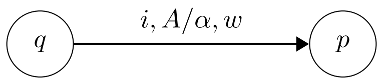


Figura 14: Transizione tra due stati in un *PDT*

- η è definita solo dove è definita δ
- La pila può essere necessaria perché richiesta dal linguaggio da riconoscere o perché richiesta dalla traduzione

L'illustrazione della transizione tra due stati con $\delta(q, I, A) = \langle p, \alpha \rangle$ e $\eta(q, I, A) = w$ è mostrata in Figura 14.
I *PDT* sono molto utilizzati per sopperire a due grosse lacune dei *FST*: *contare* i caratteri e *invertire* le stringhe.

3.10.2 Configurazione di un *PDT*

Una configurazione c di un *PDT* è una tupla $\langle q, x, \gamma, z \rangle$ in cui:

- $q \in Q$ è lo **stato corrente** del dispositivo di controllo
- $x \in I^*$ è la **porzione non letta** della stringa d'ingresso
- $\gamma \in \Gamma$ è la stringa di **simboli nella pila**
- z è la stringa già scritta sul **nastro di uscita**

3.10.3 Transizioni di un *PDT*

Per un dato *PDT* T, la **relazione binaria di transizione** $\vdash_T$ nello spazio di tutte le possibili configurazioni di A è definita da:

$$\langle q, x, \gamma, z \rangle \vdash_a c' = \langle q', x', \gamma', z' \rangle, \quad z' = z\overline{z}$$

se e solo se vale una delle due condizioni:

$$x = ay,\ x' = y,\ \gamma' = A\beta,\ \gamma' = \alpha\beta,\ \delta(q, a, A) = \langle q', \alpha \rangle,\ \overline{z} = \eta(q, a, A) \tag{3.1}$$

$$x = x',\ \gamma = A\beta,\ \gamma' = \alpha\beta,\ \delta(q, \epsilon, A) = \langle q', \alpha \rangle,\ \overline{z} = \eta(q, \epsilon, A) \tag{3.2}$$

Ovverosia:

- La condizione 3.1 descrive il passaggio da una configurazione all'altra quando viene letto un simbolo in ingresso
- La condizione 3.2 prende in considerazione il caso di ϵ-mosse *(quando la testina in ingresso rimane ferma)*

3.10.4 Condizione di accettazione di un *PDT*

La condizione di accettazione di un *PDT* può essere descritta come:

$$\forall\, x \in I^*,\ x \in L \Leftrightarrow c_0 = \langle q_0, x, Z_0, \epsilon \rangle \vdash^* c_F = \langle q, \epsilon, \gamma, z \rangle,\ q \in F$$

La traduzione di x è definita se e solo se la stringa x è accettata

4 Macchine di Turing - *TM*

Come visto in precedenza (Sezione 3.9.2), l'unione di alcuni linguaggi riconosciuti dai *PDA* non può essere riconosciuta dai *PDA* perché la classe di questi ultimi non è chiusa rispetto all'unione.

Si consideri il linguaggio $L = \{a^n b^n c^n \,|\, n > 0\}$:

- La pila può essere usata per contare le a
- I simboli sulla pila possono essere usati per controllare che il numero di b sia uguale al numero di a
  - ✖ Il numero di c, di conseguenza, **non può essere contato**

Questa limitazione è dovuta alla **pila**. Infatti essa è una memoria **distruttiva**, perché una volta che il simbolo viene letto, esso è distrutto. Questa proprietà può essere dimostrata formalmente attraverso una generalizzazione del *pumping lemma* (visto in Sezione 2.3.3).
Per risolvere questo problema, sono state introdotte le **Macchine di Turing** (in Inglese *Turing Machine* o *TM*), che fanno uso di **nastri di memoria**. Essi, infatti, non sono di natura distruttiva e possono venire letti più volte.

Un diagramma semplificato di una *TM* è mostrato in Figura 15.

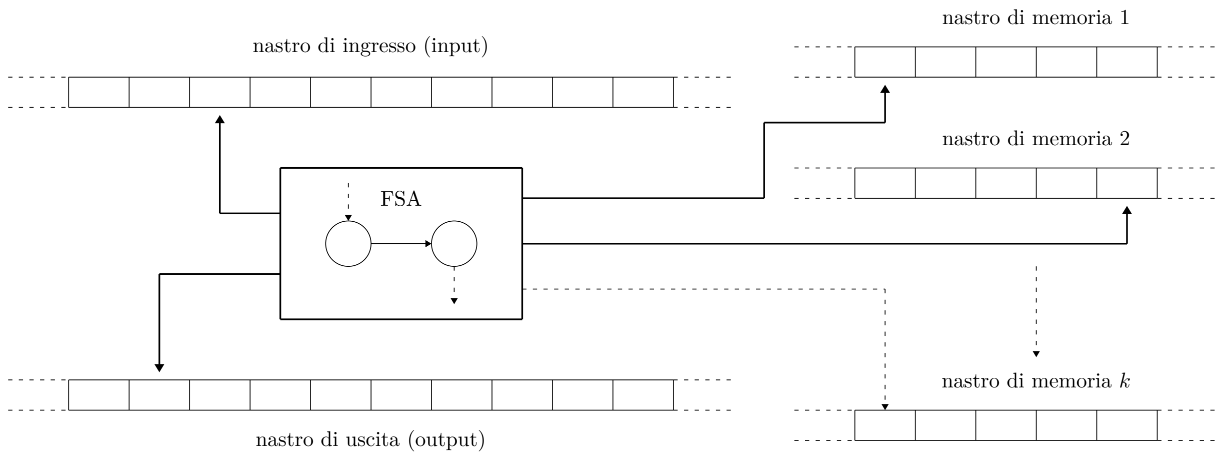


Figura 15: Diagramma semplificato di una TM

Informalmente, una *TM* è costituita da:

- **Stati** e **alfabeto** definiti come negli *FSA* (Sezione 2.2)
- Un nastro di **ingresso**, con la testina T_I
- Un nastro di **uscita**, con la testina T_F
- Un **dispositivo di controllo**
- Un **alfabeto di memoria**
- k nastri di **memoria**:
  - ogni nastro presenta una **testina di lettura**, $T_0, \dots, T_k$
  - rappresentati come **sequenze infinite**
  - gli spazi **vuoti** sono occupati da simboli speciali detti `blank` (anche `b`, $-$)
  - inizialmente, ogni nastro di memoria è **vuoto**
  - in ogni momento, i nastri contengono **un numero finito** di simboli non-`blank`

Un passo di **computazione** di una *TM* è basato su:

- Un **simbolo letto** dal nastro di ingresso
- K simboli, uno per ogni nastro di **memoria**
- **Stato** del dispositivo di controllo

e può eseguire le seguenti operazioni **(mosse)**:

1. **Cambio di stato** del dispositivo di controllo
2. **Sovrascrittura** di un simbolo in nelle celle sottostanti alle testine dei nastri di memoria
3. **Spostamento** delle $k+1$ testine:

   $\rightarrow$ i k nastri di memoria possono spostarsi a *sinistra* o *destra*
   $\rightarrow$ il nastro di uscita può spostarsi solo a *destra*

4. Nessuna operazione, fermandosi **definitivamente**

La direzione di spostamento di ogni testina deve essere specificata esplicitamente, indicando:

$\rightarrow$ R per *destra di una posizione*
$\leftarrow$ L per *sinistra di una posizione*
$\downarrow$ S per *nessuno spostamento*

4.1 Definizione formale *TM*

Formalmente, una *TM* con k nastri è una tupla di 9 elementi $\langle Q, I, \Gamma, O, \delta, \eta, q_0, Z_0, F \rangle$ dove:

- Q è un insieme di **stati**, finito
- I è l'**alfabeto** di **ingresso**
- O è l'**alfabeto** di **uscita**
- Γ è l'**alfabeto** di **memoria**
- δ è la **funzione** di **transizione**:
  - $\delta : (Q - F) \times I \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{R, L, S\}^{k+1}$
  - **dominio**: prodotto cartesiano di tutti gli stati meno quelli finali, alfabeto di ingresso e alfabeto di memoria *elevato al numero di nastri*
  - **codominio**: prodotto cartesiano di tutti gli stati, dell'alfabeto di memoria elevato al numero di nastri e dei possibili movimenti dei nastri *elevati a* $k+1$

    $\rightarrow$ $k+1$ perché si riferisce ai k nastri di memoria e il nastro di uscita
  - può essere **parziale**, come avviene nei *FSA* (Sezione 2.2)
  - è definita in modo tale che non ci siano transizioni uscenti da uno stato finale
- η è la **funzione** di **uscita**:
  - $\eta : (Q - F) \times I \times \Gamma^k \rightarrow O \times \{R, S\}$
  - **dominio**: coincide con quello della funzione δ
  - **codominio**: prodotto cartesiano dell'alfabeto di uscita con i possibili movimenti del nastro di uscita
  - può essere **parziale**
- $q_0 \in Q$ è lo stato **iniziale**
- $Z_0 \in \Gamma$ è il **simbolo iniziale** di **memoria**
- $F \subseteq Q$ è l'**insieme di stati finali**

Se il valore di k non è indicato, la *TM* viene semplicemente chiamata *TM multinastro (con un numero arbitrario di nastri)*.
La presenza di O e η non è obbligatoria. Essi infatti sono definiti se è previsto un *nastro di uscita*, come per esempio nelle *TM traduttrici* (Sezione 4.7).

Si consideri una transizione in cui:

- q_0, q_1 sono due **stati**
- i è il **simbolo** di **ingresso**
- A_j è il **simbolo** letto dal j-esimo nastro di memoria
- A'_j è il **simbolo** che rimpiazza A_j
- M_0 è la **direzione** della testina del nastro di ingresso T_i
- $M_j,\ \forall 1 \leq j \leq k$ è la **direzione** della testina del j-esimo nastro di memoria

La notazione grafica di questa situazione è mostrata in Figura 16.

$$q_0 \xrightarrow{i, \langle A_1, A_2, \ldots, A_k \rangle \mid \langle A'_1, A'_2, \ldots, A'_k \rangle, \langle M_0 M_1 \ldots, M_k \rangle} q_1$$

Figura 16: Transizione di una TM

4.2 Configurazione di una *TM*

Analogamente a quanto visto *PDA* (Sezione 3.4), anche le *TM* ammettono la definizione di **configurazione**. Essa dovrà includere:

- Lo **stato** del dispositivo di controllo
- La **stringa** sul nastro di ingresso e la **posizione della testina**
- la **stringa** e la **posizione della testina** per ogni nastro di memoria

Formalmente: una configurazione c di una *TM* con k nastri di memoria è la tupla di $k+3$ elementi:

$$c = \langle x \uparrow iy, \alpha_1 \uparrow A_1\beta_1, \ldots, \alpha_k \uparrow A_k\beta_k, u \uparrow o \rangle$$

- $q \in Q$ è lo **stato corrente** del dispositivo di controllo
- $x, y \in I^*, i \in I$ rappresentano il **contenuto** del nastro di ingresso
- $\alpha_r, \beta_r \in \Gamma^*,\ A_r \in \Gamma\ \forall\, r\ 1 \leq r \leq k$ rappresentano il contenuto dei **nastri di memoria**
  - le testine di ogni nastro sono posizionate sulla cella che memorizza il primo simbolo della stringa che segue il simbolo $\uparrow$
  - $\uparrow \notin \{I \cup \Gamma \cup O\}$
- uo è il contenuto del **nastro di uscita** (se definito, come nelle *TM* traduttrici viste in Sezione 4.7)
  - $u \in O^*$ è la stringa già scritta sul nastro di uscita
  - $o \in O$ è l'ultimo carattere scritto

4.2.1 Configurazione iniziale

La **configurazione iniziale** c_0 di M è del tipo: $c_0 = \langle q_0, \uparrow iy, \uparrow Z_0, \ldots, \uparrow Z_0, \uparrow Z_0, \uparrow \mathtt{b} \rangle$ dove:

- $q = q_0$, lo stato attuale è quello iniziale
- $x = \epsilon$, nessun carattere della stringa è ancora stata letta
- $A_r = Z_0, \alpha_r = \beta_r = \epsilon\ \forall r$, tutti i nastri di memoria sono vuoti
- $u = \epsilon, o = \mathtt{b}$ il nastro di uscita è vuoto *(se definito)*

Inoltre, tutte le testine sono all'inizio del corrispondente nastro.

4.3 Transizioni di una *TM*

La relazione di **transizione** $\vdash_M$ (detta anche *mossa* o *passo computazionale*) fra due configurazioni c e c' di una *TM* multinastro M è definita nel modo seguente:
Siano c, c' le due configurazioni tra le quali si esegue la transizione:

$$c = \langle q, x \uparrow iy, \alpha_1 \uparrow A\beta_1, \ldots, \alpha_k \uparrow A\beta_k, u \uparrow o \rangle$$
$$c = \langle q', x' \uparrow i'y', \alpha'_1 \uparrow A'\beta'_1, \ldots, \alpha'_k \uparrow A'\beta'_k, u' \uparrow o' \rangle$$

Sia δ la funzione di transizione, definita come:

$$\delta(q, i, A_1, \ldots, A_k) = \langle p, C_1, \ldots, C_k, N, N_1, \ldots N_k \rangle$$

con:

$$p \in Q,\ N \in \{R, L, S\},\ C_r \in \Gamma,\ N_r \in \{R, L, S\}\ \forall 1 \leq r \leq k$$
$$x = \overline{xi},\ y = \overline{jy},\ \alpha_r = \overline{\alpha_r A_r},\ \beta_r = \overline{B_r\beta_r}$$

Sia η la funzione di uscita, definita come:

$$\eta(q, i, A_1, \ldots, A_k) = \langle v, M \rangle$$

con:

$$v \in O \quad M \in \{R, S\}$$

Allora la transizione $c \vdash_M c'$ (da c a c') se e solo se:

$$p = q' \tag{4.1}$$

Se vale **una delle condizioni mutuamente esclusive**:

$$x = x',\ i = i',\ y = y',\ N = S \tag{4.2}$$
$$x' = xi,\ i' = \overline{j},\ y' = \overline{y},\ N = R \tag{4.3}$$
$$x' = \overline{x},\ i' = i,\ y' = iy,\ N = L \tag{4.4}$$

Inoltre, per $1 \leq r \leq k$ deve valere anche **uno dei casi mutuamente esclusivi**:

$$\alpha'_r = \alpha_r,\ A'_r = C_r,\ \beta'_r = \beta_r,\ N_r = S \tag{4.5}$$
$$\alpha'_r = \alpha_r C_r,\ A'_r = B_r,\ \beta'_r = \overline{\beta_r},\ N_r = R \tag{4.6}$$
$$\alpha'_r = \overline{\alpha_r},\ A'_r = \overline{A_r},\ \beta'_r = C_r \beta_r,\ N_r = L \tag{4.7}$$

Infine anche **una delle due condizioni mutualmente esclusive**:

$$u' = u,\ o' = v,\ M = S \tag{4.8}$$
$$u' = uv,\ o' = \mathtt{b},\ M = R \tag{4.9}$$

4.3.1 Singificato delle condizioni di transizione della *TM*

- La condizione 4.1 vincola lo stato di c' a essere quello di arrivo della transizione
- Le condizioni dal 4.2 al 4.4 definiscono l'evoluzione del nastro di ingresso nel passaggio da c a c':
  - Se la testina rimane ferma (condizione 4.2, in cui $N = S$), le tre parti in cui essa divide il nastro *(parte sinistra, destra e simbolo corrente)* rimangono invariate tra le due configurazioni
  - Se la testina si muove a destra (condizione 4.3, in cui $N = R$), la parte a sinistra della testina in c' conterrà anche il simbolo corrente di c, il simbolo corrente di c' sara il primo simbolo della parte destra in c e la rimanente parte destra di c sarà la parte destra di c'
  - Se la testina si muove a sinistra (condizione 4.4, in cui $N = L$), laa parte a sinistra della testina in c' conterrà tutti i simboli della parte sinistra in c, tranne l'ultimo che diverrà il simbolo corrente di c' e la parte destra di c' sarà composta dal simbolo corrente in c seguito dalla sua parte destra
- Le condizioni dal 4.5 alla 4.7 specificano l'evoluzione dei nastri di memoria in analogia con i precedenti casi. In particolare:
  - La testina rimane ferma, condizione 4.5
  - La testina si muove a destra, condizione 4.6
  - La testina si muove a sinistra, condizione 4.7
- Le condizioni 4.8 e 4.9 specificano l'evoluzione del nastro di uscita
  - Non è specificato il comportamento per $N = L$ perché la testina del nastro di uscita si muove solo a destra
  - Se la *TM* non ha un nastro di uscita, queste condizioni non vanno considerate

4.4 Condizioni di accettazione di una *TM*

Sia M una TM multinastro Una stringa $x \in I^*$ è **accettata** da M se e solo se:

$$c_0 = \langle q_0, \uparrow x, \uparrow Z_0, \dots, \uparrow Z_0 \rangle \vdash_M^* \langle q, x' \uparrow iy, \alpha_1 \uparrow A_1 \beta_1, \dots, \alpha_k \uparrow A_k \beta_k \rangle$$

dove:

- c_F è detta **configurazione finale**
- $\vdash_M^*$ è la chiusura riflessiva e transitiva della relazione $\vdash_M$
- $q \in F$, quindi lo stato q fa parte dell'insieme di stati finali
- Le testine possono trovarsi in un qualsiasi punto dei rispettivi nastri di memoria

Il linguaggio *accettato* da M è definito come:

$$L(M) = \{x \mid x \in I^* \text{ e } x \text{ è accettato da } M\}$$

Intuitivamente, il linguaggio riconosciuto da una TM M è composto da tutte e sole le stringhe che permettono di andare dallo stato iniziale a uno degli stati finali. Poiché il nastro di ingresso è in grado di muoversi in entrambe le direzioni, non è richiesto che al termine dell'esecuzione la testina si trovi al termine della stringa di ingresso.
Una volta che M raggiunge uno stato finale, per definizione della funzione di transizione δ, non potrà più lasciarla e la computazione termina.

4.5 Operazioni sulle *TM*

È possibile dimostrare che le TM **sono chiuse** rispetto a:

- **Intersezione**
- **Unione**
- **Concatenazione**
- **Stella di Kleene**

Perché una TM può facilmente simulare due altre TM (siano esse in *serie* o *parallelo*), si dimostra la chiusura rispetto alle prime due operazioni. Di conseguenza, si dimostra la chiusura rispetto alle altre.

Analogamente, si dimostra che **non sono chiuse** rispetto a:

- **Complemento**
- **Differenza**, conseguenza della non chiusura rispetto al complemento

La (non) chiusura rispetto a queste due operazioni è dovuta alla presenza di *cicli* all'interno delle TM.
Infatti, se essi non fossero presenti in una TM sarebbe sufficiente definire l'insieme di stati di arresto e partizionarli in stati di di *accettazione* e *non accettazione*. I problemi sorgono qualora una computazione non dovesse terminare (come visto in Sezione 5.3).

4.6 Proprietà delle *TM*

Di seguito sono enunciate alcune proprietà delle TM:

1. Ogni TM è equivalente ad un'opportuna TM dotata solo di **due stati non finali** e di **uno finale**
   - può essere necessario accrescere l'alfabeto
   - di conseguenza, sono sufficienti 3 stati per implementare qualsiasi algoritmo
2. Ogni TM è equivalente ad un'opportuna TM avente un alfabeto formato solo da **due simboli distinti**
   - può essere necessario accrescere il numero di stati
   - di conseguenza, sono sufficienti 2 simboli *(più il simbolo `blank`)* per implementare qualsiasi algoritmo

Le due proprietà impongono una scelta: è possibile ridurre gli stati di una TM agendo sull'alfabeto (1) o viceversa ridurre la dimensione dell'alfabeto agendo sul numero di stati (2). La scelta si riduce quindi ad un problema progettuale.

4.7 *TM* traduttrice

Quando si definiscono le *TM* con il nastro di uscita, esse diventano dei *trasduttori*, ossia possono essere usate per tradurre stringhe di I^* in stringhe di O^*. In questo caso, le *TM* vengono viste come le tuple di 9 elementi viste in Sezione 4.1.

4.7.1 Traduzione tramite *TM*

Una *TM* multinastro M definisce una **traduzione** $\tau_M : I^* \to O^*$ secondo la regola seguente:

$$\tau_M(x) = y \textit{ se e solo se } \langle q_0, \uparrow x, \uparrow Z_0, \ldots, \uparrow Z_0, \uparrow \mathtt{b}\rangle \vdash_M^* \langle q, x' \uparrow iy, \alpha_1 \uparrow A_1\beta_1, \ldots, \alpha_k \uparrow A_k\beta_k, y \uparrow o\rangle$$

con $q \in F$.
In generale, una *TM* M definisce una **traduzione parziale** $\tau_M : I^* \to O^*$. Infatti, τ_M è indefinita se:

1. M raggiunge una configurazione di arresto il cui stato non appartiene a F
2. M non si ferma mai quando opera su x

Intuitivamente, una stringa x viene tradotta in una stringa y da una *TM* M se esiste un cammino che parte da una configurazione **iniziale** con x sul nastro di ingresso e termina in una configurazione **finale** con y sul nastro di uscita.

4.8 Confronto di *TM* con altre macchine

Si vuole ora confrontare la classe di macchine computazionali appartenente alle *TM* con altri tipi di macchine.

4.8.1 *TM* vs *PDA*

Come già visto, i linguaggi $a^nb^nc^n$ e $a^nb^n \cup a^nb^{2n}$ non possono essere riconosciuti da un *PDA* (Sezione 2.3.3.1) mentre tramite *TM* il riconoscimento funziona. Ogni linguaggio riconoscibile da un *PDA* può essere riconosciuto da una *TM*: si può sempre costruire una *TM* che usa un nastro di memoria come se fosse una pila.
I linguaggi accettati dalle *TM* sono detti **ricorsivamente enumerabili**.

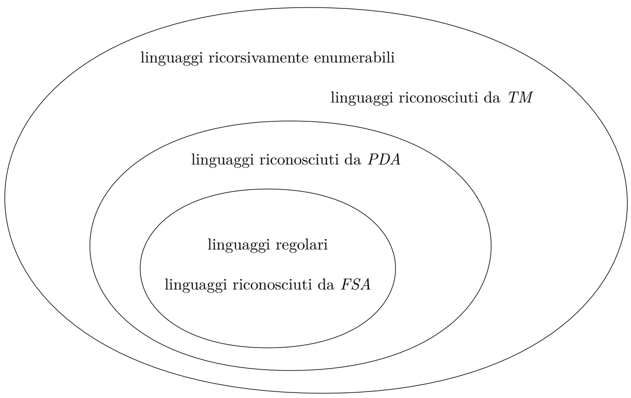


Figura 17: Relazione tra linguaggi

4.8.2 *TM* vs macchine di *Von Neumann*

Le *TM* possono simulare una macchina di *Vonn Neumann* (in Inglese *VNM*), un modello astratto di computer. La principale differenza avviene nell'accesso alla memoria: mentre nelle *TM* è *sequenziale*, nelle *VNM* è *diretto*. Tuttavia, il metodo di accesso alla memoria non influenza il potere espressivo di una macchina: non cambia la classe di problemi risolvibili con essa, ma può cambiarne la complessità. Infatti si tramite *TM* si possono anche calcolare funzioni ed eseguire algoritmi *sebbene implementare questo tipo di operazioni possa essere estremamente complicato.*

In conclusione, la *TM* è un modello più astratto di computer con accesso **sequenziale** al suo spazio di memoria.

4.9 Memoria delle *TM*

Esistono *TM* con diversi modelli di memoria:

- A nastro **singolo** (Figura 18):
  - Normalmente è **illimitato** in entrambe le direzioni
  - Serve da contemporaneamente da **ingresso, uscita e memoria**
  - È il modello più simile alla macchina originalmente ideata da *Alan Turing*
- A nastro **bidimensionale** (Figura 19):
  - È presente **una testina per ogni dimensione**
  - Può essere generalizzato a un numero *arbitrario* di dimensioni

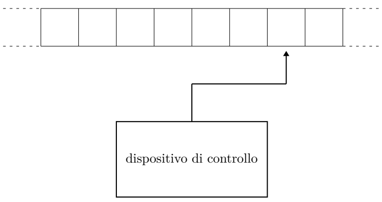


Figura 18: *TM* a nastro singolo

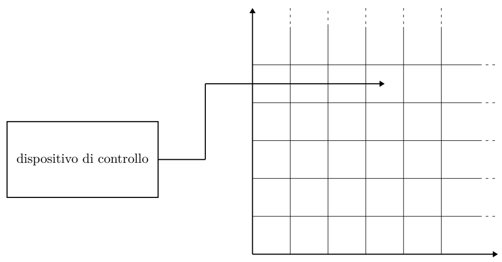


Figura 19: *TM* bidimensionale

Entrambi i modelli di *TM* sono **equivalenti**, poiché riconoscono la stessa classe di linguaggi. A tal proposito, si consulti la Figura 20, che mostra la configurazione della *TM* a nastro singolo M':

- $c(T_x)'$ rappresenta il contenuto non vuoto del nastro T_x di M alla sinistra della testina di T_x
- $c(T_x)''$ rappresenta il contenuto dello stesso nastro alla destra della testina, compreso il carattere sotto di essa
- $*$ e \$ sono simboli che non appartengono a $I \cup \Gamma \cup O$ e vengono usati per marcare i limiti fra il contenuto dei diversi nastri e posizioni della testina (rispettivamente)
- $\overline{q}$ è un'opportuna codifica dello stato M

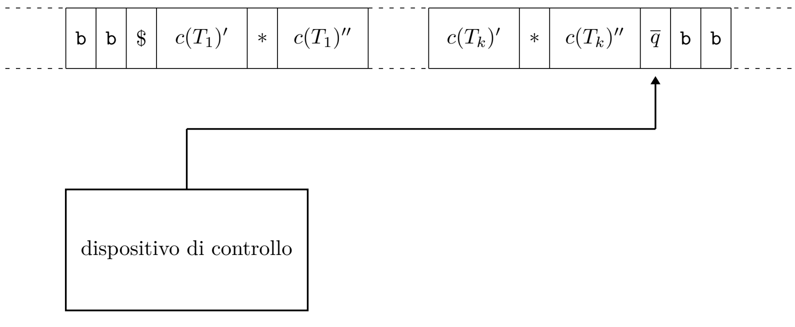


Figura 20: Equivalenza tra *TM* multinastro e *TM* a nastro singolo

5 Modelli operazionali non deterministici

Tutti i modelli considerati fino ad ora (*FSA*, *PDA*, *TM*, ...) sono **deterministici**: una volta fissato uno *stato iniziale* e un *ingresso*, la loro evoluzione è *univocamente determinata*. In certe situazioni, però, il modello che si desidera modellare non può essere descritto in modo così deterministico, in quanto l'osservatore non possiede informazioni sufficientemente accurate da consentirgli di prevederne l'esatta evoluzione per ogni configurazione. Per esempio, si consideri la serie di istruzioni:

```
if x >= y then:
  max := x

if y >= x then:
  max := y
```

Essa definisce il massimo tra due numeri `a` e `b`. Tale notazione rappresenta una specifica non deterministica: non viene illustrato il comportamento per `a == b` e l'applicazione di entrambe le regole risulta in un output valido.
Normalmente, nei linguaggi di programmazione, si usa una precedenza di tipo lessicale: viene applicata la prima conzidione scelta.
Un'esempio analogo può essere fatto sfruttando gli algoritmi di *ricerca binaria*. Essi infatti compiono una *simulazione* di algoritmi non deterministici:

```
while true:
  l'elemento ricercato è nella radice?
    termina

  cerca nel sotto albero sinistro
  cerca nel sotto albero destro
```

Siccome non è possibile determinare a priori quale dei due cammini *(sinistro o destro)* sia migliore, la scelta della priorità nei cammini è spesso **arbitraria**. Alternativamente, non viene fatta una scelta del cammino da intraprendere ma entrambi venengono esplorati in **parallelo**.
In entrambi i casi, grazie al non determinismo, è possibile implementare l'algoritmo senza la necessità di usare *backtracking*, che sarebbe invece obbligatorio nel caso si cercasse di usare un sistema deterministico.

Il **non determinismo** (*ND* per brevità) è quindi un modello di computazione. Viene spesso sfruttato nei linguaggi di programmazione per consentire la computazione parallela.
Il *ND* viene applicato a vari modelli computazionali, inclusi tutti quelli visti fin'ora.

5.1 *FSA* non deterministici

Un *FSA* non deterministico (*NFA*) è una tupla di 5 elementi $\langle Q, I, \delta, q_0, F\rangle$:

- Q, I, q_0, F sono definiti come in un *FSA* (Sezione 2.2)
- δ è la **funzione di transizione**:
  - $\delta : Q \times I \to \wp(Q)$
    $\to$ $\wp(Q)$ rappresenta l'insieme delle parti di Q
    $\to$ gli elementi di $\wp(Q)$ sono insiemi di stati
  - δ^* è definito induttivamente a partire da δ:
    1. $\delta^*(q, \epsilon) = \{q\}$
    2. $\delta^*(q, y \cdot i) = \bigcup\limits_{q' \in \delta^*(q,y)} \delta(q', i)$
       $\to$ i è l'ultimo carattere della stringa di ingresso

Nel caso di *FSA* accettori con *ND*, si dice che $x \in I^*$ è **accettata** da un *NFA* $\langle Q, I, \delta, q_0, F\rangle$ se e solo se $\delta^*(q_0, x) \cap F \neq \emptyset$.

Informalmente: tra le varie possibili esecuzioni (a parità di ingresso) dell'*NFA*, è sufficiente che una di esse vada a buon fine *(la stella di Kleene delle transizioni δ^* ha almeno uno stato che fa parte dell'insieme di stati finali)*. Questa definizione prende il nome di **non determinismo esistenziale**, che si oppone al non determinsimo **universale**: $\delta^*(q_0, x) \subseteq F$.

Gli *NFA* non sono più potenti degli *FSA*, ma presentano due grossi vantaggi:

- Può essere più semplice progettare un *NFA*
- Il numero di stati di un *NFA* può essere esponenzialmente più minore dell'analogo *FSA*

Per maggiori dettagli, si consulti la Sezione 5.1.2

5.1.1 Esempio di un *NFA*

In Figura 21 è mostrato un esempio di *NFA*. In esso, infatti, $\delta(q_0, a) = \{q_1, q_2\}$ e quindi la funzione di transizione **non è univocamente definita**.

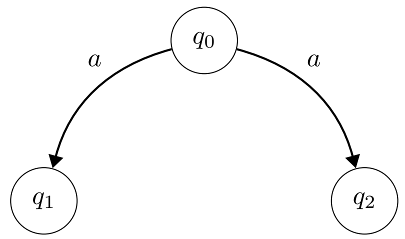


Figura 21: Esempio di un *NFA*

5.1.2 *DFA* vs *NFA*

Si osservi il *NFA* in Figura 22. Partendo da q_0 e leggendo `ab`, l'automa raggiunge uno stato che appartiene all'insieme $\{q_3, q_4, q_5\}$. Negli *NFA* viene ancora chiamato **stato** l'insieme dei possibile stati in cui esso può trovarsi durante l'esecuzione.

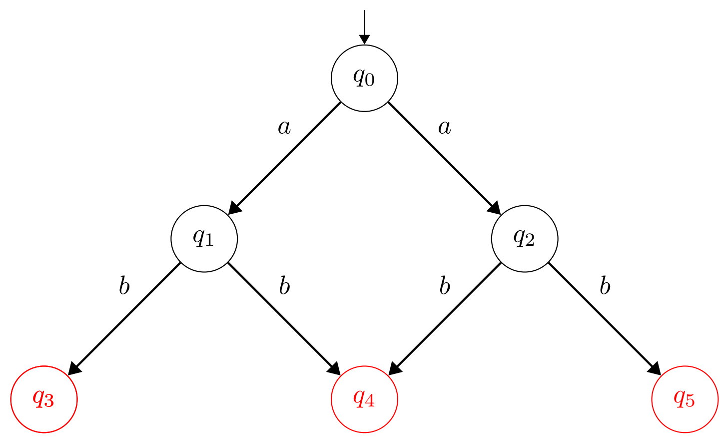


Figura 22: *DFA* e *NFA*

Formalmente, *NFA* e *DFA* **hanno lo stesso potere espressivo.** Dato un *NFA*, A, è possibile costruire un *DFA* A_D che accetti il medesimo linguaggio.

Sia $A = \langle Q, I, \delta, q_0, F \rangle$, $\delta : Q \times I \to \wp(Q)$. Si definisca $A_D = \langle Q_D, I, \delta_D, q_{0D}, F_D \rangle$ con:

- $Q_D = \wp(Q)$

  → l'insieme dei suoi stati è uguale all'insieme delle parti degli stati di A

- $\delta_D(q_D, i) = \bigcup_{q \in q_D} \delta(q, i)$

  → se un insieme di stati è raggiungibile a partire da uno stato del *NFA*, allora tale relazione viene preservata nel *DFA* sfruttando la costruzione degli stati come insiemi di stati del *NFA*

- $F_D = \{q_D \mid q_D \in Q_D \land F \cap q_D \neq \emptyset\}$

  → l'insieme dei suoi stati finali è dato dall'insieme di stati finali di A raggiungibili senza *ND*

- $q_{0d} = \{q_0\}$

  → il suo stato iniziale è uguale allo stato iniziale di A

Gli *NFA* non sono quindi più potenti dei corrispondenti *DFA* ma sono di dimensione ridotta. Il precedente teorema, infatti, produce un insieme di stati di cardinalità 2^n partendo da n stati di partenza del *NFA*. Inoltre, la formalizzazione più naturale di un problema è quella descritta mediante un *NFA*.

5.1.3 Conversione da *NFA* a *DFA*

Operativamente, per convertire un *FSA* in un *DFA* bisogna:

1. Creare l'insieme vuoto di stati del *DFA* Q'
2. Creare la tabella di transizione del *DFA* T'
3. Aggiungere lo stato iniziale del del *NFA* a Q'
4. Aggiungere le transizioni dello stato iniziale alla tabella di transizione T'

   → se lo stato di partenza porta a più stati per un certo input, allora essi vanno trattati come un singolo stato del *DFA*

5. Se un nuovo stato è presente in T':

   → Aggiungere il nuovo stato a Q'
   → Aggiungere le sue transizioni a T'

6. Ripetere il passo 5 finché non vengono più aggiunti nuovi stati a T'
7. T' è ora la tabella di transizione completa del *DFA* ricercato

5.2 *PDA* non deterministici

Un *PDA* non deterministico (*NPDA*) è una tupla di 7 elementi $\langle Q, I, \Gamma, \delta, q_0, Z_0, F\rangle$:

- $Q, I, \gamma, q_0, Z_0, F$ sono definiti come in un *PDA* (Sezione 3.1)
- δ è la **funzione di transizione**:
  - $\delta : Q \times (I \cup \{\epsilon\}) \times \Gamma \to \wp_F(Q \times \Gamma^*)$
  - $\wp_F$ indica l'insieme delle parti **finito** dell'insieme Q

    → Γ^* è un insieme **infinito**

Inoltre, la relazione $\vdash$ su $Q \times I^* \times \Gamma^*$ è definita da $\langle q, x, \gamma\rangle \vdash \langle q', x', \gamma'\rangle$ se e sono se è valida una delle due condizioni mutualmente esclusive:

$$x = ay, \quad x' = y, \quad \gamma = a\beta, \quad \gamma' = \alpha\beta, \quad \langle q', \alpha\rangle \in \delta(q, a, A) \tag{5.1}$$
$$x = x', \quad \gamma = a\beta, \quad \gamma' = \alpha\beta, \quad \langle q', \alpha\rangle \in \delta(q, \epsilon, A) \tag{5.2}$$

La stringa $x \in I^*$ è accettata dall'automa se e solo se:

$$\langle q_0, x, Z_0\rangle \vdash^* \langle q, \epsilon, \gamma\rangle, \quad q \in F, \gamma \in \Gamma^*$$

Informalmente, una stringa è accettata da un *NPDA* se esiste un cammino coerente con x che va dallo stato iniziale a uno stato finale quando essa viene letta **interamente**.

I *PDA*, tuttavia, sono intrinsecamente non deterministici. Nella loro definizione era infatti stato aggiunto il vincolo per cui:

$$\delta(q, \epsilon, A) \neq \perp \Rightarrow \delta(q, I, A) = \perp, \ \forall i \in I$$

Informalmente, se una transizione da uno stato è una ϵ-mossa, allora la stessa transizione non può essere definita per nessun altro ingresso.
Rimuovere questo vincolo, infatti, priverebbe i *PDA* del loro non determinismo. Analogamente è possibile avere non determinismo cambiando la funzione di transizione di *PDA*, modificando al contempo la transizioni tra configurazioni e la condizione di accettazione.

5.2.1 Chiusura delle *NPDA*

Come per gli altri formalismi con *ND*, è possibile dimostrare che un *NPDA* può riconoscere ogni linguaggio riconoscibile tramite *DPDA*. Tuttavia, siccome gli *NPDA* sono più potenti dei *DPDA*, la classe di linguaggi riconosciuti dai primi è più ampia di quella riconosciuta dai secondi e di conseguenza non è scontato che valgano le stesse proprietà di chiusura.
Gli *NPDA*, infatti, sono chiusi rispetto all'unione. È possibile costruire un *NPDA* che è collegato con due ϵ-mosse agli stati iniziali di due *DPDA*, così mostrando una chiusura rispetto all'unione (come mostrato nella Figura 23). Tuttavia, non rimanendo chiusi rispetto all'intersezione, non possono *(e non sono)* essere chiusi rispetto al complemento.

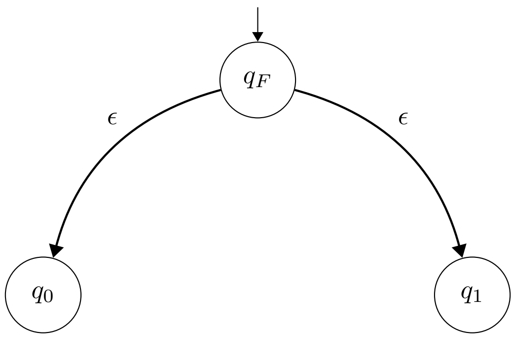


Figura 23: Intersezione di *PDA* tramite *NPDA*

5.2.2 Complemento e *ND*

Se una macchina è deterministica e la sua computazione termina, il **complemento** può essere ottenuto:

1. **Completando** la macchina
2. **Scambiando gli stati** di accettazione con quelli di non accettazione

Tuttavia, il non determinismo *(analogamente alla non terminazione)* rende questo approccio inapplicabile.
Come nei *DPDA*, le computazioni nei *NPDA* possono sempre essere fatte terminare.
Tuttavia, si possono avere due computazioni del tipo:

$$c_0 = \langle q_0, x, Z_0 \rangle \vdash^* c_1 = \langle q_1, \epsilon, \gamma \rangle$$
$$c_0 = \langle q_0, x, Z_0 \rangle \vdash^* c_2 = \langle q_2, \epsilon, \gamma \rangle$$

Con:

$$q_1 \in F \quad q_2 \notin F$$

Con questa configurazione, scambiando stati di accettazione e non, x verrebbe comunque accettato (e di conseguenza il complemento non sarebbe valido).

5.2.3 Conseguenze della chiusura delle *NPDA*

Grazie a questa proprietà degli *NPDA* è possibile riconoscere il linguaggio generato dall'unione di linguaggi $\{a^n b^n \mid n \geq 1\} \cup \left\{a^n b^{2n} \mid n \geq 1\right\}$.

5.2.4 Linguaggi riconosciuti dalle *NPDA*

I linguaggi riconosciuti da *NPDA* prendono il nome di **linguaggi non contestuali** (o **context-free**).

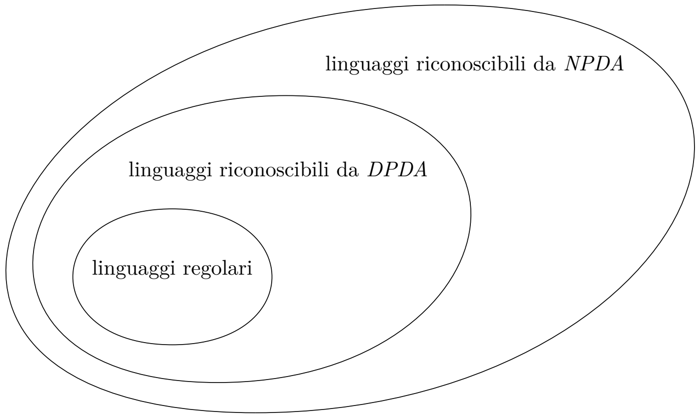


Figura 24: Linguaggi non contestuali

5.3 *TM* non deterministiche

Un *TM* deterministico (*NTM*) è un tupla di 9 elementi $\langle Q, I, \Gamma, O, \delta, \eta, q_0, Z_0, F\rangle$:

- $Q, I, \Gamma, O, q_0, Z_0, F$ sono definiti come in una *TM* (Sezione 4.1)
- δ è la **funzione di transizione**:
  - $\delta : (Q - F) \times I \times \Gamma^k \rightarrow \wp\left(Q \times \Gamma^k \times \{R, L, S\}^{k+1} \times \{R, S\}\right)$
- η è la **funzione di uscita**:
  - $\eta : (Q - F) \times I \times \Gamma^k \rightarrow \wp(O \times \{R, S\})$

Contrariamente a quanto avvenuto per la definizione delle *NPDA* (Sezione 5.2), non è necessario specificare che nella definizione δ l'insieme delle parti sia finito. Ciò è dovuto al fatto che l'insieme $Q \times \Gamma^k \times \{R, L, S\}^{k+1} \times \{R, S\}$ è finito, poiché costruito dal prodotto cartesiano di insiemi finiti.

5.3.1 Albero di computazione di una *NTM*

Per come è definita la funzione di transizione di M, se si considera una computazione di M su una stringa in ingresso, essa è ben descritta da un albero di configurazioni in cui è inserita ogni configurazione raggiungibile dalla configurazione iniziale. Una parola viene accettata se esiste almeno un cammino che termina in una configurazione finale. Un esempio di albero di configurazione per una *NTM* è mostrato in Figura 25, dove:

- I cerchi indicano configurazioni di accettazione
- I rettangoli indicano configurazioni di `halt` ma non di accettazione
- Le linee tratteggiate configurazione che non terminano

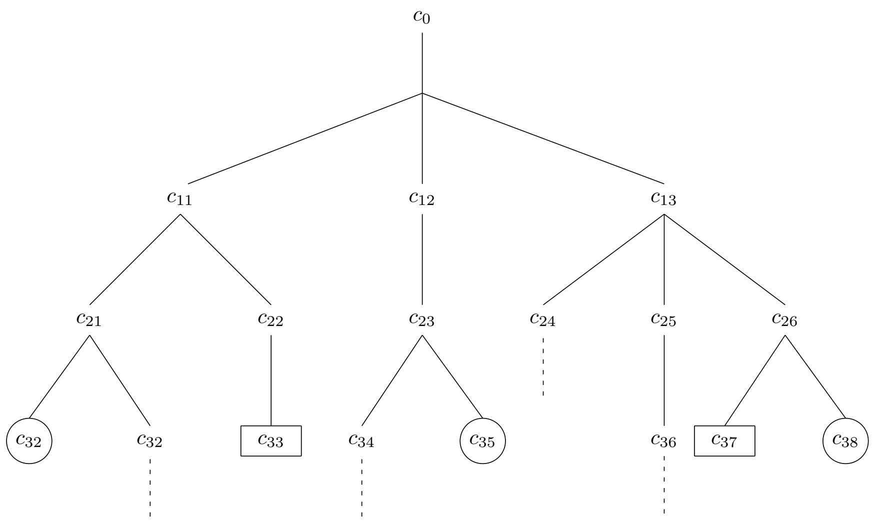


Figura 25: Albero delle configurazioni di una *NTM*

5.3.2 Accettazione di una stringa da un *NTM*

Una stringa $x \in I^*$ è accettata da una *NTM* se e solo se esiste una computazione che termina in uno stato di accettazione. Il problema dell'accettazione di una stringa si può ridurre quindi alla visita di un albero di computazione.
Per cercare la via verso uno stato di accettazione, si riconoscono due tipi di approccio:

- Visita in *"profondità"*, detta **depth first**
  - Un percorso viene seguito fino al suo termine
  - Quando viene raggiunta la fine, un'altro ramo viene selezionato ed esplorato tramite *backtracking*
- Visita in *"ampiezza"*, detta **breadth first**
  - Si crea una coda di nodi da visitare
  - Ogni volta che un nuovo nodo viene trovato, i suoi successori si aggiungono alla fine della coda

Si noti tuttavia che se parola non viene accettata, una *TM* può entrare in uno ciclo infinito e quindi non terminare mai la sua computazione. Questo comportamento può essere causato dalla presenza di rami di lunghezza infinita all'interno dell'albero delle configurazioni. Una ricerca di tipo **depth first** non può funzionare per questo tipo di problema, perché finirebbe con molta probabilità in un ciclo infinito.
Una macchina deterministica può simulare l'attraversamento di un albero tramite una ricerca **breadth first**.

5.3.3 *DTM* vs *NTM*

È possibile costruire una *DTM* che visita un'albero di computazione costruito da una *NTM* livello dopo livello, implementando una ricerca di tipo **breadth first**.
Quindi, data una *NTM*, è possibile costruire:

- Una *DTM* analoga che determina se la *NTM* riconosce una stringa x
- La sua equivalente *DTM*

Infine, può essere dimostrato che il *ND* **non aggiunge potere espressivo** alle *TM*.

5.3.4 *NPDA* vs *NTM*

Nella Figura 26 è possibile osservare varie possibilità di relazione confronto tra *NPDA* e *NTM*, specialmente nei casi:

a) $NTM \cup NPDA \neq \emptyset$, quindi *NTM* e *NPDA* possono riconoscere dei linguaggi in comune

b) $NPDA \subseteq NTM$, quindi le NTM rappresentano una sottocategoria delle $NPDA$
c) $NTM \subseteq NPDA$, quindi le $NPDA$ rappresentano una sottocategoria delle NTM
d) $NTM \equiv NPDA$, qundi le due categorie coincidono

Tuttavia, si dimostra che i casi:

- (a), (c) sono **falsi** perché una NTM può simulare un $NPDA$ usando il nastro come pila
- (d) è **falso** perché la pila é una memoria distruttiva, al contrario del nastro

Quindi rimane vero il caso (c) e le NTM hanno potere espressivo superiore.

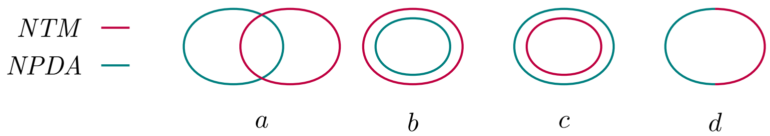


Figura 26: Confronto tra $NPDA$ e NTM

6 Grammatiche

Fino ad ora, gli automi sono stati usati come modelli astratti nei problemi di riconoscimento dei linguaggi. Dato un *riconoscitore* A e il suo linguaggio di ingresso I, il linguaggio da esso definito è:

$$L(A) = \{x \mid x \in I^*, \, x \text{ è accettata da } A\}$$

Tuttavia, esistono altri tipi di formalismi per descrivere un linguaggio: le **grammatiche formali**. Essi sono costituiti da un **insieme di regole** (sono quindi di tipo *generativo*) che costruiscono le frasi di un linguaggio. Una grammatica formale genera stringhe di un linguaggio attraverso un processo detto di *riscrittura.* Esso è costituito da un insieme di tecniche per sostituire sottotermini di una formula con altri termini. Non si applica solo ai linguaggi in senso naturale ma a un'ampia gamma di contesti.

Esempi di riscrittura:

- $A \wedge B$ viene riscritto come $\neg(\neg A \vee \neg B)$
- $\neg A \vee B$ viene riscritto come $A \rightarrow B$

In generale, un meccanismo di riscrittura è un insieme di *regole linguistiche* che descrivono l'*oggetto principale* (come la frase) come sequenza di *componenti.* Ogni componente può essere poi *"raffinato"* da oggetti più dettagliati e così via, fino a ottenere una sequenza di *oggetti elementari.*
Una grammatica è quindi composta da:

- Oggetto principale, detto **simbolo iniziale**
- Inseme di componenti da sostituire durante il processo di derivazione, detti **simboli non terminali**
- Insieme di elementi di base, detti **simboli terminali**
- Regole di sostituzione, dette **produzioni**

Noam Chomsky, uno degli studiosi più importanti delle grammatiche formali, all'interno del libro *"On Certain Formal Properties of Grammars, Information and Control"* afferma che:

> *"A grammar can be regarded as a device that enumerates the sentences of a language"*

> *"A grammar of L can be regarded as a function whose range is exactly L"*

6.1 Definizione formale di grammatica

Una **grammatica** (detta anche *grammatica non ristretta*) G è una tupla di 4 elementi $\langle V_T, V_N, P, S \rangle$:

- V_T è un insieme finito di *simboli terminali*, detto **alfabeto terminale**
  - $\rightarrow$ gli elementi di V_T sono normalmente scritti in **minuscolo**
- V_N è un insieme finito di *simboli non terminali*, detto **alfabeto non terminale**
  - $\rightarrow$ V_N è costruito in modo che $V_T \cap V_N = \emptyset$
  - $\rightarrow$ l'alfabeto V è quindi dato da $V_T \cup V_N = V_T \oplus V_N$
  - $\rightarrow$ gli elementi di V_N sono normalmente scritti in **maiuscolo**
- P è un insieme finito, detto **insieme delle produzioni** di G
  - $P \subseteq V^* \cdot V_N^+ \cdot V^* \times V^*$
  - un elemento $p = \langle \alpha, \beta \rangle$ di P verrà indicato con $\alpha \rightarrow \beta$
  - la stringa α è detta **parte sinistra** di p
  - la stringa β è detta **parte destra** di p
  - due regole $s \rightarrow d_1, s \rightarrow d_2$ con la stessa parte sinistra si possono essere accorpate come $s \rightarrow d_1 \mid d_2$
- S è un elemento *"particolare"* di V_N, detto **assioma** o **simbolo iniziale**
  - $\rightarrow$ S non può essere mai parte destra di una derivazione

6.1.1 Relazione di derivazione immediata

Data una grammatica G, si definisce su V^* la relazione binaria di **derivazione immediata**, indicata dalla notazione

$$\alpha \underset{G}{\Longrightarrow} \beta$$

è definita se e solo se

$$\alpha = \alpha_1 \gamma \alpha_2, \quad \beta = \alpha_1 \delta \alpha_2$$
$$\alpha_1, \alpha_2, \delta \in V^*, \quad \gamma \in V_N^+, \quad \gamma \to \delta \in P$$

Il simbolo G in $\underset{G}{\Longrightarrow}$ viene normalmente omesso qualora il contesto sia univocamente definito, indicando quindi le derivazioni come $\Rightarrow$.

Come già visto con le operazioni tra linguaggi, (Sezione 1.3.1), $\overset{*}{\underset{G}{\Longrightarrow}}, \overset{+}{\underset{G}{\Longrightarrow}}, \overset{n}{\underset{G}{\Longrightarrow}}$ indicano rispettivamente la chiusura riflessiva e transitiva, la chiusura transitiva e la potenza n di $\underset{G}{\Longrightarrow}$.

6.1.2 Linguaggio di accettazione di una grammatica

Una grammatica $G = \langle V_N, V_T, P, S \rangle$ genera un linguaggio $L(G)$ sull'alfabeto V_T. Esso è definito come:

$$L(G) = \left\{ x \mid S \overset{*}{\underset{G}{\Longrightarrow}} X, x \in V_T^* \right\}$$

Informalmente, il linguaggio generato da una grammatica è quindi costruito da tutte le *e sole* stringhe di **soli simboli terminali** che possono essere generate a partire dall'**assioma** S applicando un numero qualsiasi di sostituzioni (*passi*). Notare che gli elementi non terminali di G non fanno parte della stringa da essa generata. Le regole non sono univoche: più sostituzioni diverse possono essere applicate alla stessa stringa di partenza. Il processo di derivazione è quindi intrinsecamente non deterministico. Inoltre, alcuni cammini di derivazione possono portare a sequenze di terminali e non terminali, generando quindi stringhe non valide.

6.2 Gerarchia di Chomsky

La definizione delle grammatiche (come vista in questi appunti) è opera principalmente di *Noam Chomsky*, linguista e filosofo statunitense. Egli ha infatti introdotto una loro classificazione, nota come **gerarchia di Chomsky** (il cui diagramma è osservabile in Figura 27). In Tabella 1 sono mostrate alcune caratteristiche di ciascuna grammatica.

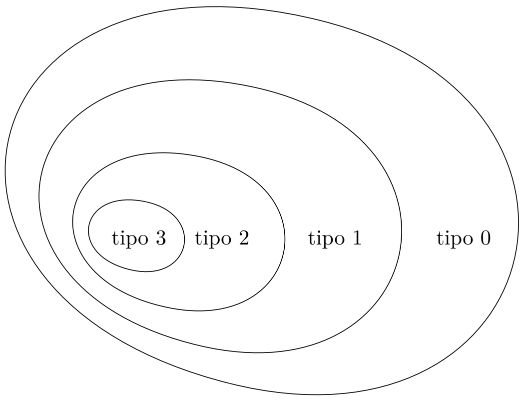


Figura 27: Gerarchia di Chomsky

tipo	*grammatica e linguaggio accettato*	*automatismo*	*forma regole*
0	non ristretta - *GG*	*TM*	*tutte*
1	dipendente dal contesto	*linear bounded automaton*	$\alpha A\beta \rightarrow \alpha\gamma\beta$
2	non contestuale - *CFG*	*NPDA*	$A \rightarrow \gamma$
3	regolare - *RG*	*FSA*	$X \rightarrow a$ o $X \rightarrow aY$

Tabella 1: Gerarchia di Chomsky

La gerarchia è composta dalle grammatiche così come segue:

Tipo 0: Include tutte le grammatiche formali
Tipo 1: Hanno regole in forma $\alpha A\beta \rightarrow \alpha\gamma\beta$

- A è un non terminale
- α, β, γ sono stringhe di terminali e non terminali
- γ è non vuota
- il contesto viene preservato *(solo il non terminale viene sostituito)*
- la regola $S \rightarrow \epsilon$ è consentita se S non appare a destra in alcuna regola

Tipo 2: Hanno regole in forma $A \rightarrow \gamma$

- A è un non terminale
- γ è una stringa di terminali e non terminali

Tipo 3: Hanno regole in forma $X \rightarrow a$ e ($X \rightarrow aY$ o $X \rightarrow Ya$)

- X, Y sono non terminali
- a è un terminale
- a può essere seguito (o preceduto) da un terminale *(le due possibilità sono mutualmente esclusive)*
- La regola $S \rightarrow \epsilon$ è consentita se S non appare a destra in alcuna regola

Le grammatiche di tipo 3 sono le **meno potenti**, mentre quelle di tipo 0 sono le **più potenti**.

6.2.1 Grammatiche lineari

Sia $G = \langle V_T, V_N, P, S\rangle$ una grammatica. Si supponga che, per ogni produzione $\alpha \rightarrow \beta \in P$:

- $|\alpha| = 1$, ossia $\alpha \in V_N$
- β sia nella forma aB, a può essere ϵ
  - $B \in V_N$, alfabeto dei non terminali
  - $a \in V_T$, alfabeto dei terminali

Allora G è una **grammatica lineare**, oppure *regolare* o di *tipo* 3 (si veda la Sezione 6.2). Le grammatiche di questa categoria hanno al massimo un non terminale nella parte destra di ognuna delle sue derivazioni.

Si riconoscono due tipi particolari di grammatiche lineari:

← Lineare **sinistra** (L-grammatica)

- Tutte le derivazioni sono nella forma $\alpha \rightarrow \alpha w$
- $|\alpha| = 1$ è vuoto o singolo non terminale
- w è una stringa di terminali

→ Lineare **destra** (R-grammatica)

- Tutte le derivazioni sono nella forma $\alpha \rightarrow w\alpha$

- w è una stringa di terminali
- $|\alpha| = 1$ è vuoto o singolo non terminale

Una **grammatica** è **regolare** (in breve *RG*) se e solo se è regolare sinistra o regolare destra. Inoltre, un **linguaggio** è **regolare** se e solo se è generato da una grammatica regolare (e quindi esiste almeno una grammatica che lo genera).

6.2.2 Grammatiche non contestuali

Sia $G = \langle V_T, V_N, P, S\rangle$ una grammatica. Se, per ogni produzione $\alpha \to \beta$ si verifica:

- $\alpha \to \beta \in P$, la produzione è contenuta in P
- $|\alpha| = 1$, α è un non terminale

allora G è una **grammatica non contestuale** o *CFG* (dall'Inglese *context-free grammar*). Questa denominazione è dovuta al fatto che la riscrittura di α non dipende dal suo contesto *(la parte della stringa che la circonda).*
Le *CFG* sono anche dette *BNF* (da *Backus-Naur Form*) e vengono impiegate per definire la sintassi di linguaggi di programmazione. Le *RG* sono anche *CFG* ma non è vero il contrario:

$$RG \subseteq CFG$$

6.2.3 Grammatiche generali

Le **grammatiche generali** (per brevità *GG* o *non ristrette*) sono tutte le grammatiche che non presentano limitazioni sulle produzioni. Corrispondono al tipo 0 della gerarchia di Chomsky (Sezione 6.2).
Sia le *RG* che le *CFG* sono **non ristrette**:

$$RG \subseteq CFG \subseteq GG$$

In questa categoria cadono le grammatiche **ricorsivamente enumerabili** e le grammatiche **ricorsive**. Per più dettagli sul significato di queste locuzioni, si consultino le Sezioni 9.11 e 9.10.

6.3 *RG* e *FSA*

Dato un *FSA* A, è possibile costruire una *R*-grammatica a esso equivalente, ossia in grado di generare lo stesso linguaggio riconosciuto da A e viceversa (Sezioni 6.3.1 e 6.3.2).
La conseguenza diretta è che *RG*, *FSA* ed espressioni regolari (Sezione 7) sono modelli diversi per descrivere la stessa classe di linguaggi.

6.3.1 Costruzione di *RG* partendo da *FSA*

Sia $A = \langle Q, I, \delta, q_0, F\rangle$ un *FSA*. È possibile costruire una *RG* $G = \langle V_N, V_T, S, P\rangle$ tale che:

- $V_T = I$, l'insieme dei **terminali** di G corrisponde all'**alfabeto di ingresso** di A
- $V_N = Q$, l'insieme dei **non terminali** di G corrisponde agli stati di A
- $S = q_0$, l'**assioma** di G corrisponde allo **stato iniziale** di A
- Gli elementi di P **(l'insieme delle produzioni di** G**)** hanno la seguente forma:
  - $B \to bC$ se e solo se $C \in \delta(B, b)$ - *transizione tra stati*
  - $B \to \epsilon$ se $B \in F$ - *transizione verso lo stato finale*
  - $\delta^*(q, x) = q'$ se e solo se $q \overset{*}{\Rightarrow} xq'$

6.3.2 Costruzione di *FSA* partendo da *RG*

Sia $G = \langle V_N, V_T, S, P\rangle$ una *RG*. È possibile costruire un *FSA* $A = \langle Q, I, \delta, q_0, F\rangle$ tale che:

- $Q = V_N \cup \{q_F\}$, gli **stati** di A sono i **non terminali** di G
- $I = V_T$, l'**alfabeto di ingresso** di A corrisponde all'**insieme di terminali** di G
- $q_0 = S$, lo **stato iniziale** di A corrisponde all'**assioma** di G
- $F = \{q_F\}$
- La **funzione di transizione** δ ha la seguente forma:
  - $\delta(A, b) = C$ se e solo se $A \rightarrow bC$ - derivazione di terminali e non terminali
  - $\delta(A, b) = q_F$ se e solo se $A \rightarrow b$ - derivazioni di terminali

6.4 *CFG* e *NPDA*

Analogamente a quanto trattato nella Sezione 6.3.1, è possibile costruire un *NPDA* partendo da una *CFG* *(context-free grammar, grammatica non contestuale)* e viceversa.
Intuitivamente, la pila contiene la parte intermedia delle produzioni, fatta di terminale e non terminali. Rispettando le regole definite della *CFG*, è possibile completare le produzioni fino ad ottenere una stringa di non terminali tramite mosse **non deterministiche** del *NPDA*.
Un esempio del funzionamento del *NPDA* che emula una *CFG* con la regola $S \Rightarrow aSb \Rightarrow aabb$ è mostrato in Figura 28.

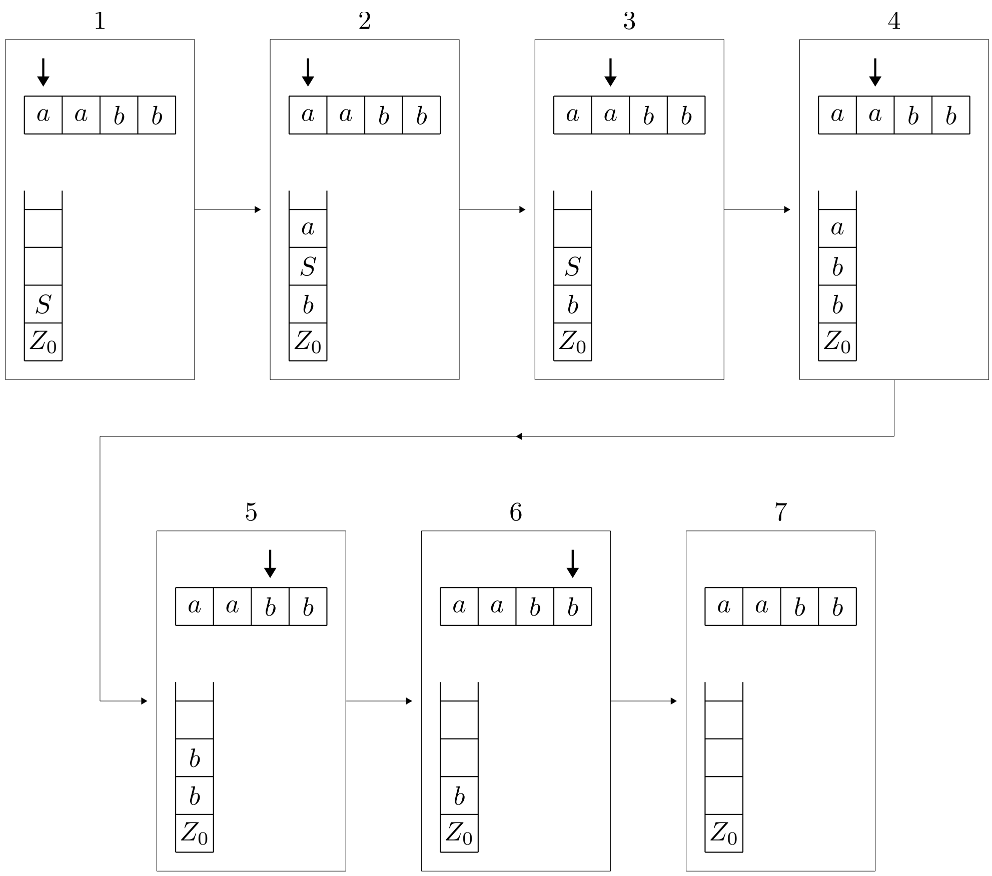


Figura 28: Analogia *CFG* e *NPDA*

6.5 *GG* e *TM*

Analogamente a quanto trattato nella Sezione 6.3.1, è possibile costruire una *TM* partendo da una *GG* *(grammatica generale)* e viceversa.

6.5.1 Costruzione di *GG* partendo da *TM*

Sia M una TM a nastro singolo. È possibile costruire una GG detta G tale che $L(G) = L(M)$ *(il linguaggio generato dalla GG e accettato dalla NTM sono equivalenti)*:

- Inizialmente, G genera tutte le stringhe di tipo $x\$X$ con
  - $x \in V_T^*$ e X
  - X è la copia di x composta solo dai simboli **non terminali**
- Successivamente, G simula le diverse configurazioni di M sfruttando la stringa a destra del simbolo \$
  - la derivazione $x\$X \stackrel{*}{\Rightarrow} x$ se e solo se x è accettata da M
  - ogni mossa di M viene **emulata** con una derivazione immediata di G
  - G ha quindi delle derivazioni della forma $x\$X \Rightarrow x\$q_0X$ *(configurazione iniziale di M)*
- Infine, per simulare le mosse di M vengono introdotte le seguenti produzioni:
  1. Se è definita $\delta(q, A) = \langle q', A', R\rangle$ si definisce in G la produzione $qA \to A'q'$
  2. Se è definita $\delta(q, A) = \langle q', A', S\rangle$ si definisce in G la produzione $qA \to q'A'$
  3. Se è definita $\delta(q, A) = \langle q', A', L\rangle$ si definisce in G la produzione $BqA \to q'BA'$, $\forall B$ dell'alfabeto di M

     $\to$ gli alfabeti di ingresso, uscita e memoria coincidono
- Per completare la costruzione è infine necessario aggiungere a M le produzioni che permettono a G di derivare da $x\$\alpha BqAC\beta$ la sola x solo nei cai in cui M giunge a una configurazione di accettazione ($x\$\alpha Bq_FAC\beta$) cancellando tutto ciò che si trova a destra di \$, esso compreso

6.5.2 Costruzione di *TM* partendo da *GG*

Sia $G = \langle V_N, V_T, S, P\rangle$ una GG. È possibile costruire una NTM M tale che $L(M) = L(G)$ *(il linguaggio accettato dalla NTM e generato dalla GG sono equivalenti)*:

- M ha un nastro di **memoria**
- La stringa di ingresso x è sul nastro di **ingresso**
- Il nastro di memoria è inizializzato con l'**assioma** Z_0S
- Il nastro di memoria in generale conterrà una **stringa** $\alpha \in V^*$:
  - Viene scandito per cercare la parte **sinistra** di una produzione P
  - Una volta trovata, M compie una scelta **non deterministica** e la parte scelta è sostituita dalla parte destra corrispondente
  - Se ad essa corrispondono **più parti destre**, ne viene scelta una in modo **non deterministico**

In questo modo vi è in G una derivazione che porta dalla stringa α alla stringa β (quindi una derivazione $\alpha \stackrel{*}{\Rightarrow} \beta$) se e solo se esiste una sequenza di mosse tale per cui

$$c_s = \langle q_s, Z_0\alpha\rangle \vdash^* \langle q_s, Z_0\beta\rangle$$

per qualche q_s (come illustrato in Figura 29).

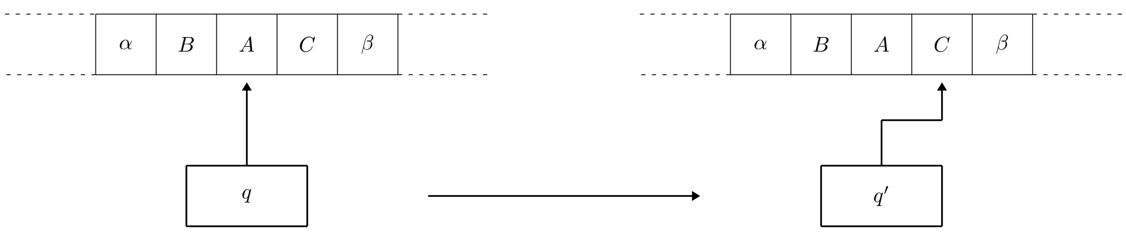


Figura 29: Derivazione $\alpha \stackrel{*}{\Rightarrow} \beta$ in una TM costruita da GG

Se il nastro contiene una stringa $y \in V_T^*$ (composta da soli simboli terminali), essa è confrontata con x:

- ✔ Se coincidono, allora x è **accettata**
- ✖ Altrimenti, questa particolare sequenza di mosse **non porta all'accettazione**

6.5.2.1 Note sulla costruzione di *NTM*

- Usare una *NTM* facilita la costruzione di una *GG* ma non è l'unico metodo ammesso per farlo.
  - Poiché il non determinismo può essere emulato da macchine deterministiche, una *TM* è sufficiente a tale scopo
- Se $x \notin L(G)$, allora M può tentare infinite computazioni, nessuna delle quali porta ad accettazione
  - alcune di queste potrebbero non terminare mai, quindi senza concludere che $x \in L(G)$
  - analogamente, non si concluderebbe se $x \notin L(G)$
- La definizione di accettazione richiede che:
  - M raggiunga una configurazione di accettazione se e solo se $x \in L$
  - essa non richiede che M termini la computazione in uno stato non finale se $x \notin L$
  - il problema del complemento è risolto e si ripresenta l'asimmetria tra risoluzione di un problema in senso positivo o negativo

7 Espressioni regolari - *RE*

Come visto fino ad ora, i linguaggi possono essere rappresentati tramite diverse **classi di modelli**, tra cui si riconoscono:

- *Insiemi*
- *Pattern*
- *Espressioni regolari*
- *Modelli operazionali*
  - Automi *(Sezione 2)*
  - Trasduttori *(Sezione 2.3)*
  - Reti di Petri
  - Diagrammi di stato
- *Modelli generativi*
  - Grammatiche *(Sezione 6)*
- *Modelli dichiarativi*
  - Logica *(Sezione 8)*

In questo capitolo si tratterà delle **espressioni regolari** (scritto come *RE* o *Regex*). Esse sono delle espressioni utilizzabili per denotare un linguaggio attraverso la scrittura delle stringhe che lo compongono.

7.1 Pattern

Prima di dare una definizione formale di *RE*, è necessario introdurre il concetto di **pattern**.

Un sistema di pattern è una tripla $\langle A, V, p\rangle$ dove:

- A è un **alfabeto**
- V è un **insieme di di variabili**

  $\rightarrow$ è definito in modo che $A \cup V = \emptyset$
- p è una stringa su $A \cup V$ detta **pattern**

Il linguaggio generato dal sistema di pattern consiste di tutte le stringhe su A ottenute da p sostituendo ogni variabile in p con una stringa su A.

Esempio: $\langle \underbrace{\{0,1\}}_{alfabeto}, \underbrace{\{v_1, v_2\}}_{variabili}, \underbrace{v_1 v_1 0 v_2}_{pattern} \rangle$

- Stringhe che iniziano con 0 ($v_1 = \epsilon$)
- Stringhe che iniziano con una stringa su A ripetuta due volte, seguita da uno 0 e da qualunque stringa (inclusa ϵ)

7.2 Sintassi e semantica delle *RE*

Dato un alfabeto di simboli terminali, mediante le seguenti regole si definiscono le **espressioni regolari** e i corrispondenti linguaggi denotati. Sono *RE* su un alfabeto Σ:

1. $\emptyset$ è una *RE* che denota il linguaggio vuoto ($\emptyset$)
2. ϵ è una *RE* che nota il linguaggio $\{\epsilon\}$
3. Ogni simbolo di σ è una *RE* che denota il linguaggio $\{\sigma\}$, $\sigma \in \Sigma$
4. Se R_1 e R_2 sono *RE*, anche $R_1 \cup R_2$, *(scritto anche come $R_1 + R_2$ o $R_1 \,|\, R_2$)* è una *RE*
5. Se R_1 e R_2 sono *RE*, anche $R_1 \cdot R_2$, *(scritto anche come R_1R_2)*, è una *RE*
6. Se R è una *RE*, lo è anche R^*
7. Nient'altro è una *RE*

Le *RE* seguono la stessa idea dei sistemi di pattern, ma con diverso potere espressivo. Le espressioni regolari sono diverse dai sistemi di pattern e hanno diverso potere espressivo.
Le *RE* **corrispondono esattamente** ai linguaggi regolari e hanno lo stesso potere espressivo di *RG* e *FSA*: per ogni *FSA* è possibile costruire la *RE* equivalente.

Per dimostare l'enunciato è sufficiente osservare che:

- Ogni linguaggio denotato da una *RE* è regolare. Infatti:
  1. i casi base sono linguaggi regolari
  2. i linguaggi regolari sono chiusi rispetto a agli operatori di **concatenazione**, **unione** e **stella di Kleene**
- Data una *RG* G, è sempre possibile trovare una *RE* r tale che $L(G) = L(r)$

7.2.1 Operatori delle *RE*

Sono definiti i seguenti **operatori** delle *RE*:

- La **concatenazione**, $\cdot$
- L'**alternativa** (detta anche *pipe*), $|$
- La **stella di Kleene** (già definita nella Sezione 1.2), $*$

  $\rightarrow$ di conseguenza, anche il **più di Kleene**, $+$
- L'**opzionalità**, ?

7.2.2 *RE* e pattern

Le espressioni regolari seguono la stessa idea dei sistemi di pattern, ma con diverso potere espressivo. Infatti, per riconoscere i linguaggi generati dai pattern servirà una *TM*, mentre per le *RE* sono sufficienti le *FSA*.
Di conseguenza, le *RE* non definiscono la stessa classe di linguaggi definita dai pattern. Inoltre le due classi non sono confrontabili, e non sono una sottoclasse dell'altra.
Per la prima volta dall'inizio del corso si osserva un caso di non confrontabilità:

$$RE \neq \text{ pattern}$$

7.2.3 *RE* `POSIX`

Lo standard `POSIX` è una API standard per i sistemi operativi UNIX/LINUX e definisce anche le *RE*. Esso include:

- I **metacaratteri** ( ) [ ] ^ \ \$ $*$ + ? { }
- La notazione $[\alpha]$ indica **un singolo carattere** $\in \alpha$
- La notazione $[\hat{}\alpha]$ indica **un qualunque simbolo** $\notin \alpha$
- Il simbolo ^ indica ϵ **all'inizio di una riga** di testo
- Il simbolo \$ indica ϵ **alla fine di una riga** di testo
- I simboli $*, +, |, (,)$ rappresentano gli **operatori** come già definiti *(a esempio in Sezione 1.3.1)*
- Il simbolo \ funge da **escape**

In aggiunta agli operatori delle *RE*, sono definiti:

- $\alpha?$ indica che α è **opzionale**

  $\rightarrow$ α appare 0 o 1 volte
- $\alpha\{n\}$ indica la **potenza** α^n

  $\rightarrow$ α appare n volte
- $\alpha\{n, m\}$ indica $\alpha^n \cup \alpha^{n+1} \cup \ldots \cup a^{m-1} \cup a^m$

  $\rightarrow$ α appare tra le m e le n volte

8 Logica nell'informatica

In questa sezione verrà analizzata la logica dal punto di vista del suo uso nell'ingegneria informatica. La logica è un *formalismo descrittivo e universale*, cioè permette di descrivere le proprietà che si vogliono ottenere (o evitare) da un sistema, senza dover per forza formalizzare anche quest'ultimo.
Grazie a questa sua proprietà, la logica può essere applicata in numerosi contesti, come:

- Le porte logiche nell'architettura dei calcolatori
- La specifica e la verifica di sistemi nell'ingegneria del software
- La definizione della semantica dei linguaggi di programmazione
- La programmazione logica
- I database

e molti altri.
All'interno del corso verranno usati la *PL* e la *FOL (introdotte nelle Sezioni 8.1 e 8.2)* per:

- Definire i linguaggi *(Sezione 8.3)*
- Specificare le proprietà di programmi *(Sezione 8.6)*
- Specificare le proprietà dei sistemi

8.1 Logica proposizionale - *PL*

8.1.1 Sintassi

Sia $\mathcal{L}$ un linguaggio della logica proposizionale.

- L'**alfabeto**, di $\mathcal{L}$ è composto da:
  - Un insieme numerabile *(finito o infinito)* di **proposizioni** (simboli di relazione nullaria): $A, B, C, \ldots$
  - Un insieme di $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
  - I **simboli di punteggiatura** (,) - *parentesi tonde*
  - I simboli dell'alfabeto sono privi di significato
- L'**insieme di formule** di $\mathcal{L}$ è il più piccolo insieme tale che:
  - Ogni *proposizione* è una **formula**
  - Se F, G sono formule, allora $\neg F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$, $F \leftrightarrow G$ sono **formule**
- Le **parentesi sono omesse** ovunque possibile usando l'ordine di precedenza:

$$\neg \quad \wedge \quad \vee \quad \rightarrow \quad \leftrightarrow$$

- Se A è una proposizione, allora A e $\neg A$ sono detti **letterali**
  - A è detto letterale **positivo**
  - $\neg A$ è detto letterale **negativo**
- se L è un letterale, allora $\overline{L}$ è il **letterale complementare** definito come $\neg A$ *se* $L = A$ o A *se* $L = \neg A$
- Informalmente, una **sotto formula** è una formula inclusa in un'altra formula
- L'insieme $\tau(F)$ delle sotto formule di $\mathcal{F}$ è definito come il più piccolo insieme di formule tale che:
  - $F \in \tau(F)$
  - se $\neg G \in \tau(F)$, allora $G \in \tau(f)$
  - se $G \wedge H, G \vee H, G \rightarrow H, G \leftrightarrow H$ appartengono a $\tau(F)$, allora $H, G \in \tau(F)$

8.1.2 Semantica

La **semantica** è introdotta per assegnare un significato alle formule.
Nella logica proposizionale, ogni formula può corrispondere a un solo valore di verità (`vero` o `falso`): è quindi una logica **a due valori**.
Un'**interpretazione** I è una funzione totale dell'insieme di proposizioni ai valori di verità. Ogni interpretazione può essere convenientemente rappresentata come l'insieme delle proposizioni vere.
Con la notazione $I \vDash F$ si indica che I rende vera F.

Definizioni:

- Se $I \vDash F$, allora I è un **modello** di F. Questa nozione può essere estesa agli insiemi di formule
- F è **valida** (detta anche **tautologia**) se e solo se per ogni interpretazione I vale che $I \vDash F$

  → in questo caso si può anche scrivere $\vDash F$

- F è **soddisfacibile** se e solo se esiste un'interpretazione I tale per cui $I \vDash F$
- F è **falsificabile** se e solo se esiste un'interpretazione I tale per cui $I \nvDash F$
- F è **insoddisfacibile** se e solo se per ogni interpretazione I vale $I \nvDash F$
- F è **contingente** se e solo se è sia *soddisfacibile* sia *falsificabile*
- Ogni formula del tipo $F \wedge \neg F$ è detta una **contraddizione**, indicata con $\bot$
- La formula $F \vee \neg F$ è detta **principio del terzo escluso**, indicata con $\top$
- Un insieme di formule $\mathcal{F}$ **comporta logicamente** una formula G (o G è una *conseguenza logica* di $\mathcal{F}$) se ogni modello di $\mathcal{F}$ è anche un modello di G e si scrive con $\mathcal{F} \vDash G$. *Esempio:*
  - $\{A, A \to B\} \vDash B, \{A, A \to B\} \vDash B \wedge C, \{A, A \to B\} \nvDash C$
- Per determinare sistematicamente se una formula segua da un insieme di formule si possono usare le **tabelle di verità**

Si considerino:

- La *proposizione* A
- La *formula* F
- L'*interpretazione* I

Allora:

$$I \models A \quad sse \quad I(A) = \texttt{vero}$$

Di conseguenza:

$$\begin{aligned}
I \vDash \neg F &\quad \textbf{sse} \quad I \nvDash F \\
I \vDash F \wedge G &\quad \textbf{sse} \quad I \vDash F \textbf{ e } I \vDash G \\
I \vDash F \vee G &\quad \textbf{sse} \quad I \vDash F \textbf{ o } I \vDash G \\
I \vDash F \to G &\quad \textbf{sse} \quad I \nvDash F \textbf{ o } I \vDash G \\
I \vDash F \leftrightarrow G &\quad \textbf{sse} \quad I \vDash F \to G \textbf{ e } I \vDash G \to F
\end{aligned}$$

Alternativamente, i connettivi possono anche essere esplicitati sotto forma di tabella (Tabella 2).

F	G		$\neg F$	$F \wedge G$	$F \vee G$	$F \to G$	$F \leftrightarrow G$
vero	vero		falso	vero	vero	vero	vero
vero	falso		falso	falso	vero	falso	falso
falso	vero		vero	falso	vero	vero	falso
falso	falso		vero	falso	falso	vero	vero

Tabella 2: Tabella di verità

8.1.2.1 Esempio di interpretazione

Se:

- $I_1 = \{A, C\}$
- $I_2 = \{C, D\}$
- $F = (A \vee B) \wedge (C \vee D)$

Allora:

- $I_1 \vDash F$
- $I_2 \nvDash F$

8.1.3 Forme normali

- Due formule F, G sono **semanticamente equivalenti** se e solo se vale sia $F \vDash G$ che $G \vDash F$ e si scrive $F \equiv G$
- Quando G, sotto-formula di F, viene sostituita con una formula H, la formula risultante viene indicata con $F[G \backslash H]$
- Un insieme di connettivi è detto **funzionalmente completo** se e solo se qualunque formula proposizionale può essere trasformata in una formula semanticamente equivalente che contiene solo connettivi dell'insieme
  - → L'insieme $\{\neg, \wedge\}$ è funzionalmente complesso
  - → Esistono altri due connettivi booleani che singolarmente sono funzionalmente completi: `nand` e `nor`

Equivalenze notevoli:

$(F \wedge F) \equiv F$	idempotenza di $\wedge$
$(F \vee F) \equiv F$	idempotenza di $\vee$
$(F \wedge G) \equiv (G \wedge F)$	commutatività di $\wedge$
$(F \vee G) \equiv (G \vee F)$	commutatività di $\vee$
$(F \wedge (G \wedge H)) \equiv ((F \wedge (G \wedge H)$	associatività di $\wedge$
$(F \vee (G \vee H)) \equiv ((F \vee (G \vee H)$	associatività di $\vee$
$((F \wedge G) \vee F) \equiv F$	assorbimento
$((F \vee G) \wedge F) \equiv F$	assorbimento
$(F \wedge (G \vee H)) \equiv ((F \wedge G) \vee (F \wedge H))$	distributività
$(F \vee (G \wedge H)) \equiv ((F \vee G) \wedge (F \vee H))$	distributività
$(\neg(\neg F)) \equiv F$	doppia negazione
$(\neg(F \wedge G)) \equiv (\neg F \vee \neg G)$	legge di De Morgan
$(\neg(F \vee G)) \equiv (\neg F \wedge \neg G)$	legge di De Morgan
$(F \leftrightarrow G) \equiv (F \to G) \wedge (G \to F)$	equivalenza
$(F \to G) \equiv (\neg F \vee G)$	implicazione materiale
$(F \to G) \equiv (\neg G \to \neg F)$	implicazione contronominale

Il *teorema di sostituzione* e le *equivalenze notevoli* possono essere utilizzate per introdurre le cosiddette **forme normali**:

- Una formula è in **forma normale negativa** se e solo se è composta solo da letterali, congiunzioni ($\wedge$) e disgiunzioni ($\vee$)
- Una formula è in **forma normale congiuntiva** (*CNF*) se e solo se ha la forma $C_1 \wedge C_2 \wedge \ldots \wedge C_n$ dove ogni C_i è la disgiunzione di letterali
  - → $C_1 \wedge C_2 \wedge \ldots \wedge C_n \equiv C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge \top$, quindi si può affermare che $\top$ è in *CNF* con $n = 0$
- Una formula è in **forma normale disgiuntiva** (*DNF*) se e solo se ha la forma $C_1 \vee C_2 \vee \ldots \vee C_n$ dove ogni C_i è la congiunzione di letterali
  - → $D_1 \vee D_2 \vee \wedge \vee D_n \equiv D_1 \vee D_2 \vee \ldots \vee D_n \vee \bot$, quindi si può affermare che $\bot$ è in *DNF* con $n = 0$
- I C_i sono detti **clausole**, mentre i D_i sono detti **clausole duali**. Normalmente si usa una notazione insiemistica

8.1.4 Sistemi formali - *calculi*

Un *sistema formale (assiomatico deduttivo)*, oppure **calculus** in Inglese, consiste in un insieme di assiomi e un insieme di regole di inferenza che producono conseguenze logiche all'interno di una logica. Questi elementi definiscono una *relazione di derivabilità* (detta anche **dimostrabilità**) tra un insieme di formule $\mathcal{F}$ e una formula G.

Se una formula G può essere ottenuto da $\mathcal{F}$ applicando solo regole di inferenza e assiomi, si scrive che $\mathcal{F} \vdash G$. Idealmente, la relazione di derivabilità dovrebbe essere **corretta** (cioè se $\mathcal{F} \vdash G$ allora $\mathcal{F} \models G$) e **completa** (cioè se $\mathcal{F} \models G$ allora $\mathcal{F} \vdash G$).
Se una formula F piò essere derivata in una teoria $\mathcal{F}$ usando gli assiomi e le regole d'inferenza di un sistema, allora diciamo che F è un **teorema**.

8.2 Logica del primo ordine - *FOL*

La logica proposizionale (vista in Sezione 8.1) ha molte applicazioni, ma il suo potere espressivo è ristretto. Infatti, frasi come *"tutti gli esseri umani sono mortali"* e *"ogni bambino ha dei genitori"* possono essere espresse come proposizioni in un modo che non cattura le relazioni sottintese tra esseri umani, mortali, bambini e genitori.
Frege ha sviluppato la *FOL* (**logica del prim'ordine** o **logica dei predicati**) nel 1879, estendendo la logica proposizionale con *funzioni*, *variabili* e *quantificatori*.

- Dal punto di vista *epistemologico* (stati della conoscenza) sia *PL* che *FOL* considerano **verità** e **falsità** (ossia quando la logica assume i valori di `vero` e `falso`).
- Dal punto di vista *ontologico* (ciò che esiste), *PL* considera i *fatti*, mentre *FOL* considera anche:
    - **oggetti**, quali: *persone, case, numeri, corsi, ...*
    - **proprietà** e **relazioni**, quali: *essere rosso, essere felice, essere più grande di, essere parte di, ...*
    - **funzioni**, quali: *padre di, età di, successore di, ...*

8.2.1 Sintassi

- L'**alfabeto** di un linguaggio della *FOL* $\mathcal{L}$ è composto da:
    - Un insieme infinito numerabile di **variabili**: $X, Y, Z, \ldots$
    - Un insieme di simboli di **funzione**: $f, g, \ldots$
    - Un insieme di simboli di **predicati**: $p, q, r, \ldots$
    - I seguenti **connettivi**: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
    - I seguenti **quantificatori**: $\exists, \forall$
    - I **simboli di punteggiatura**: $(,), ", "$ - *parentesi tonde e virgola*
- Ogni simbolo di funzione e relazione ha una **arietà** fissata che indica il numero di argomenti a esso associati.
    - la notazione p/n indica che il *predicato* (o *funzione*) p ha **arietà** n
    - un *predicato* (o *funzione*) con arietà zero è detto **nullario**
    - un *predicato* nullario $p()$ è scritto semplicemente come p
    - una *funzione* nullaria $c()$ è semplicemente scritta come c
- Le funzioni nullarie sono dette **costanti**, i predicati nullari sono detti **proposizioni**.
- I **termini** denotano tutti gli oggetti che $\mathcal{L}$ può trattare
    - sono definiti induttivamente come segue:
        1. Ogni *variabile* è un *termine*
        2. Se f/n è un simbolo di funzione e $t_1, \ldots t_n$ sono termini, allora $f(t_1, \ldots t_n)$ è un *termine*
    - i **termini generici** sono tipicamente indicati con $s, t, \ldots$
- L'insieme di **formule della *FOL*** è definito induttivamente come il più piccolo insieme tale che:
    - Se p/n è un simbolo di relazione e $t_1, \ldots t_n$ sono termini, allora $p(t_1, \ldots, t_n)$ è una formula detta **formula atomica** o **atomo**
    - se F, G sono formule e X è una variabile, allora anche

    $$\neg F \quad F \wedge G \quad F \vee G \quad F \rightarrow G \quad F \leftrightarrow G \quad \exists X F \quad \forall X F$$

    sono formule
- I **letterali** sono atomi (letterali positivi) o atomi negati (letterali negativi)
- Le **parentesi sono omesse** ovunque possibile usando l'ordine di precedenza

$$\exists \quad \forall \quad \neg \quad \wedge \quad \vee \quad \rightarrow \quad \leftrightarrow$$

- Per convenienza:

$$\begin{array}{lll} \exists X_1 (\ldots (\exists X_n(F)) \ldots) & \text{è abbreviato come} & \exists X_1, \ldots, X_n(F) \\ \forall X_1 (\ldots (\forall X_n(F)) \ldots) & \text{è abbreviato come} & \forall X_1, \ldots, X_n(F) \end{array}$$

- Se $QX(F)$ è una formula e Q è un quantificatore, allora F si dice **ambito** di Q e Q è **applicato** a F
- Un'occorrenza di una variabile in una formula è **legata** se e solo se la sua occorrenza è entro l'ambito di un quantificatore che impiega quella variabile
  - → è legata al quantificatore di ambito più piccolo che la rende legata
  - → in caso contrario, è **libera**
- Una formula è **chiusa** se e solo se non contiene occorrenze libere di variabili
  - La valutazione è omessa quando si considerano formule chiuse
- Un'interpretazione $\mathcal{I}$ è detta **modello** per F sse per ogni valutazione Φ si ha $\mathcal{I} \vDash_\Phi F$
  - Se $\mathcal{F}$ è un insieme di formule, un'interpretazione è un **modello** di $\mathcal{F}$ sse è un modello $\forall F \in \mathcal{F}$

8.2.1.1 Esempi di traduzioni in *FOL*

- *"tutti gli esseri umani sono mortali"* diventa

$$\forall X\, (h(X) \rightarrow m(X))$$

- *"ogni bambino ha dei genitori"* diventa

$$\forall X\ (c(X) \rightarrow (\exists Y \exists Z p(X, Y, Z)))$$

8.2.1.2 Osservazioni sulle traduzioni in *FOL*

- Il connettivo principale usato con $\forall$ è $\rightarrow$
  - *"al Polimi sono tutti brillanti"* `(bleah)` si può tradurre con

$$\forall X\, (at(X, polimi) \rightarrow smart(X))$$

  - notare che la formula

$$\forall X\, (at(X, polimi) \wedge smart(X))$$

    significa che tutti sono al Polimi e tutti sono brillanti
- Analogamente, il connettivo principale da usare con $\exists$ è $\wedge$
  - *"al Polimi qualcuno è brillante"* si può tradurre con

$$\exists X\, (at(X, polimi) \wedge smart(X))$$

  - notare che la formula

$$\forall X\, (at(X, polimi) \rightarrow smart(X))$$

    significa che c'è qualcuno che o non è al Polimi o è brillante (o entrambe)

8.2.2 Semantica

Come per *PL*, anche *FOL* ha una semantica a due valori di verità basata sulla nozione di interpretazione.
Una interpretazione $\mathcal{I}$ di un alfabeto $\mathcal{A}$ è un **dominio** non vuoto D (indicato anche come $|I|$) a una funzione che associa:

- Ogni costante $c \in \mathcal{A}$ a un elemento $c_\mathcal{I} \in D$
- Ogni simbolo di funzione $f/n \in \mathcal{A}$ a una funzione $f_\mathcal{I} : D^n \rightarrow D$
- Ogni simbolo di predicato $p/n \in \mathcal{A}$ a una relazione $p_\mathcal{I} \subseteq \underbrace{D \times \ldots \times D}_{n \text{ volte}}$

Tuttavia, prima di assegnare un significato alle formule, va definito il significato di ciascun termine. Poiché essi possono contenere variabili, occorre una **valutazione** (o **stato**) ossia una funzione dalle variabili di $\mathcal{A}$ a $|\mathcal{I}|$.
Il **significato** $\Phi_{\mathcal{I}}(t)$ di un termine t nell'interpretazione $\mathcal{I}$ e valutazione Φ è quindi definito induttivamente come:

1. $c_{\mathcal{I}}$ se t è una costante c
2. $\Phi(X)$ se t è una variabile X
3. $f_{\mathcal{I}}(\Phi_{\mathcal{I}}(t_1), \ldots, \Phi_{\mathcal{I}}(T_n))$ se t è della forma $f(t_1, \ldots, t_n)$

8.2.2.1 Proprietà delle valutazioni

Si considerino:

- Una *valutazione* Φ
- Una *variabile* X
- Un'*interpretazione* $\mathcal{I}$, $c_{\mathcal{I}} \in |\mathcal{I}|$

Allora $\Phi[X \mapsto c_{\mathcal{I}}]$ è una valutazione analoga a Φ che *mappa* X in $c_{\mathcal{I}}$.
Il significato di una formula è un valore di verità che è definito induttivamente come segue:

1. Si indica $I \vDash_{\Phi} F$ una formula F vera rispetto a $\mathcal{I}$ e Φ
2. Si applicano le seguenti uguaglianze:

$$
\begin{array}{lll}
\mathcal{I} \vDash_{\Phi} p(t_1, \ldots, t_n) & \textit{sse} & \langle \Phi_{\mathcal{I}}(t_1), \ldots, \Phi_{\mathcal{I}}(t_n) \rangle \in P_{\mathcal{I}} \\
\mathcal{I} \vDash_{\Phi} (\neg F) & \textit{sse} & \mathcal{I} \nvDash_{\Phi} F \\
\mathcal{I} \vDash_{\Phi} (F \wedge G) & \textit{sse} & \mathcal{I} \vDash_{\Phi} F \ \textit{e}\ \mathcal{I} \vDash_{\Phi} G \\
\mathcal{I} \vDash_{\Phi} (F \vee G) & \textit{sse} & \mathcal{I} \vDash_{\Phi} F \ \textit{o}\ \mathcal{I} \vDash_{\Phi} G \\
\mathcal{I} \vDash_{\Phi} (F \rightarrow G) & \textit{sse} & \mathcal{I} \nvDash_{\Phi} F \ \textit{o}\ \mathcal{I} \vDash_{\Phi} G \\
\mathcal{I} \vDash_{\Phi} (F \leftrightarrow G) & \textit{sse} & \mathcal{I} \vDash_{\Phi} (F \rightarrow G) \ \textit{e}\ \mathcal{I} \vDash_{\Phi} (G \rightarrow F) G \\
\mathcal{I} \vDash_{\Phi} (\forall X(F)) & \textit{sse} & \mathcal{I} \vDash_{\Phi[X \mapsto c_{\mathcal{I}}]} F \ \forall c_{\mathcal{I}} \in |\mathcal{I}| \\
\mathcal{I} \vDash_{\Phi} (\exists X(F)) & \textit{sse} & \mathcal{I} \vDash_{\Phi[X \mapsto c_{\mathcal{I}}]} F \ \exists c_{\mathcal{I}} \in |\mathcal{I}|
\end{array}
$$

8.2.2.2 Da *PL* a *FOL*

La relazione di conseguenza logica *vDash* tra insiemi di formule e formule può essere esteso anche a *FOL*, così come i concetti di **validità, soddisfacibilità, falsificabilità,, contingenza** e **insoddisfacibilità** (visti nella Sezione 8.1.2).
Analogamente, anche le equivalenze mostrate per *PL* (Sezione 8.1.3) possono essere estese a *FOL*, con l'aggiunta di:

$$
\begin{array}{rl}
\forall X(F) \equiv \neg(\exists(\neg F)) & \textit{dualità dei quantificatori } 1 \\
\exists X(F) \equiv \neg(\forall(\neg F)) & \textit{dualità dei quantificatori } 2 \\
\forall X(F) \wedge (\forall X)G \equiv \forall X(F \wedge G) & \\
\exists X(F) \wedge (\exists X)G \equiv \exists X(F \vee G) & \\
(\forall X)(\forall Y)F \equiv (\forall Y)(\forall X)F & \\
(\exists X)(\exists Y)F \equiv (\exists Y)(\exists X)F & \\
(\forall X(F)) \wedge G \equiv \forall X(F \wedge G) & \textit{solo se } X \textit{ non è libera in } G \\
(\forall X(F)) \vee G \equiv \forall X(F \vee G) & \textit{solo se } X \textit{ non è libera in } G \\
(\exists X(F)) \wedge G \equiv \exists X(F \wedge G) & \textit{solo se } X \textit{ non è libera in } G \\
(\exists X(F)) \vee G \equiv \exists X(F \vee G) & \textit{solo se } X \textit{ non è libera in } G
\end{array}
$$

8.3 Logica e linguaggi

La logica può aiutare a descrivere i linguaggi: gli insiemi di stringhe, infatti, possono essere visti come abbreviazioni di formule *FOL*. Tramite formalismi matematici, infatti, è possibile mostrare i linguaggi in modo descrittivo focalizzandosi sulle proprietà rilevanti delle stringhe anziché sul modo in cui esse sono generate o riconosciute. Per raggiungere tale obiettivo, bisogna analizzare:

- **Cosa** va descritto
- **Come** le varie parti vanno definite
- **Quali primitive** possono essere assunte

Una volta che la lingua viene definita in modo logico, occorrerebbe definire tutti i predicati e le funzioni non elementari, quali $=, >, +, -, \times, *, \cdot, \ldots$. Per evitare di doverlo fare, si fa normalmente riferimento a logiche più ristrette, con sintassi specifica già predisposta.

8.3.1 Esempi di descrizione della lingua tramite *FOL*

8.3.1.1 Esempio 1

L'insieme:

$$\{a^n b^n \mid n \geq 0\}$$

può essere vista come una abbreviazione di:

$$x \in L \Leftrightarrow \exists\, n \mid n \geq 1 \land x = a^n b^n$$

All'interno di questa formula si possono riconoscere:

- I **predicati** $\in L, \geq, =$
- Le **funzioni** di *concatenazione, elevamento a potenza*

Come già enunciato, sarebbe necessario definire ciascuno di questi elementi sopra elencati. Per esempio, x^n può essere descritto *ricorsivamente* come:

$$\forall\, n\, \forall\, x\, ((n = 0 \Rightarrow x^n = \epsilon) \land (n > 0 \Rightarrow x^n = x^{n-1} \cdot x))$$

8.3.1.2 Esempio 2

Sia L il linguaggio definito come:

$$L = a^* b^*$$

Quindi L è il linguaggio delle stringhe su $\{a, b\}$ che iniziano con a. Più precisamente, fanno parte di L le stringhe:

- vuota ϵ
- composta da un carattere a e un suffisso appartenente a L
- composta da un prefisso appartenente a L e da un carattere b

Questo linguaggio può quindi essere espresso in *FOL* come:

$$x \in L \Leftrightarrow (x = \epsilon) \lor (\,\exists\, y \mid x = at \land y \in L) \lor (\,\exists\, y \mid x = yb \land y \in L)$$

8.3.1.3 Esempio 3

Sia L il linguaggio delle stringhe su $\{a, b, c\}$ definito come:

$$L = a^* b^* c^*$$

Quindi L è il linguaggio delle stringhe su $\{a, b, c\}$ con tutte le a all'inizio, poi tutte le b e poi tutte le c.
Tra i molti modi possibili di rappresentare formalmente L, si consideri il seguente: è possibile definire due linguaggi *"ausiliari"* più semplici $L_1 = a^* b^*, L_2 = b^* c^*$ e affermare che una stringa appartiene a L se:

- è in L_1
- è in L_2
- composta da un carattere a e un suffisso appartenente a L
- composta da un prefisso appartenente a L seguito da un carattere c

Questo linguaggio può quindi essere espresso in *FOL* come:

$$x \in L \Leftrightarrow (x \in L_1) \lor (x \in L_2) \lor \exists\, y\,((x = ay \land y \in L) \lor (x = yc \land y \in L))$$

in cui L_1, L_2 sono definite come nell'esempio precedente (Sezione 8.3.1.2).

8.3.1.4 Esempio 4

Sia L il linguaggio delle stringhe su $\{a, b\}$ in cui il numero di a sia uguale al numero di b.
Per definire questo linguaggio in *FOL* si può introdurre la funzione di *arietà* 2 $\#(a, x)$, che conta il numero di apparizioni del simbolo a nella stringa x. Questa funzione è definita formalmente dalla formula:

$$(x = \epsilon \Rightarrow \#(a, x) = 0) \land \forall\, y\,((x = ay \Rightarrow \#(a, x) = \#(a, y) + 1) \land (x = by \Rightarrow \#(a, x) = \#(a, y))$$

la cui definizione dipende dall'alfabeto.

Questo linguaggio può quindi essere espresso in *FOL* come:

$$x \in L \Leftrightarrow x \in \{a, b\}^* \land \#(a, x) = \#(b, x)$$

8.3.2 Osservazioni sulla formulazione logica

Non esiste una *"formula magica"* o un sistema di risoluzione unico per ottenere la descrizione in *FOL* di un linguaggio, ma può essere utile evidenziare delle linee guida:

- Dall'esempio nel Paragrafo 8.3.1.3 si ricava che, quando si rivolge l'attenzione all'ordine con cui si susseguono le lettere in un linguaggio, si può far ricorso a formule in cui le stringhe siano decomposte nella concatenazione di sottostringhe, appartenenti ad altri linguaggi, eventualmente definiti *ricorsivamente*
- Quando è necessario contare le occorrenze di alcune lettere, risulta utile definire una funzione in grado di contare i simboli a cui si è interessati, come nell'esempio nel Paragrafo 8.3.1.4

8.4 Logica monadica del primo ordine - *MFO*

Dato un alfabeto di ingresso Σ, le formule sulla logica monadica del primo ordine (o MFO per brevità) sono costruite dai seguenti elementi:

- **Variabili del primo ordine**
  - rappresentate da lettere minuscole $(x, y, \ldots)$
  - interpretate sull'insieme di numeri positivi $\mathbb{N}$
- **Predicati monadici** (unari), uno per ogni simbolo di Σ
  - rappresentate come $(a(\cdot), b(\cdot), \ldots)$
  - $a(x)$ è valutata come `vero` in una stringa w se e solo se il carattere di w in posizione x è a
- La **relazione di minore** $<$ tra numeri naturali
- Le normali proposizioni connettive e i quantificatori di primo ordine

Più precisamente, sia $\mathcal{V}$ un insieme finito di variabili del primo ordine, e sia Σ un alfabeto. Le formule ben formate (*WFF*, dall'Inglese *well formed formula*) della logica *MFO* sono definite secondo la seguente sintassi:

$$\phi \coloneqq a(x) \mid x < y \mid \neg\phi \mid \phi \lor \phi \mid \exists\, x(\phi)$$

con $a \in \Sigma,\ x, y \in \mathcal{V}$.

Sono verificate le seguenti definizioni di **connettivi proposizionali**:

$$
\begin{aligned}
\phi_1 \land \phi_2 &\triangleq \neg(\neg\phi_1 \lor \neg\phi_2) \\
\phi_1 \lor \phi_2 &\triangleq \neg(\neg\phi_1 \land \neg\phi_2) \\
\phi_1 \Rightarrow \phi_2 &\triangleq \neg\phi_1 \lor \phi_2 \\
\phi_1 \Leftrightarrow \phi_2 &\triangleq (\phi_1 \Rightarrow \phi_2) \land (\phi_2 \Rightarrow \phi_1) \\
\forall\, x(\phi) &\triangleq \nexists x(\neg\phi)
\end{aligned}
$$

I seguenti **quantificatori del primo ordine**:

$$
\begin{aligned}
x \geq y &\triangleq \neg(x < y) \\
x \leq y &\triangleq \neg(x > y) \\
x = y &\triangleq x \leq y \land y \leq x \\
x \neq y &\triangleq \neg(x = y) \\
x > y &\triangleq y < x
\end{aligned}
$$

E le seguenti definizioni di:

- **Costante**: $x = 0 \triangleq \forall\, y \neg(y < x)$
- **Successore** di un numero naturale: $\text{succ}(x, y) \triangleq x < y \neg \exists\,(x < z \lor z < y)$
- **Somma** di valori costanti: $y = x + k \triangleq \exists\, z_0, \dots, z_k(z_0 = x \land \text{succ}(z_0, z_1) \land \dots \text{succ}(z_{k-1}, z_k) \land y = z_k)$
- **Primo**: $\text{first}(x) \triangleq \neg\exists\, y\,(y < x)$
  - equivalente a $x = 0$
- **Ultimo**: $\text{first}(x) \triangleq \neg\exists\, y\,(y > x)$
- **Sottrazione** di valori costanti: $y = x - k \triangleq x = y + k$

  $\rightarrow$ k è una costante in $\mathbb{N}$

8.4.1 Esempi di *MFO*

- Formula che è vera se tute e sole le parole il cui primo simbolo esiste ed è a:

$$\exists\, x\,(x = 0 \land a(x))$$

- Formula che è vera su tutte le parole in cui ogni a è seguita da una b:

$$\forall\, x\ (a(x) \Rightarrow \exists\, y\,(y = \text{succ}(x) \land b(y)))$$

- Parole non vuote il cui ultimo simbolo è a:

$$\exists\, x\,(\text{last}(x) \land a(x))$$

- Parole di almeno 3 simboli in cui il terzultimo simbolo è a:

$$\exists\, x\,(a(x) \land \exists\, y\,(y = x + 2 \land \text{last}(y)))$$

8.4.2 Semantica

Una formula *MFO* è interpretata su una stringa $w \in \Sigma^+$ rispetto all'assegnazione $v : \mathcal{V} \rightarrow U$ con $U = \{0, \dots, |w| - 1\}$, che mappa $\mathcal{V}$ a una posizione nella stringa w.
La relazione di assegnamento (indicata con $\models$) è definita nel modo seguente:

$$
\begin{aligned}
&w, v \models a(x) && \textit{sse}\ w = uav, |u| = v(x) \\
&w, v \models x < y && \textit{sse}\ v(x) < v(y) \\
&w, v \models \neg\Phi && \textit{sse}\ w, v \not\models \Phi \\
&w, v \models \Phi_1 \land \Phi_2 && \textit{sse}\ w, v \models \Phi_1 \land w, v \models \Phi_2 \\
&w, v \models \forall\, x(\Phi) && \textit{sse}\ w, v' \models \Phi\, \forall\, v', v'(y) = v(y), y \neq x
\end{aligned}
$$

Data una stringa ϕ, il linguaggio $L(\phi)$ è definito come:

$$L(\phi) = \left\{ w \in \Sigma^+ \mid \exists\, v : w, v \vDash \phi \right\}$$

8.4.3 Proprietà di *MFO*

I linguaggi esprimibili mediante *MFO* sono **chiusi** rispetto a:

- **Unione**
- **Intersezione**
- **Complemento**

Mentre non **non sono chiusi** rispetto alla stella di Kleene. Infatti, i linguaggi definiti tramite questa logica prendono il nome di *star-free*, cioè definibili unicamente tramite unione, intersezione, complemento e concatenazione di linguaggi finiti.

- *MFO* è strettamente meno potente degli *FSA*
  - data una formula *MFO* si può sempre costruire un *FSA* equivalente
  - L_P può invece essere riconosciuto facilmente mediante *FSA*
- In *MFO* non è possibile esprimere il linguaggio L_P rappresentante tutte e sole le parole di lunghezza pari con $l = \{a\}$
  - La formula *MFO* $a(0) \wedge a(1) \wedge \text{last}(1)$ definisce il linguaggio L_{P2} fatto dalla sola parola $\{aa\}$ di lunghezza 2
  - Poiché $L_P = L_{P2}^*$, a causa della mancata chiusura rispetto alla stella di Kleene, si spiega perché

8.5 Logica monadica del secondo ordine - *MSO*

Per ottenere lo stesso potere espressivo degli *FSA* è necessario poter quantificare sui predicati monadici.
Quindi la **logica monadica del secondo ordine** è costruita sugli stessi elementi della *MFO* (visti nella Sezione 8.4) con in aggiunta:

- **Variabili del secondo ordine**
  - rappresentate da lettere maiuscole $X, Y, \ldots$
  - interpretate su *insiemi* di numeri naturali

Le formule ben formate *(WFF)* della logica *MSO* sono definite secondo la seguente sintassi:

$$\phi \coloneqq a(X) \mid X(x) \mid x < y \mid \neg\phi \mid \phi \vee \phi \mid \exists\, x(\phi) \mid \exists\, X(\phi)$$

con $a \in \Sigma,\ x, y \in \mathcal{V}_1, X \in \mathcal{V}_2$.

Le definizioni della logica *MFO* sono sempre valide, con l'aggiunta di:

$$\begin{aligned}
x \in X &\triangleq X(x) \\
X \subseteq Y &\triangleq \forall\, x\,(x \in X \Rightarrow x \in Y) \\
X = Y &\triangleq (X \subseteq Y) \wedge (Y \subseteq X) \\
X \neq Y &\triangleq \neg(X = Y)
\end{aligned}$$

con $a \in \Sigma,\ x, y \in \mathcal{V}_1,\ X, Y \in \mathcal{V}_2$.

8.5.1 Semantica

Una formula *MSO* è interpretata su una stringa $w \in \Sigma^+$ rispetto alle assegnazioni:

$$v_1 : \mathcal{V}_1 \to \{0, \ldots, |w| - 1\}$$
$$v_2 : \mathcal{V}_2 \to \wp(\{0, \ldots, |w| - 1\})$$

dove:

- v_1 mappa ogni variabile di prim'ordine di $\mathcal{V}_1$ a una posizione nella stringa w
- v_2 mappa ogni variabile di second'ordine di $\mathcal{V}_2$ a un insieme di posizioni della stringa w

Quindi, la relazione di assegnamento per le formule della logica *MSO* è definita nel modo seguente:

$$
\begin{aligned}
&w, v_1, v_2 \models a(x) && \text{sse } w = w_1 a w_2 \wedge |w_1| = v_1(x) \\
&w, v_1, v_2 \models X(x) && \text{sse } v_1(x) \in v_2(X) \\
&w, v_1, v_2 \models x < y && \text{sse } v_1(x) < v_1(y) \\
&w, v_1, v_2 \models \neg\phi && \text{sse } w, v_1, v_2 \not\models \phi \\
&w, v_1, v_2 \models \phi_1 \vee \phi_2 && \text{sse } w, v_1, v_2 \models \phi_1 \vee w, v_1, v_2 \models \phi_2 \\
&w, v_1, v_2 \models \exists x(\phi) && \text{sse } w, v_1', v_2 \models \phi \text{ per un } v_1',\ v_1'(y) = v_1(y) \,\forall\, y \in \mathcal{V}_1 \backslash \{x\} \\
&w, v_1, v_2 \models \exists X(\phi) && \text{sse } w, v_1, v_2' \models \phi \text{ per un } v_2',\ v_2'(Y) = v_2(Y) \,\forall\, Y \in \mathcal{V}_2 \backslash \{X\}
\end{aligned}
$$

8.5.2 Espressività della logica *MSO*

Poiché ogni formula *MFO* è anche formula di *MSO*, e poiché è risultato impossibile descrivere il linguaggio L_P (come nella Sezione 8.4.3), si deduce immediatamente che la logica *MSO* è più espressiva della logica *MFO*.
Per lo stesso motivo, si può dimostrare che la *MSO* ha la stessa espressività degli *FSA*, e quindi per ogni *FSA* sarà possibile costruire una formula *MSO* e viceversa.

8.5.2.1 Da *FSA* a *MSO*

In generale, grazie alle quantificazioni del second'ordine è possibile trovare, per ogni *FSA*, una formula *MSO* equivalente.
L'idea generale della costruzione consiste nell'usare una variabile di secondo ordine X_q per ogni stato q dell'*FSA* M. Il valore di ciascun X_q corrisponde all'insieme delle posizioni di tutti i caratteri che M può leggere in una transizione partendo dallo stato q.
Assumendo che l'insieme di stati di M sia $Q = \{0, 1, \ldots, k\}$ per un qualsiasi k, con 0 che indica lo stato iniziale, la definizione di M che riconosce il linguaggio L è data dalla congiunzione di molteplici condizioni, ognuna traducente una parte della definizione di M.

8.5.2.2 Da *MSO* a *FSA*

Data una formula *MSO* Φ, è possibile costruire un *FSA* che accetta un linguaggio L definito da Φ tramite il teorema di *Büchi Elgot Trakhtenbrot*.
I passaggi che portano alla costruzione non sono oggetto di questo corso e non saranno mostrati.

8.6 Precondizioni e postcondizioni

Quando si programma una funzione è importante definire precisamente cosa fa, senza necessariamente specificare come lo fa. Questo è lo scopo di **precondizioni** e **postcondizioni**.

A questo scopo viene introdotta la **notazione di Hoare**:

$$\{\text{Precondizione: } \mathit{Pre}\}$$
$$\text{Funzione } F$$
$$\{\text{Postcondizione: } \mathit{Post}\}$$

dove:

- La **precondizione** indica cosa deve valere **prima** che la funzione F sia invocata
- La **postcondizione** indica cosa deve valere **dopo** che la funzione F ha finito terminato la propria esecuzione

Le precondizioni e postcondizioni possono essere definite in modi diversi, come:

- Linguaggi naturali
- Linguaggi per asserzioni
- Linguaggi ad hoc

Soprattutto, tra di questi si riconosce la *FOL*, che può essere usata per questo scopo.

8.6.1 Specifiche

Una specifica deve essere usata come un *"contratto"*, che contiene tutte le informazioni necessarie senza assunzioni a priori. Quando qualche condizione è eliminata dalla precondizione, la specifica diventa insoddisfacente.

Una **specifica formale** è una descrizione matematica di un sistema. Come per le specifiche, esistono più linguaggi di specifica diversi.
Dopo aver specificato i requisiti di un algoritmo (o di un sistema), è necessario verificare la correttezza del medesimo. Utilizzando un modello matematico dell'implementazione costruita, è possibile ottenere la prova di correttezza come una dimostrazione di teorema.

Il problema nella specifica dei sistemi è detto **frame problem**. Esso rappresenta il problema di esprimere un dominio in logica senza dover descrivere esplicitamente le condizioni che non sono modificate da un'azione. Per questo motivo situazioni estremamente semplici possono richiedere formalizzazioni complesse.

8.6.2 Esempi di specifiche

8.6.2.1 Algoritmo di ricerca

Sia F una funzione che implementa la ricerca di un elemento x in un array ordinato a di n elementi.
Allora, *informalmente*:

- *Precondizione*: l'array a è ordinato
- *Postcondizione*: la variabile logica `found` deve essere vera se e solo se l'elemento x esiste nell'array a

Formalizzando le precedenti definizioni in *FOL* si ottiene:

- *Precondizione:* $\forall\, i\, (1 \leq n \leq n-1 \rightarrow a[1] \leq a[i+1])$
- *Postcondizione:* $\texttt{found} \leftrightarrow \exists\, i\, (1 \leq i \leq n \wedge a[i] = x)$

8.6.2.2 Agoritmo di ordinamento

Sia *ORD* un programma che ordina un'array a di n elementi senza ripetizioni.
Allora, *informalmente*:

- *Precondizione*: l'array a non contiene ripetizioni
- *Postcondizione*: l'array ottenuto o è ordinato

Tuttavia questa definizione **non è sufficiente**: bisogna anche indicare che ogni elemento di a deve essere presente in o. Si aggiunge un array b (non utilizzato da *ORD*) con gli stessi elementi di a per poter fare rifermento a quest'ultimo prima che venga modificato.
Formalizzando le precedenti definizioni in *FOL* si ottiene:

- *Precondizione:*

$$\neg\exists\, i, j\, (1 \leq i \leq n \wedge 1 \leq j \leq n \wedge i \neq j \wedge a[i] = a[j]) \wedge \forall\, i\, (1 \leq i \leq n \rightarrow a[i] = b[i])$$

- *Postcondizione:*

$$\forall i\,(i \leq i \leq n \rightarrow a[i] \leq a[i+1]) \wedge$$
$$\forall i\,(1 \leq j \leq n \rightarrow \exists j\,(1 \leq j \leq n) \wedge a[i] = b[j]) \wedge$$
$$\forall j\,(1 \leq i \leq n \rightarrow \exists i\,(1 \leq i \leq n) \wedge a[i] = b[j])$$

8.6.3 Esempi di specifica formale

8.6.3.1 Comportamento di una lampada

Se premo il tasto, la luce si accende entro Δ unità di tempo.
È necessario introdurre dei predicati opportuni:

- $P_B(t)$: istante di pressione del tasto
- $L_{ON}(T)$: istante di accensione della luce

A questo punto si potrebbe **erroneamente** definire la equivalente formula *FOL* della specifica come:

$$\forall t\,(P_B(t) \rightarrow \exists t_1\,((t \leq t_1 \leq t + \Delta) \wedge L_{ON}(t_1)))$$

Tuttavia:

- **Non è specificato** che qualcuno debba premere il pulsante affinché la luce si possa accendere.
- **Non è specificato** cosa succede dopo che la luce si è accesa
- **Non è specificato** cosa succede se si preme il pulsante quando la luce è accesa
- **Non è specificato** cosa succede se si preme il pulsante due volte
- **Non è specificato** se la luce può essere accesa senza premere il tasto

Per poter dare una specifica corretta, si altera leggermente la definizione di funzionamento, supponendo che dopo essersi accesa la lampada rimanga nel suo stato per k unità di tempo per poi spegnersi. Lo schema di funzionamento è riportato nella Figura 30.

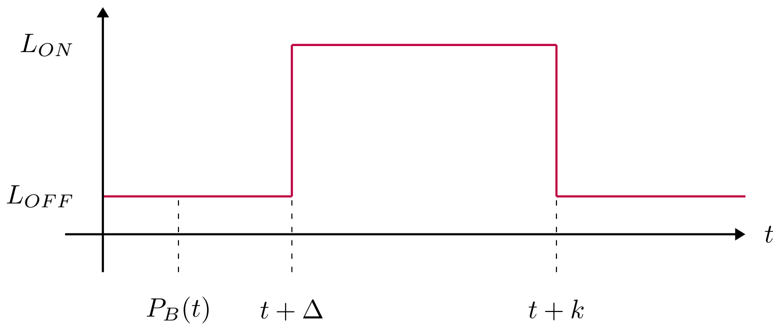


Figura 30: Schema di funzionamento della lampada

La specifica corretta sarà:

$$\forall t\,((P_B(t) \wedge L_{OFF}(t)) \rightarrow \forall t_1\,(t \leq t_1 \leq t + k) \rightarrow L_{ON}(t_1) \wedge L_{OFF}(t + k))$$
$$\wedge$$
$$\forall t_3, t_4\,((L_{OFF}(t_3) \wedge \forall t_5\,(t_3 \leq t_5 \leq t_4) \rightarrow \neg P_B(t_5)) \rightarrow L_{OFF}(t_4))$$

9 Teoria della computabilità

Dopo aver illustrato un numero significativo di modelli atti a descrivere problemi di elaborazione delle informazioni e la loro soluzione e dopo averne studiato le capacità e i limiti, ci dedicheremo ora a studiare quali problemi possano essere affrontati e risolti mediante macchine da calcolo. Prima di poter continuare, è importante riassumere alcuni concetti appresi fino ad ora:

1. Automi, grammatiche e altri formalismi possono essere considerati dispositivi meccanici per risolvere problemi matematici (e quindi i problemi pratici che essi modellano)
2. Alcuni formalismi sono più potenti di altri, ossia sono in grado di riconoscere alcuni linguaggi che altri sono incapaci di riconoscere
3. Nessun formalismo, tra quelli visti fino ad ora, è più potente di una *TM*
   - $\rightarrow$ alcuni hanno tuttavia la stessa medesima potenza delle *TM*
   - $\rightarrow$ questi modelli verranno d'ora in poi chiamati **formalismi massimi**

Una volta costruita una visione di insieme più completa, può venire naturale porsi delle domande del tipo:

- *"I formalismi introdotti sono adeguati per catturare l'essenza di un solutore meccanico?"*
- *"La capacità di un meccanismo di risolvere un problema dipende da come è formalizzato?"*
- *"Esistono formalismi e modelli più potenti delle TM?"*
- *"Una volta che un problema è stato formalizzato adeguatamente, è sempre possibile risolverlo tramite dispositivi meccanici?"*

Per rispondere a queste *(e altre)* domande, è nata una branca dell'informatica detta **teoria della computabilità**.

9.1 Formalizzazione di un problema matematico

I formalismi fino ad ora studiati sono adeguati per tutti i problemi con domini numerabili, ovverosia gli insiemi i cui elementi possono essere messi in corrispondenza biunivoca con gli elementi di $\mathbb{N}$. Il problema originale, quindi, si riduce a quello del calcolo di una funzione $f : \mathbb{N} \rightarrow \mathbb{N}$, esattamente come una traduzione o come un riconoscimento di linguaggio in un determinato alfabeto.

Si consideri, come *esempio*, il problema relativo alla ricerca delle soluzioni di un sistema di equazioni del tipo:

$$\begin{cases} a_1x_1 + a_2x_2 = c_1 \\ b_1x_1 + b_2x_2 = c_2 \end{cases}$$

Esso può essere descritto come il calcolo della funzione f da $\mathbb{Z}^6$ a $\mathbb{Q}^2$ definita come:

$$f(a_1, a_2, b_1, b_2, c_1, c_2) = \langle x_1, x_2 \rangle$$

dove x_1, x_2, se esistono, sono i numeri razionali che soddisfano il sistema.
Si nota poi che è possibile far corrispondere biiettivamente $\mathbb{Z}$ *(numeri interi)* e $\mathbb{Q}$ *(numeri razionali)* con $\mathbb{N}$ *(numeri interi positivi)*, riducendo il problema al calcolo dell'opportuna funzione $f : \mathbb{N} \rightarrow \mathbb{N}$ come proposto prima.
Poiché tutti e 3 gli insiemi sono **enumerabili**, la correlazione esiste e può essere costruita. Un esempio di tale correlazione è mostrata nella Tabella 3.

$\mathbb{Z}$	0	1	−1	2	−2	3	−3	...
$\mathbb{N}$	0	1	2	3	4	5	6	...

Tabella 3: Correlazione tra $\mathbb{Z}$ e $\mathbb{N}$

9.2 Riconoscimento e traduzione

Il **riconoscimento** e la **traduzione** sono due formulazioni di un problema che possono essere ridotte l'una nell'altra.
Infatti, il problema di stabilire se, per una data stringa x e un dato linguaggio L, $x \in L$ si può anche impostare come la traduzione τ_L, dove

$$\tau_L(x) = \begin{cases} 1 & \text{se } x \in L \\ 0 & \text{altrimenti} \end{cases}$$

Viceversa, data una traduzione $\tau : V_1^* \to V_2^*$ si può definire il linguaggio:

$$L_\tau = \{z \mid z = x\$y,\ x \in V_1^*,\ y \in V_2^*,\ \$ \notin (V_1 \cup V_2),\ y = \tau(x)\}$$

cioè il linguaggio delle stringhe formate da una stringa su V_1^*, seguita dalla sua traduzione separata dal carattere speciale \$.
Un dispositivo che riconosce L_τ può essere usato come trasduttore che calcola τ: per ogni x, è possibile enumerare **tutte le infinite** $y \in V_2^*$ e verificare se $x\$y \in L_\tau$. Se ciò avviene, si può concludere che la stringa y è la traduzione rispetto a τ della stringa x, cioè $\tau(x) = y$. Tuttavia questo processo termina se e solo se $\tau(x)$ è definita, perché se così non fosse si procederebbe a enumerare le $y \in V_2^*$ senza mai trovare una stringa che appartenga a L_τ.

Infine, è importante ricordare che:

- Tutti i formalismi esaminati sono discreti, perché domini matematici numerabili definiti in modo finito
- La classe dei problemi che possono essere risolti da una *TM* è indipendente dall'alfabeto scelto *(a patto sia composto da almeno due simboli)*

9.3 *TM* e linguaggi di programmazione

Data una *TM M*, è possibile scrivere un programma che la simuli in un qualsiasi linguaggio. Analogamente, dato un qualunque programma, in un qualsiasi linguaggio di programmazione, è possibile costruire una *TM* che calcola la stessa funzione eseguita dal primo.
Quindi, è immediato dedurre che **le *TM* hanno lo stesso potere espressivo dei linguaggi di programmazione di alto livello** e di conseguenza sarà possibile sfruttare una *TM* per eseguire un algoritmo.

9.3.1 Algoritmi

Il concetto di **algoritmo** è uno dei concetti centrali dell'informatica. *Intuitivamente*, esso indica la procedura di risoluzione di un problema mediante un dispositivo automatico di calcolo, come un calcolatore elettronico. Alternativamente può indicare un metodo per descrivere una serie astratta di comandi elementari, indipendenti dal linguaggio comprensibile a un calcolatore, per risolvere un dato problema.
Una definizione formale di algoritmo sarebbe di difficile formulazione, quindi ci si limita a descriverne delle proprietà in modo informale:

A) Un algoritmo contiene una sequenza **finita** di istruzioni
B) Ogni istruzione deve essere **immediatamente eseguibile** tramite qualche procedimento meccanico
C) Il processore è dotato di **dispositivi di memoria** in cui possono essere immagazzinati i risultati intermedi
D) La computazione è un processo **discreto**
   - → l'informazione è codificata in forma digitale
   - → la computazione procede attraverso passi discreti

E) Gli algoritmi sono eseguiti in modo **deterministico**
F) **Non esiste un limite** alla quantità di **dati** di ingresso e uscita
G) **Non esiste un limite** alla quantità di **memoria** richiesta per effettuare i calcoli
H) **Non esiste limite** al numero di passi discreti richiesti per effettuare un calcolo
   - → si possono avere computazioni infinite

Le ipotesi dalla (A) alla (D) si applicano agli algoritmi nei calcolatori digitali, mentre quelle dalla (E) alla (H) si applicano in senso generale. Il punto (D) relativo al non determinismo è una semplificazione *(analogamente al punto (C) riguardante la memoria)* rispetto al formalismo delle *TM*.

9.4 Tesi di Church

La tesi di Church è intrinsecamente non dimostrabile perché è basata solo sull'esperienza precedente e sull'evidenza intuitiva. Per questo motivo non è considerato un teorema.
È divisa in due parti.

9.4.1 Prima parte

La prima parte della **tesi di Church** afferma che:

> *"Non esiste alcun formalismo, per modellare una determinata computazione meccanica, che sia più potente delle TM e dei formalismi a esse equivalenti."*

Con la locuzione *potenza di calcolo* si intende la classe di problemi risolvibili tramite un determinato formalismo e non lo *"sforzo"* che la macchina impiega nel calcolo della soluzione.
Questa tesi è da verificare ogni qualvolta viene inventato un modello più potente di calcolo *(come i computer quantistici)* ma tutt'oggi non è ancora stata provata la sua falsità.

9.4.2 Seconda parte

La seconda parte della **tesi di Church** afferma che:

> *"Ogni algoritmo per la soluzione automatica di un problema può essere codificato in termini di una TM (o di un formalismo equivalente)."*

Ciò implica che nessun algoritmo può risolvere problemi che non possano già essere risolti da una *TM*: essa infatti rappresenta il più potente calcolatore che sia possibile costruire.

Per la definizione di algoritmo, si faccia riferimento alla Sezione 9.3.1.

9.5 Enumerazione delle *TM* e *TM* universali

Fino ad ora le *TM* sono state analizzate in quanto dispositivi in grado di risolvere **un particolare** problema. In particolare, è stato definito il problema definito dalla funzione di transizione δ, facente parte della struttura della macchina stessa.
Per questo motivo, le *TM* possono essere considerate come macchine calcolatori astratti, specializzati e non programmabili. Una volta caricato il programma nella memoria *(l'automa di controllo)*, la macchina potrà eseguire solo quella particolare sequenza di istruzioni.
A questo punto può venire naturale chiedersi:

> *"È possibile modellare un calcolatore programmabile con una TM? Le TM sono in grado di calcolare tutte le funzioni da* $\mathbb{N}$ *a* $\mathbb{N}$*?"*

Le prossime Sezioni si dedicheranno alla ricerca di una soluzione a entrambe le domande.

9.5.1 Enumerazione algoritmica

Dato un qualsiasi insieme S, esso può essere **numerato algoritmicamente** se è possibile stabilire una relazione biettiva fra lo stesso S e l'insieme dei numeri naturali $\mathbb{N}$, tramite un algoritmo o una *TM (grazie alla tesi di Church)*. È possibile dimostrare formalmente che le *TM* possono essere enumerate **algoritmicamente**.

Per chiarire le idee sulla enumerazione, si consideri il seguente *esempio*:
Sia L l'insieme $\{a, b\}^*$ delle stringhe sull'alfabeto $\{a, b\}$. Tale insieme è algoritmicamente enumerabile: per esempio, si possono ordinare le stringhe per lunghezza crescente e, a parità di lunghezza, assegnare un ordine tra gli elementi dell'alfabeto su cui sono definite. Il risultato è mostrato nella Tabella 4.

ϵ	a	b	aa	ab	ba	bb	aaa	aab	...
0	1	2	3	4	5	6	7	8	...

Tabella 4: Enumerazione algoritmica su $\{a, b\}^*$

9.5.2 Algoritmo di enumerazione

Per semplicità, si fissi un alfabeto A e si consideri l'insieme $\{TM_A\}$ delle TM a nastro singolo e senza stati finali, aventi A come insieme di simboli. Le TM in $\{TM_A\}$ accettano una stringa se e solo se arrivano in uno stato di `halt`. In caso contrario, compierebbero infinite mosse, entrando in un loop da cui non possono più uscire. Queste semplificazioni non comportano perdita di generalità perché queste operazioni sono possibili su ciascuna TM vista fino ad ora.
L'obiettivo di questa dimostrazione sarà spiegare come $\{TM_A\}$ possa essere numerato, ossia come sia possibile stabilire una biiezione $\mathscr{E} : \mathbb{N} \leftrightarrow \{TM_A\}$.
Si osservi in primo luogo che $\forall\, k \in \mathbb{N}$ esiste un numero finito di TM, con k stati e con alfabeto A. Infatti, la funzione di transizione δ sarà definita su dominio e codominio finiti. In senso più generale, data una funzione $f : D \to R$, con D, R finiti, vi sono esattamente $|R|^{|D|}$ funzioni totali f diverse.
Seguendo lo stesso ragionamento per le TM, poiché la loro funzione di transizione di transizione parziale δ è definita come

$$\delta : Q \times A \to Q \times A \times \{R, L, S\}^{k+1} \cup \{\perp\}$$

esisteranno esattamente:

$$(1 + 3 \times |Q| \cdot |A|)^{|Q|\cdot|A|} = (1 + 3 \times h \cdot k)^{h\cdot k}$$

TM, aventi $|A| = h, |Q| = k$ *(cardinalità dell'alfabeto e degli stati)*. Il termine unitario è dovuto alla presenza di $\perp$ nel codominio.
Per poter enumerare le TM dell'insieme sarà però prima necessario ordinarle in base all'*ordine lessicografico*. A tal fine è sufficiente imporre un ordinamento all'interno degli insiemi Q, A ed $\{R, L, S\}$, ottenendo un insieme ordinato:

$$\mathscr{E} : \{TM_A\} \mapsto \mathbb{N}$$

L'enumerazione $\mathscr{E}$ è algoritmica: è possibile implementare la costruzione con un algoritmo *(sia esso implementato in un linguaggio di programmazione o a sua volta una TM)* che emetta in uscita tutti gli elementi di $\{TM_A\}$ nell'ordine lessicografico appena imposto. Analogamente, sarà possibile costruire un algoritmo che faccia l'operazione inversa, cioè che partendo da una TM ne estragga il corrispondente $k \in \mathbb{N}$.

Infine l'enumerazione calcolabile da una TM prende il nome di **Gödelizzazione** e il numero naturale associato verrà chiamato **numero di Gödel** dell'oggetto. Nella precedente enumerazione, ogni TM è biiettivamente identificata dal suo numero di Gödel.

9.5.3 Riassunto dell'enumerazione

Per riassumere i concetti visti nell Sezione precedente, si ottiene un enunciato fondamentale:

1. Per ogni alfabeto A, il corrispondente l'insieme di TM $\{TM_A\}$ può essere numerato algoritmicamente, perché può essere sempre stabilita una biiezione $\mathscr{E} : \{TM_A\} \leftrightarrow \mathbb{N}$
2. Tutte le funzioni **algoritmicamente computabili** si possono enumerare algoritmicamente
3. Tutti i linguaggi **algoritmicamente riconoscibili** si possono enumerare algoritmicamente

Si noti che, sebbene $\mathscr{E}$ sia una funzione biettiva, le enumerazioni delle funzioni computabili e dei linguaggi riconoscibili non sono sempre tali, perché si verifica che molte TM sono in grado di calcolare la medesima funzione.

Per comodità, nel resto del corso si farà riferimento a TM che si comportano come dispositivi che calcolano funzioni da $\mathbb{N}$ a $\mathbb{N}$. Per ogni funzione $f : D \to R$, con D, N diversi da $\mathbb{N}$, si presupporrà implicitamente la codifica algoritmica di D e R in funzione di $\mathbb{N}$. Analogamente si farà riferimento alla y-esima TM generata dalla enumerazione $\mathscr{E}$ con M_y (quindi $M_y = \mathscr{E}(y)$).

9.5.4 Macchina di Turing universale - *UTM*

Fino a ora le *TM* sono state considerate come calcolatori astratti, specializzati e non programmabili. In questa sezione verrà introdotta la **macchina di Turing universale** (o *UTM*, dall'Inglese *Universal Turing Machine*), ossia una *TM* in grado di modellare dispositivi di risoluzione dei problemi, in cui il problema da risolvere non viene codificato nella struttura del dispositivo ma viene fornito in ingresso insieme ai dati su cui operare.

La *UTM* può essere definita come una *TM* che calcola la funzione $g(y, x) = f_y(x)$, dove:

- $y \in \mathbb{N}$ indica l'indice della **funzione** f_y calcolata dalla *TM* M_y
- $x \in \mathbb{N}$ rappresenta l'**ingresso** su cui opera M_y

Apparentemente la *UTM* così definita non sembrerebbe appartenere all'insieme $\{TM_A\}$ in quanto le funzioni a esso associate sono in una variabile, mentre g è definita a due variabili. Questo problema si dimostra facilmente risolvibile in quanto esiste **sempre** una biezione $\mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$. Un esempio di tale biiezione è dato dalla funzione:

$$d(x, y) = \frac{(x + y)(x + y + 1)}{2} + x$$

che permette di associate una coppia $x, y \in \mathbb{N}$ a un numero $k \in \mathbb{N}$ in modo biunivoco. Allo stesso modo sarà possibile costruire la funzione inversa d^{-1}.
La Figura 31 illustra il funzionamento della biezione.

Poiché si dimostra che d e d^{-1} sono computabili, la *TM* sarà **sempre** un grado di calcolarla.

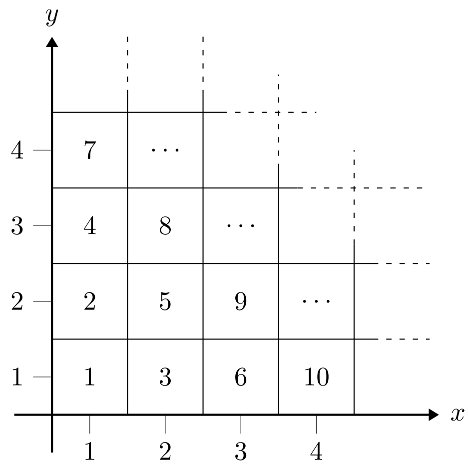


Figura 31: Illustrazione dell'associazione biunivoca

Si osservi che g è una funzione *TM*-computabile, ossia esiste un $i \in \mathbb{N}$ tale che $f_i = g$ ed è possibile calcolarla mediante i passi seguenti:

1. Si sceglie un alfabeto finito A per codificare ogni informazione richiesta per la computazione.

   → qualunque A con $|A| \geq 2$ è adatto allo scopo

2. Si traduce la rappresentazione di n nella opportuna coppia $\langle x, y \rangle$ tramite la funzione biettiva scelta
3. Si traduce il numero y in un'opportuna codifica della *TM* y-esima nella enumerazione $\mathscr{E}$, M_y

   → La traduzione viene memorizzata sul nastro della *UTM*

4. Si simula la computazione di M_y su x

   → la *UTM* lascia sul nastro $f_y(x)$ se e solo se M_y termina la sua computazione su x

Quindi viene dimostrato che la *UTM* è in grado di calcolare $g(x, y) = f_y(x)\ \forall x, y$, cioè il comportamento di ogni altra *TM*. La *UTM* è quindi in grado di calcolare anche il comportamento di **sé stessa**.

Con questo enunciato viene risposta la prima domanda posta nella Sezione 9.5.1.

9.6 Problemi algoritmicamente irrisolvibili

Prima di potersi occupare della risolvibilità dei problemi *(e della loro relativa irrisolvibilità)*, è necessario studiare il numero delle stesse funzioni computabili. Come concluso nella Sezione 9.5.3, infatti, tutte le funzioni computabili $f_y : \mathbb{N} \to \mathbb{N}$ sono enumerabili. Questo significa che la cardinalità del loro insieme è $\aleph_0$.
Analogamente, la cardinalità della classe $\mathscr{F}$ delle funzioni su $\mathbb{N}$, cioè $\mathscr{F} = \{f \mid f : \mathbb{N} \to \mathbb{N}\}$ è:

$$|\mathscr{F}| = |\mathbb{N}| \times |\mathbb{N}| = \aleph_0 \times \aleph_0 = 2^{\aleph_0}$$

Questo valore è noto come **cardinalità del continuo** perché pari alla cardinalità dell'insieme $\mathbb{R}$ dei numeri reali.
Quindi rapportando le due cardinalità *(rispettivamente $\aleph_0$ e $2^{\aleph_0}$ per le funzioni risolvibili e per tutte le funzioni)* si conclude che la prima è minore della seconda ($\aleph_0 < 2^{\aleph_0}$), e quindi **non tutte le tutte le funzioni sono risolvibili**.

Così facendo è stata data risposta anche alla seconda domanda posta nella Sezione 9.5.1.

9.6.1 Problemi definibili

Intuitivamente, le conclusioni raggiunte nella sezione precedente non forniscono grandi limitazioni. Infatti, nonostante la classe di tutte le funzioni $f : \mathbb{N} \to \mathbb{N}$ sia di cardinalità superiore alla classe delle funzioni computabili, le sole funzioni *interessanti* da calcolare sono quelle possono essere *definite*.
Per poter quindi definire un problema, è necessario usare una stringa di un qualche linguaggio, sia esso di tipo naturale o formale. Per esempio, alcune descrizioni di problemi possono essere:

- *il numero che moltiplicato per se stesso è uguale a y*
- $f(x) = x^4 - \pi \cdot x^2$
- $f(x) = \displaystyle\int_0^x z \cdot sin(z)dz$
- $f(x) = \displaystyle\lim_{x\to\infty} \frac{e^x}{x^2 + x}$

Ogni linguaggio è un sottoinsieme di A^* *(il linguaggio delle UTM prese in esame)* e quindi è numerabile. Così si prova che l'insieme dei problemi che si possono definire è esso stesso numerabile e:

$$\{\text{problemi risolvibili}\} \subseteq \{\text{problemi definibili}\}$$

ma, *purtroppo*, i due insiemi non coincidono **e parte dei problemi definibili non sarà risolvibile**. La relazione tra i due insiemi sarà quindi:

$$\{\text{problemi risolvibili}\} \subset \{\text{problemi definibili}\}$$

In pratica, la classe dei problemi non definibili è rappresentata da tutti quei problemi che, per carenze di tipo linguistico *(sia esso naturale, matematico o altro)* non è possibile esprimere. L'assenza di una soluzione per essi non è rilevante in quanto non è neanche possibile porsi il problema di cercarla.

9.7 Problema della terminazione - *halting problem*

Una delle proprietà che si vuole garantire durante la stesura di un programma, qualunque sia il linguaggio, è la capacità del programma stesso di *terminare* correttamente, ossia che *"non vada in loop"*.
Sarebbe estremamente utile poter determinare a priori, prima di eseguire una funzione con un determinato input, se essa porti a risultati in tempo finito o cada in loop infiniti che la portino a non terminare mai.
Questo problema, detto **problema della terminazione** (o in Inglese *halting problem*), è *descrivibile*, ma purtroppo *non decidibile*.

Analizzando il problema in modo *formale*, si indicherà con M_y la *TM* y-esima secondo la numerazione $\mathscr{E}$ e con f_y la funzione calcolata da M_y. Allora, è verificato che nessuna *TM* può calcolare la funzione totale $g : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ definita nel seguente modo:

$$g(x, y) = \begin{cases} 1 \text{ se } f_y(x) \neq \bot \\ 0 \text{ altrimenti} \end{cases}$$

e che quindi nessuna *TM* può decidere se, per una generica *TM* M e un generico ingresso x, M si arresta in uno stato finale *(denotato come* $f_y(x) = \perp$*)* con l'ingresso x.
La dimostrazione avverrà tramite diagonalizzazione, dopo l'introduzione della tecnica stessa nella prossima Sezione.

9.7.1 Dimostrazioni per diagonalizzazione

Il ragionamento originale per diagonalizzazione fu usato da *Georg Cantor* nel 1891 per dimostrare che $\mathbb{R}$ ha una cardinalità maggiore di $\mathbb{N}$. È un metodo spesso usato per mostrare l'indecidibilità di alcuni problemi famosi e si riferisce a uno schema comune: si cerca una contraddizione quando esiste un insieme di funzioni $A \to B$ indicizzato su A. Se f_k è una funzione indicizzata da $k \in A$, un ragionamento per diagonalizzazione è dato dall'analisi degli elementi sulla diagonale della tabella ottenuta enumerando le f_k, come nella Tabella 5 (in cui la diagonale è evidenziata in **bianco su grigio**).

n	1	2	3	4	...
f_1	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	...
f_2	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	...
f_3	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$	...
f_4	$f_4(1)$	$f_4(2)$	$f_4(3)$	$f_4(4)$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Tabella 5: Dimostrazione per diagonalizzazione

Allora si definisce l'elemento diagonale come una funzione $d : A \to B$ che differisce dalla diagonale in ogni valore. Poiché d è una funzione $A \to B$, essa appare nella lista.
Visto che essa differisce anche da ogni elemento della lista sulla diagonale, la dimostrazione è avvenuta **per assurdo**.

9.7.2 Dimostrazione della cardinalità del continuo

La dimostrazione avviene per assurdo ed è strutturata in passi:

1. Si supponga, per assurdo, che l'intervallo $[0, 1]$ sia numerabile
2. Ciò implica che gli elementi di $[0, 1]$ possano essere messi in corrispondenza biunivoca con i numeri naturali, tramite successione di numeri reali $\{n_1, n_2, \ldots\}$ che esaurisce tutti i numeri reali compresi tra 0 e 1
3. È possibile rappresentare ciascun numero della successione in forma decimale e visualizzare la successione di numeri reali come una matrice infinita che avrà circa questo aspetto:

$$\begin{array}{lccccccccc}
n_1 = 0, & 3 & 1 & 3 & 5 & 6 & 3 & 1 & \ldots \\
n_2 = 0, & 6 & 8 & 1 & 3 & 9 & 8 & 7 & \ldots \\
n_3 = 0, & 1 & 6 & 6 & 5 & 7 & 1 & 1 & \ldots \\
n_4 = 0, & 1 & 3 & 6 & 6 & 5 & 5 & 8 & \ldots \\
n_5 = 0, & 8 & 5 & 2 & 4 & 1 & 8 & 4 & \ldots \\
n_6 = 0, & 4 & 1 & 2 & 4 & 2 & 5 & 7 & \ldots \\
n_7 = 0, & 7 & 8 & 8 & 4 & 6 & 6 & 3 & \ldots
\end{array}$$

4. Si osservino ora le cifre lungo la diagonale della matrice, cioè sulla successione il cui k-esimo elemento è la k-esima cifra decimale di r_k come mostra la figura:

$$
\begin{array}{lccccccccc}
n_1 = 0, & \boxed{3} & 1 & 3 & 5 & 6 & 3 & 1 & \dots \\
n_2 = 0, & 6 & \boxed{8} & 1 & 3 & 9 & 8 & 7 & \dots \\
n_3 = 0, & 1 & 6 & \boxed{6} & 5 & 7 & 1 & 1 & \dots \\
n_4 = 0, & 1 & 3 & 6 & \boxed{6} & 5 & 5 & 8 & \dots \\
n_5 = 0, & 8 & 5 & 2 & 4 & \boxed{1} & 8 & 4 & \dots \\
n_6 = 0, & 4 & 1 & 2 & 4 & 2 & \boxed{5} & 7 & \dots \\
n_7 = 0, & 7 & 8 & 8 & 4 & 6 & 6 & \boxed{3} & \dots
\end{array}
$$

Questa successione di cifre sulla diagonale, vista come un'espansione decimale, definisce un numero reale (che in questo caso può corrispondere a $0.3866153\dots$).
5. Si consideri ora un nuovo numero reale x che abbia tutte le cifre differenti dalla sequenza sulla diagonale, definendolo come:
   - se la k-esima cifra decimale di n_k è 5, allora la k-esima cifra di x è 4
   - se la k-esima cifra decimale di n_k non è 5, allora la k-esima cifra di x è 5
6. All inizio della dimostrazione si era supposto che la lista $\{n_1, n_2, \dots\}$ enumerasse tutti i numeri reali compresi tra 0 e 1, quindi dovrebbe essere verificato che $n_k = x_k$ per un qualche k e poiché x non ha dei 9 tra le cifre decimali, la sua rappresentazione è unica e sarà presente nella riga k
7. A questo punto emerge la contraddizione: sia a la n-esima cifra decimale di $n_k = x$. Essa può essere 4 o 5: per la definizione, a potrà essere 4 se e solo se è uguale a 5, e 5 negli altri casi. Questo è impossibile e ne segue che l'ipotesi di partenza è falsa.

9.7.3 Dimostrazione del problema di terminazione

Come già annunciato, la dimostrazione avviene tramite tecnica diagonale. Si assuma, per assurdo, che

$$g(y, x) = \begin{cases} 1 \text{ se } f_y(x) \neq \perp & \textit{termina} \\ 0 \text{ altrimenti} & \textit{non termina} \end{cases}$$

```
int g(y, x) {
  if (halts(f_y(x)))
    return 1;

  return 0;
}
```

(descritta sia matematicamente che in pseudocodice) sia computabile.
Partendo da g si definisce ora una funzione h tale che:

$$h(x) = \begin{cases} 1 \text{ se } g(x, x) = 0 & f_x(x) \ \textit{termina} \\ \perp \text{ altrimenti} & f_x(x) \ \textit{non termina} \end{cases}$$

```
int h(x) {
  if (g(x, x) == 0)
    return 1;

  while (1) {};
}
```

dove $h(x) = 1 \Rightarrow g(x, x) = 0,\ f_x(x) = \perp$. L'indefinizione non implica che la funzione h non sia calcolabile, ma significa che la funzione calcolata dalla corrispondente *TM* non termina.
Usando h, ci si trova quindi nella *diagonale* della tabella delle $f_y(x)$.

Se g fosse computabile, allora anche h lo sarebbe. Analogamente, se h fosse computabile, allora si verificherebbe che $h(i) = f_i(i)$ per una qualche i.

Si calcoli ora $h(i)$ ricordando che $f_x(i) = h(i)$ e applicando la definizione appena data:

$$h(i) = \begin{cases} 1 \Rightarrow f_i(i) \neq \perp \Rightarrow g(i,i) = 0 \Rightarrow f_i(i) = \perp & \textbf{assurdo} \\ \perp \Rightarrow f_i(i) = \perp \Rightarrow g(i,i) = \perp \Rightarrow f_i(i) \neq \perp & \textbf{assurdo} \end{cases}$$

Quindi entrambi i casi sono inammissibili e la dimostrazione è terminata.

9.7.4 Lemma 1

Sia h' la funzione definita come:

$$h'(x) = \begin{cases} 1 \text{ se } f_x(x) \neq \perp \\ 0 \text{ altrimenti} \end{cases}$$

Allora h' **non è computabile.**
Si noti che $h'(x)$ *"si muove"* ancora sulla diagonale (coincide con $g(x,x)$) ed è quindi una semplificazione di $f(x)$. È un problema analogo al problema di terminazione, di cui è appunto un lemma, ma non dipende da esso.

9.7.4.1 Dimostrazione del lemma 1

La dimostrazione segue la traccia della dimostrazione del teorema di terminazione, nel avvenuta nella Sezione 9.7.3.

Si definisca inizialmente la funzione

$$h(x) = \begin{cases} 1 \text{ se } k(x) = 0 \\ \perp \text{ altrimenti} \end{cases}$$

Si osservi ora che h è computabile se k è computabile. Una volta ottenuta una *TM* M in grado di calcolare k, sono sufficienti solo semplici modifiche per adattarla al calcolo di h.
Considerando ora x_0 il numero di Gödel di una terminata *TM* M_{x_0} che calcola h, ossia $h = f_{x_0}$ e quindi h è la funzione calcolata dalla x_0-esima *TM*.
Si verifica facilmente, tuttavia, che $h(x_0) = 1$ implica $f_{x_0}(x_0) = 1$, mentre $h(x_0) = \perp$ implica $h(x_0) = 1$.
Quindi $f_{x_0}(x_0) = h(x_0) = 1$, che è una contraddizione.

9.7.5 Lemma 2

La funzione $h'(x)$ è un caso particolare della funzione $g(y,x)$ perché:

- $h'(x) = g(x,y)$
- g non è computabile, perché $g(y,x)$ è essa stessa non computabile

Se un problema non è risolvibile, allora un suo caso speciale può essere risolvibile. Contrariamente, la generalizzazione di un problema non risolvibile è necessariamente non risolvibile.
Infine, se un problema è risolvibile, allora una sua generalizzazione potrebbe non essere risolvibile, mentre qualunque sua specializzazione è certamente risolvibile.

9.7.6 Lemma 3

Sia $k(y)$ la funzione definita come:

$$k(y) = \begin{cases} 1 \text{ se } \forall x \in \mathbb{N},\ f_y(x) \neq \perp \\ 0 \text{ altrimenti} \end{cases}$$

Allora $k(y)$ **non è computabile.**
Questa è una quantificazione su tutti i possibili dati in ingresso. In qualche caso potrebbe essere possibile stabilire se $f_y(x) \neq \perp$ per un qualche x specifico, ma non sarà mai possibile per ogni $x \in \mathbb{N}$ *(perché è un insieme infinito)*.
Se ciò, *per assurdo*, fosse possibile, determinare se f_y è una funzione totale sarebbe impossibile.

9.7.6.1 Dimostrazione lemma 3

La dimostrazione segue la traccia della dimostrazione del teorema di terminazione, nel avvenuta nella Sezione 9.7.3.
Per ipotesi, sia $k(y)$ la funzione definita come:

$$k(y) = \begin{cases} 1 \text{ se } \forall\, x \in \mathbb{N},\ f_y(x) \neq \perp \\ 0 \text{ altrimenti} \end{cases}$$

Sempre per ipotesi, $k(y)$ è **totale** e **computabile**.
Allora sarà possibile definire $g(x) = w$, con w pari al numero di Gödel della x-esima *TM* (definita nell'insieme $\mathscr{E}$) che calcola una funzione totale.
Se k fosse veramente computabile e totale, allora lo sarebbe anche g. Infatti:

1. Sarebbe possibile calcolare $k(0),\ k(1),\ \ldots$
2. Sia w_0 il primo valore tale per cui $k(w_0) = 1$
3. Sia $g(0) = w_0$

La procedura è algoritmica, e siccome esistono infinite funzioni totali, $g(x)$ è sicuramente definita per ogni $x \in \mathbb{N}$, quindi è **totale** e **strettamente monotona**, perché $w_{x+1} > w_x$.
Di conseguenza, anche g^{-1} è strettamente monotona ma non totale, perché g^{-1} è definita solo se w è il numero di Gödel di una funzione totale.
Allora si definisce la funzione $h(x)$ come

$$h(x) = f_{g(x)} + 1 = f_w(x) + 1$$

e poiché $f_w(x)$ è computabile e totale, anche $h(x)$ lo sarà.
Per qualche valore di w_i, infine, si verifica $h(x) = f_{w_i}(x)$ e poiché h è totale, $g^{-1}(w_i) = x_i \Rightarrow g^{-1}(w_i) \neq \perp$.
Confrontando gli ultimi due valori ottenuti, si giunge alla conclusione che:

$$\begin{aligned} h(x_i) &= f_{g(x_i)}(x_i) + 1 = f_{w_i}(x_1) + 1 \\ h(x) = f_{w_i}(x) &\Rightarrow h(x_i) = f_{w_i}(x_1) \end{aligned}$$

Cadendo quindi in una evidente contraddizione.

9.8 Osservazioni sulla risolvibilità

È importante notare che sapere che un problema è risolvibile non implica che il metodo di risoluzione sia noto.
In matematica è spesso necessario dare dimostrazioni non costruttive: si mostra che un oggetto matematico esiste, ma non lo si calcola.
In questo caso:

- Un problema è **risolvibile** se esiste una *TM* che lo risolve
- Per alcuni problemi si può concludere che esiste una *TM* che li risolve, **senza tuttavia poterla costruire**

9.9 Problema di decisione

Un **problema di decisione** è una domanda che ha due possibili risposte: `SI` o `NO`. La domanda *(e la consecutiva risposta)* sono relative a un qualche particolare ingresso.

Esempi di problemi di decisione:

- Dato un grafo G e un insieme di vertici K, quest'ultima è una cricca *(clique)*?
- Dato un grafo G e un insieme di lati M, quest'ultimo è un albero ricoprente *(spanning tree)*?
- Dato un insieme di assiomi, un insieme di regole e una formula, è possibile dimostrare quest'ultima a partire dai primi due? *problema di incompletezza*

9.9.1 Decidibilità

Un problema si dice **decidibile** se esiste un algoritmo *(procedura di decisione)* tale che:

- È un procedimento **automatico** *(da definizione di algoritmo)*
- Se il problema è **vero** per un determinato ingresso, allora ritorna `SI`
- Se il problema è **falso** per un determinato ingresso, allora ritorna `NO`

9.9.2 Semidecidibilità

Un problema si dice **semidecidibile** se esiste un algoritmo *(procedura di semidecisione)* tale che:

- È un procedimento **automatico** *(da definizione di algoritmo)*
- Se il problema è **vero** per un determinato ingresso, allora ritorna `SI`
- Negli altri casi, **potrebbe non terminare mai**

Il problema della terminazione (Sezione 9.7) è **indecidibile** e **semidecidibile** allo stesso tempo.

9.9.2.1 Problema della terminazione e semidecidibilità

È dimostrato che il problema della terminazione (Sezione 9.7) è **semidecidibile**. Si consideri infatti l'analoga formulazione $\exists\, z \mid f_x(z) \neq \bot$.
Schema di dimostrazione:

- Se si calcola $f_x(0)$ e ne risulta che $f_x(0) \neq \bot$ allora la risposta è `SI`
- Se la computazione non termina, come è possibile accorgersene?
- Dimostrazione con metodo diagonale:
  1. Si simuli 1 passo di esecuzione di $f_x(0)$. Se termina, allora la risposta è `SI`
  2. Altrimenti, si simuli un passo di computazione di $f_x(1)$. Se termina, allora la risposta è `SI`
  3. Ancora una volta si simuli 2 passi di $f_x(0)$, 1 passo di $f_x(2)$, 2 passi di $f_x(1)$, 3 di $f_x(0)$ e così via, secondo lo schema in Figura 32 *(già vista nella Sezione relativa alla Macchina di Turing universale - UTM)*

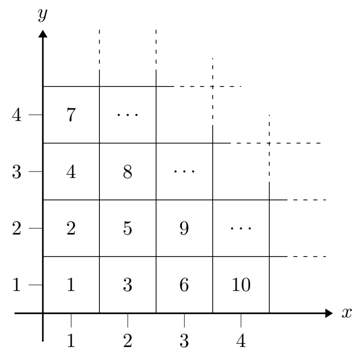


Figura 32: Illustrazione dell'associazione biunivoca

9.9.2.2 Osservazioni

Grande parte dei problemi indecidibili è **anche** semidecidibile. Un tipico esempio di questa affermazione è data dagli errori a *runtime* nei programmi.

Si nota che in questi casi il problema semidecidibile è dato dalla ricerca della presenza dell'errore, non nella dimostrazione della sua assenza. Ciò porta a una importante implicazione sulla verifica basata sul testing. Secondo Dijkstra:

> *"Il testing può dimostrare la presenza di errori e non la loro assenza."*

Molti problemi, seppur ben definiti, non possono essere risolti mediante procedimenti algoritmici. Il risultato sarà trovato tramite tecniche differenti.
Allo stesso modo esistono problemi la cui soluzione algoritmica è nota, senza che il procedimento stesso sia noto.

9.10 Insiemi ricorsivi

In questa sezione ci si dedicherà a studiare i sottoinsiemi di $\mathbb{N}$, che verranno indicati d'ora in poi con la lettera S. Dati $x \in \mathbb{N}, S \subseteq \mathbb{N}$, si vuole studiare l'appartenenza dell'elemento x allo stesso S.
Questa formalizzazione del problema è del tutto generale perché applicabile a qualsiasi insieme che si possa mettere in corrispondenza biunivoca ed effettiva con $\mathbb{N}$, ossia per tutti gli elementi numerabili.

9.10.1 Funzione caratteristica di un insieme

Dato un insieme S, la sua **funzione caratteristica** $c_s : \mathbb{N} \to \{0, 1\}$ è definita come:

$$c_s = \begin{cases} 1 \text{ se } x \in S \\ 0 \text{ altrimenti} \end{cases}$$

L'appartenenza di un elemento a un insieme e la ricorsività dello stesso è un problema la cui risolvibilità dipende dalla computabilità della sua funzione caratteristica.

9.10.2 Definizione di insieme ricorsivo

Un insieme S viene detto **ricorsivo** (oppure *decidibile*) se e solo se la sua funzione caratteristica c_s è **computabile**.
Si ricordi che, per definizione, c_s è **totale** per ogni insieme S. Infatti, dato un qualsiasi elemento $x \in \mathbb{N}$, esso appartiene (se $c_1 = 1$) o non appartiene (se $c_s = 0$) a S.

9.11 Insieme ricorsivamente enumerabile

Un insieme viene detto **ricorsivamente enumerabile** (oppure *semidecidibile*) se e solo se è l'**insieme vuoto** oppure è l'**immagine di una funzione** totale e computabile g_s, cioè:

$$S = I_{g_s} = \{x \,|\, \exists\, y,\ y \in \mathbb{N} \wedge x = g_s(y)\}$$

Gli insiemi ricorsivamente enumerabili devono il loro nome al fatto che il problema dell'appartenenza può essere risolto **meccanicamente** *(oppure algoritmicamente)*. Un calcolatore meccanico in grado di implementare la funzione caratteristica di un insieme fornirà necessariamente una risposta alla domanda $x \in S \; \forall\, x$. Gli insiemi ricorsivamente enumerabili sono una definizione *più debole* degli insiemi *ricorsivi*.

Il termine *"semidecidibile"* puo essere spiegato intuitivamente come:

> *"se si suppone che $x \in S$, allora enumerando gli elementi di S anche x verrà trovato e sarà possibile rispondere alla domanda. Ma se $x \notin S$?"*

9.12 Teorema $\frac{1}{2} + \frac{1}{2} = 1$

Sia S un insieme. Allora:

A) Se S è **ricorsivo**, è anche **ricorsivamente enumerabile**
B) S è **ricorsivo** se e solo se sia S che $\overline{S} = \mathbb{N} - S$ sono **ricorsivamente enumerabili**

Corollario: la classe di insiemi decidibili *(linguaggi, problemi, . . .)* è chiusa rispetto al complemento.

9.12.1 Dimostazione punto (A)

Sia c_s la funzione caratteristica di S. Se S è vuoto, allora è ricorsivamente enumerabile per definizione. Altrimenti, esiste almeno un elemento $k \in S$.
Sia $g_s(x)$ la funzione definita come:

$$g_s(x) = \begin{cases} x \text{ se } c_s(x) = 1 \\ k \text{ altrimenti} \end{cases}$$

Allora $g_s(x)$ è totale, computabile e $I_{g_s} = S$. Quindi poiché S é l'immagine di una funzione totale e computabile, è ricorsivamente enumerabile per definizione.

Questa è una definizione **costruttiva**. Infatti, non è definito né se $S = \emptyset$ né l'algoritmo per trovare un k specifico. Viene definita solo l'esistenza di g_s per $S \neq \emptyset$.

9.12.2 Dimostazione punto (B)

Dimostrazione che se S è ricorsivo, è anche ricorsivamente enumerabile.
Se S è vuoto o coincide con $\mathbb{N}$, allora l'enunciato è ovvio.
Altrimenti, esistono due funzioni totali e computabili $g_s, g_{\overline{s}}$ tali che $S = I_{g_s}, \overline{s} = I_{g_{\overline{s}}}$ nella forma:

$$S = \{g_s(0), g_s(1), \ldots\}$$
$$\overline{S} = \{g_{\overline{s}}(0), g_{\overline{s}}(1), \ldots\}$$

Per costruzione

$$S \cup \overline{S} = \mathbb{N} \quad S \cap \overline{S} = \emptyset$$

e quindi:

$$\forall x \in \mathbb{N} (\exists y \,|\, x = g_s(y) \vee x = g_{\overline{s}} \wedge \neg(\exists z, w \,|\, x = g_s(z) \wedge x = g_{\overline{s}}(w)))$$

cioè ogni x appartiene in modo esclusivo a S o a $\overline{S}$.

Si consideri ora l'enumerazione:

$$\mathscr{E}(L) = \{g_s(0), g_{\overline{s}}(0), g_s(1), g_{\overline{s}}(1), \ldots\}$$

Poiché $\mathscr{E} \equiv \mathbb{N}$, $\forall x, x \in \mathscr{L}$ e, se x compare in $\mathscr{L}$ in posizione parti, allora non compare mai in posizione dispari e viceversa.
Si definisca c_s come:

$$c_{s_i} = \begin{cases} 1 \text{ se } x = s_i \in \mathscr{L} \text{ e } i = 2 \cdot k + 1 \\ 0 \text{ altrimenti} \end{cases}$$

Chiaramente c_s è ben definito, poiché x potrà apparire in $\mathscr{L}$ solo in posizione o pari o dispari (e non entrambe). Inoltre risulta totale e computabile, in quanto l'enumerazione $\mathscr{L}$ può essere effettuata da una qualche TM. Basta infatti cercare se x (elemento dato) appare in posizione pari o dispari di $\mathscr{L}$ per concludere se appartiene o meno a S.

Dimostrazione che se S è ricorsivo, allora anche $\overline{S}$ lo è.
Per definizione, la funzione caratteristica di $\overline{S}$ è definita come:

$$g_{\overline{s}}(x) = \begin{cases} 1 \text{ se } c_s(x) = 1 \\ 0 \text{ altrimenti} \end{cases}$$

Quindi in virtù del punto (B) del teorema appena dimostrato, sia S che $\overline{S}$ sono ricorsivamente enumerabili.

9.12.3 Osservazioni di interesse pratico

Si consideri l'insieme S con le seguenti caratteristiche:

- $i \in S \to f_i$ totale (cioè S) contiene solo indici di funzione **totali** e **computabili**
- f è **totale** e **computabile**, $\exists\, i \in S \mid f_i = f$ e quindi S contiene ogni i

S è quindi l'insieme di indici di funzioni totali e computabili e non è ricorsivamente enumerabile.
Questa affermazione è dimostrabile per diagonalizzazione. Quindi non esiste un formalismo ricorsivamente enumerabile che possa definire tutte le funzioni computabili e totali e solo quelle. Gli *FSA* possono definire solo alcune *(non tutte)* funzioni totali, mentre le *TM* definiscono tutte le funzioni computabili, anche quelle non totali.
Il linguaggio `C` permette di codificare qualunque algoritmo, ma anche quelli che non terminano. Non esiste nessun sottoinsieme del linguaggio `C` che definisca esattamente tutti i programmi che terminano.

La relazione tra insiemi di problemi è illustrata nella Figura 33.

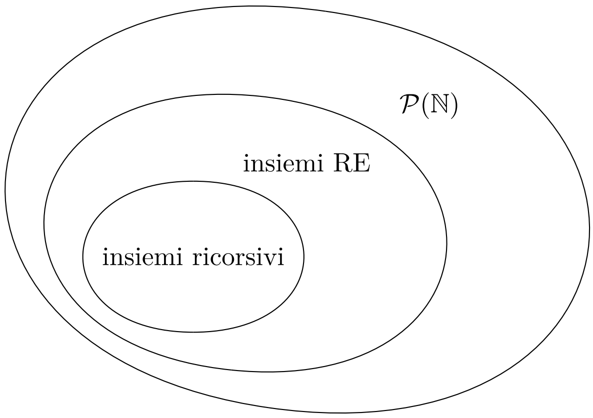


Figura 33: Relazione tra insiemi di problemi

9.13 Teoremi di *Kleene* e *Rice*

9.13.1 Teorema del punto fisso di *Kleene*

Sia t una funzione **totale** e **computabile**. Allora è sempre possibile trovare un intero p tale che:

$$f_p = f_{t(p)}$$

La funzione f_p è detta **punto fisso** di t perché t trasforma *(una macchina che calcola)* f_p in *(una macchina che calcola)* f_p stessa.

9.13.1.1 Dimostrazione

Sia u un qualsiasi numero naturale. Si definisca un *TM* che realizza la seguente procedura applicata al valore in ingresso x:

1. Calcola $z = f_u(u)$
2. Quando la computazione di $f_u(u)$ si ferma, se ciò avviene, calcola $f_z(x)$ che non è detto che sia definito

Poiché la procedura precedente è effettiva, esiste una *TM* in grado di implementarla. Inoltre è possibile costruire tale macchina e calcolare poi il suo indice $g(u)\,\forall\, u$, usando la numerazione $\mathscr{G}$ introdotta nella Sezione 9.5.

9.13.2 Teorema di *Rice*

Sia F un insieme qualunque funzioni **computabili**.
L'insieme $S = \{x \mid f_x \in F\}$ degli indici di *TM* che calcolano le funzioni di F è **decidibile** (e quindi **ricorsivo**) se e solo se $F = \emptyset$ oppure F è l'insieme di **tutte** le funzioni computabili.
Di conseguenza in tutti i casi non banali S **non è decidibile**.

9.13.2.1 Dimostrazione

Per assurdo, si supponga che S sia ricorsivo, che $F \neq \emptyset$ e che F non coincida con l'insieme di tutte le funzioni computabili.
Si consideri ora la funzione caratteristica c_s di S definita come:

$$c_s(x) = \begin{cases} 1 \text{ se } f_x \in F \\ 0 \text{ altrimenti} \end{cases}$$

Per ipotesi assurda, si supponga che c_s sia computabile.
Sempre per ipotesi, enumerando effettivamente tutte le *TM* M_i si può trovare:

a) Il primo $i \in S$ per cui $f_i \in F$
b) Il primo $j \notin S$ per cui $f_j \notin S$
c) La funzione $\overline{c}_s(x)$, computabile e totale, definita come

$$\overline{c}_s = \begin{cases} i \text{ se } f_x \notin F \\ j \text{ altrimenti} \end{cases}$$

d) Per il teorema di Kleene, un $\overline{x}$ tale che $f_{\overline{c}_s(\overline{x})} = f_{\overline{d}}$

Supponendo che $\overline{c}_s(\overline{x}) = i$, per (C), segue che $f_{\overline{x}} \notin F$. Tuttavia, per (D), si ha che $f_{\overline{x}}$ e per (A) si ha che $f_i \in F$, una **contraddizione**.
Supponendo invece che $\overline{c}_s(\overline{x}) = k$, per (D) si ha che $f_{\overline{x}} = f_j$. Tuttavia, per (C) si ha che $f_{\overline{x}} = f_j$ e, per (B) si ha che $f_j \notin F$, ottenendo ancora una **contraddizione**.

9.13.2.2 Conseguenze

Il teorema di *Rice* ha un forte impatto pratico negativo.
Ad esempio, ponendo $f = \{g\}$, quindi un insieme composto dalla sola funzione g, per il teorema **non è decidibile** se una generica *TM* calcoli g o meno. Tuttavia, per la tesi di *Church* (Sezione 9.4) il risultato non è ristretto alle *TM* e al formalismo delle funzioni. Non è quindi possibile stabilire algoritmicamente se un dato algoritmo sia in grado di risolvere un determinato problema o se due programmi siano equivalenti *(cioè se calcolano la stessa funzione)*.
Inoltre, non è possibile stabilire se un algoritmo risolve un problema che goda di una qualsiasi proprietà non banale *(e quindi una proprietà che non sia appartenenza ad una categoria non esistente o coincidente con tutti i problemi risolvibili)*.

9.14 Riduzione di problemi

Un problema P' è **ridotto** ad un problema P se un algoritmo per risolvere P viene usato per risolvere P'.
Quindi P è **risolvibile** ed esiste un algoritmo che, per ogni data istanza di P':

1. **Determina** una corrispondente istanza di P
2. **Costruisce** algoritmicamente la soluzione dell'istanza di P' dalla soluzione dell'istanza di P

Formalmente:

1. Si vuole verificare se $x \in S$
2. Si è in grado di verificare se $y \in S'$

3. Se esiste una funzione t, computabile e totale tale che

$$x \in S \Leftrightarrow t(x) \in S'$$

   allora è possibile determinare algoritmicamente se $x \in S$

Esempio: si supponga che il problema di cercare un elemento in un insieme sia risolvibile. Allora si può risolvere anche il problema di calcolare l'intersezione tra due sistemi, attuando una riduzione.

9.14.1 Implicazione della riduzione

Il procedimento della riduzione funziona anche al contrario:

- Si vuole sapere se è possibile risolvere $x \in S$
- Si sa che non è possibile risolvere $y \in S'$ (quindi S' *non è decidibile*)
- Se esiste una funzione t, computabile e totale, tale che

$$y \in S' \Leftrightarrow t(y) \in S$$

  allora è possibile concludere che $x \in S$ non è decidibile

In conclusione, è immediato dedurre che:

- Se P^* è **risolvibile**, lo è anche P
- Se P è **irrisolvibile**, lo è anche P^*

10 Complessità del calcolo

Fin'ora è stato affrontato la determinazione della computazione di un problema, senza che venisse però analizzato il vero *"costo"* della soluzione.
Si supponga infatti di avere accesso al più potente super calcolatore al mondo. Anche sfruttando tutta la sua capacità di calcolo, alcuni problemi *(come la determinazione della partita perfetta a scacchi)* può richiedere una quantità di tempo superiore all'intera vita dell'universo *(e si dimostra che questa tesi corrisponde alla realtà)*.
Di conseguenza, alcuni problemi vengono definiti **intrattabili**, perché pur esistendo un algoritmo per giungere alla sua soluzione, la sua esecuzione richiederebbe un tempo *"inaccettabile"*. L'intrattabilità costituisce una grossa limitazione alle applicazioni pratiche, in quanto esistono parecchi problemi *"interessanti"* che sono anche intrattabili.
Il concetto di **trattabilità**, è strettamente legato a quello della **complessità**, che può essere vista come la misura del *"costo"* della risoluzione: all'aumentare della prima, il secondo lieviterà. L'obiettivo di questa Sezione sarà definire in modo formale la nozione e lo studio del costo della soluzioni dei problemi, per poi essere in grado di progettare e combinare algoritmi e strutture dati in modo da realizzare soluzioni efficienti e non solo corrette.

10.1 Analisi della complessità

La tecnica che prende il nome di **analisi della complessità** consiste nel costruire strumenti per valutare la complessità di un calcolo, per poter successivamente analizzare algoritmi e strutture di nati notevoli.
Essi verranno sottoposti ad analisi quantitative su:

- Tempo di calcolo impiegato - *complessità temporale*
- Spazio di memoria occupato - *complessità spaziale*

L'analisi si limiterà a criteri di costo **oggettivi** e formalizzabili, quindi non terrà conto di parametri come i costi di sviluppo e *tradeoff* tra obiettivi contrastanti.

Per la Tesi di Church *(Sezione 9.4)*, un problema è calcolabile indipendentemente dallo strumento usato, purché esso sia Turing completo.
Questa affermazione non è valida per la complessità di calcolo: non è verosimile affermare che modelli di calcolo differenti impieghino lo stesso tempo per eseguire programmi analoghi. Infatti, poiché non esiste una *"Tesi di Church per la complessità"*, sarà necessario costruire uno strumento in grado di valutare la complessità temporale e spaziale di un calcolo che tralasci *"considerazioni superflue"* e che sia utilizzabile per la maggioranza dei modelli di calcolo.
Contrariamente a quanto appena affermato, nonostante l'inesistenza di tale teorema, sarà possibile correlare in modo sistematico soluzioni proposte per modelli diversi. Poiché non esiste un formalismo di calcolo perfettamente adatto a costruire il modello, l'analisi avverrà a partire dalle *TM* deterministiche.

10.2 Complessità temporale e spaziale

Sia M una *TM* **deterministica** con k nastri e sia $c_0 \vdash^* c_r$ una computazione su M. Allora è possibile definire:

- La **complessità temporale** come:
  - $T_M(x) = r$ se M termina in c_r
  - ∞ se M non termina
  - poiché M è deterministica, la computazione sarà **unica** sull'ingresso x
- La **complessità spaziale** come:
  - $S_M(x) = \sum_{j=1}^{k} \max_{i \in \{0,\dots,r\}} (|\alpha_{ij}|)$

    → $(|\alpha_{ij}|)$ indica il contenuto del j-esimo nastro alla i-esima mossa
    → la complessità è pari alla somma delle quantità massime di nastro occupate per ogni nastro
  - la complessità **spaziale** è sempre minore della complessità **temporale**

    → $\forall\, x \ \dfrac{S_M(x)}{k} \leq T_M(X)$

In queste definizioni tuttavia si tiene conto della natura dell'ingresso x, mentre risulta più conveniente studiare la complessità di un algoritmo in funzione della **dimensione** di x. Quindi si semplifica lo studio di $T_M(x)$ ed $S_M(x)$ ponendo $n = |x|$ *(la dimensione di x)* e studiando $T_M(n)$ e $S_M(n)$.
Questa semplificazione implica une perdita di generalità perché ingressi diversi, seppur di dimensioni uguali, potrebbero avere complessità diverse. *Formalmente*:

$$|x_1| = |x_2| \nRightarrow T_M(x_1) = T_M(x_2)$$
$$|x_1| = |x_2| \nRightarrow T_S(x_1) = T_S(x_2)$$

Per gestire la variabilità dell'ingresso così introdotta, è necessario distinguere tra:

- **Caso pessimo** $T_M(n) = \max\limits_{n=|x|} (T_M(x))$
- **Caso ottimo** $T_M(n) = \min\limits_{n=|x|} (T_M(x))$
- **Caso medio** $T_M(n) = \dfrac{\sum\limits_{n=0}^{|x|} T_M(x)}{|I|^n}$

  $\rightarrow$ somma dei tempi per parole lunghe n *diviso* il numero di parole lunghe n
  $\rightarrow$ è una media pesata

Tipicamente, si considera il caso pessimo, perché è normalmente il più rilevante *(fornisce un limite superiore)* e l'analisi risulta più semplice che nel caso medio *(che dovrebbe tenere conto di ipotesi probabilistiche sulla distribuzione dei dati, complicando ulteriormente l'analisi stessa).*

10.2.1 Crescita di $T_M(n)$ e $S_M(n)$

Subito ci si può accorgere di come i valori esatti di $T_M(n)$ e $S_M(n)$ non siano particolarmente utili ai fini dello studio della complessità di un particolare problema. Ciò che risulta rilevante, infatti, è il comportamento **asintotico** delle funzioni di costo, ossia quando $n \to \infty$.
Questa è una semplificazione *"aggressiva"*, perché porta a perdere la distinzione tra casi del tipo

$$T_M(n) = n^4 - 5n^2 + 2$$
$$T_M(n) = 10^{80} \cdot n^4$$

entrambi *ridotti* a n^4, seppur con andamenti radicalmente diversi per valori finiti.

Si introducono quindi tre notazioni per indicare il comportamento asintotico di una funzione:

- $\mathcal{O}$-grande: limite asintotico **superiore**
- Ω-grande: limite asintotico **inferiore**
- Θ-grande: limite asintotico sia **superiore** che **inferiore**

L'uso di questa notazione (tramite gli insiemi $\mathcal{O}$, Ω, Θ) permette di evidenziare con facilità la parte più importante di una funzione di complessità.
Questo modello tuttavia presenta delle limitazioni:

- Potrebbe non essere sufficientemente accurato per valori **piccoli** di n
  - la notazione Ω-grande è più precisa rispetto alla notazione $\mathcal{O}$-grande per i valori piccoli di n
- Algoritmi con complessità asintotica maggiore possono essere più veloce di uno a complessità minore per valori **piccoli** di n

Per ovviare a questi problemi, si faranno uso di tutte e tre le notazioni in base ai casi presi in analisi.

10.2.2 Notazione $\mathcal{O}$-grande

Definizione: siano $f(n),\ g(n)$ funzioni $\mathbb{N} \to \mathbb{R}^+$. Allora $\mathcal{O}(g(n))$ è l'insieme di funzioni definito come:

$$\mathcal{O}(g(n)) = \{f(n) \,|\, \exists\, c > 0, n_0 > 0 \text{ tali che } \forall\, n > n_0,\ 0 \leq f(n) \leq c \cdot g(n)\}$$

Informalmente, le funzioni in $\mathcal{O}(g(n))$ sono *dominate* da $c \cdot g(n)$ a partire da n_0. Un'illustrazione di questa relazione è rappresentata in Figura 34.

Per brevità, è comune scrivere $f(n) = \mathcal{O}(g(n))$ al posto della più corretta notazione $f(n) \in \mathcal{O}(g(n))$.

10.2.3 Notazione Ω-grande

Definizione: siano $f(n),\ g(n)$ funzioni $\mathbb{N} \to \mathbb{R}^+$. Allora $\Omega(g(n))$ è l'insieme di funzioni definito come:

$$\Omega(g(n)) = \{f(n) \,|\, \exists\, c > 0, n_0 > 0 \text{ tali che } \forall\, n > n_0,\ 0 \leq c \cdot g(n) \leq f(n)\}$$

Informalmente, le funzioni in $\Omega(g(n))$ *dominano* $c \cdot g(n)$ a partire da n_0. Un'illustrazione di questa relazione è rappresentata in Figura 35.

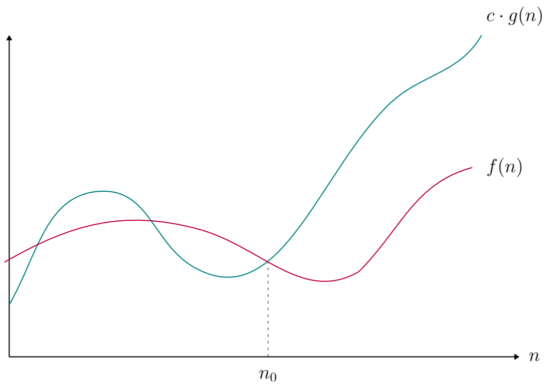


Figura 34: Notazione $\mathcal{O}$-grande

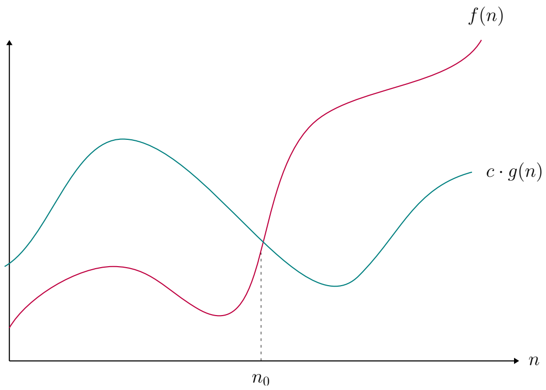


Figura 35: Notazione Ω-grande

10.2.4 Notazione Θ-grande

Definizione: siano $f(n),\ g(n)$ funzioni $\mathbb{N} \to \mathbb{R}^+$. Allora $\Theta(g(n))$ è l'insieme di funzioni definito come:

$$\Theta(g(n)) = \{f(n) \,|\, \exists\, c_1 > 0, c_2 > 0, n_0 > 0 \text{ tali che } \forall\, n > n_0,\ 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

Informalmente, le funzioni in $\Theta(g(n))$ *sono comprese* tra $c_1 \cdot g(n)$ e $c_2 \cdot g(n)$ a partire da n_0. Un'illustrazione di questa relazione è rappresentata in Figura 36.

10.2.4.1 Definzioni come limiti

Se è vero che:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c, \quad c \neq 0, \quad c \neq \infty$$

allora $f(n) \in \Theta(g(n))$ e gli andamenti asintotici di $f,\ g$ differiscono per una costante moltiplicativa.

Allo stesso modo, se è vero che:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

allora $f(n) \in \mathcal{O}(g(n)),\ f(n) \notin \Theta(g(n))$ e f cresce più velocemente di g.
Alternativamente si usa la notazione $\Theta(f(n)) < \Theta(g(n))$.

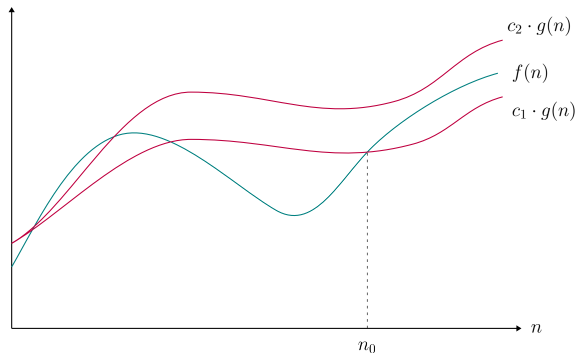


Figura 36: Notazione Θ-grande

10.2.5 Proprietà di $\mathcal{O}$, Ω, Θ

- Per definizione:
  - $f \in \Theta(g) \Leftrightarrow f \in \mathcal{O}(g) \vee f \in \Omega(g)$
- Proprietà **transitiva**
  - se $f(n) = \Theta(g(n)),\ g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - se $f(n) = \mathcal{O}(g(n)),\ g(n) = \mathcal{O}(h(n)) \Rightarrow f(n) = \mathcal{O}(h(n))$
  - se $f(n) = \Omega(g(n)),\ g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$
- Proprietà **riflessiva**
  - $f(n) = \Theta(f(n))$
  - $f(n) = \mathcal{O}(f(n))$
  - $f(n) = \Omega(f(n))$
- **Simmetria**
  - $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$
- **Simmetria trasposta**
  - $f(n) = \mathcal{O}(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

10.2.6 Complessità dei modelli deterministici di calcolo

Si vuole ora studiare la complessità dei modelli **deterministici** di calcolo studiati fin'ora. Il risultato della trattazione è illustrato in Tabella 6.

modello	*complessità spaziale*	*complessità temporale*	*note*
FSA	$S(n) \in \Theta(1)$	$T(n) \in \Theta(n)$	più precisamente $T(n) = n$
PDA	$S(n) \in \mathcal{O}(n)$	$T(n) \in \Theta(n)$	
TM a nastro singolo	$S(n) \in \Theta(n)$	$T(n) \in \Theta(n^2)$	meno efficienti dei *PDA*

Tabella 6: Complessità dei modelli di calcolo

10.3 Teoremi di accelerazione lineare

10.3.1 Teorema 1

"Se L è accettato da una TM M a k nastri in $S_M(n)$, per ogni $c \in \mathbb{R}^+$ è possibile costruire una TM M' a k nastri che accetta L con $S_{M'}(n) < c \cdot S_M(n)$"

10.3.1.1 Dimostrazione teorema 1

Schema della dimostrazione:

1. Si sceglie una fattore di compressione r tale che $r \cdot c > 2$
2. Per ogni alfabeto Γ_i dell'i-esimo nastro di M si costruisce Γ'_i dell'i-esimo nastro di M' assegnando un elemento per ogni $s \in \Gamma_i^r$
3. Si costruisce l'organo di controllo di M' in modo tale per cui:
    - calcoli i nuovi simboli sui nastri emulando le mosse di M spostando le testine sui nastri ogni r movimenti di M
    - memorizzi la posizione della testina arricchendo ulteriormente gli alfabeti di nastro Γ_i **oppure** arricchendo l'insieme degli stati

10.3.2 Teorema 2

"Se L è accettato da una TM M a k nastri in $S_M(n)$, è possibile costruire una TM M' a 1 nastro che accetta L con $S_{M'}(n) = S_M(n)$, concatenando tutti i nastri su una solo"

10.3.3 Teorema 3

"Se L è accettato da una TM M a k nastri in $S_M(n)$, per ogni $c \in \mathbb{R}^+$ è possibile costruire una TM M' a 1 nastro che accetta L con $S_{M'}(n) < S_M(n)$"

Il risultato è analogo a quello del Teorema 2, con in aggiunta la compressione già vista nel Teorema 1.

10.3.4 Teorema 4

"Se L è accettato a una TM M a k nastri in $T_M(n)$, per ogni $c \in \mathbb{R}^+$, è possibile costruire una TM M' a $k+1$ nastri che accetta L con $T_{M'}(n) = \max(n+1, c \cdot T_M(n))$"

Il risultato è analogo a quello del Teorema 2, con in aggiunta la compressione già vista nel Teorema 1.

10.3.4.1 Dimostrazione teorema 4

L'approccio della dimostrazione è analogo a quello già visto in 10.3.1.1

1. Si codifica in modo compresso i simboli dell'alfabeto di M
2. Poiché la compressione avviene a *runtime* il caso ottimale è $T_{M'}(n) = n$
3. Comprimendo r simboli in 1 nel caso pessimo servono 3 mosse di M' per emularne $r+1$ di M

10.3.5 Conseguenze pratiche

Questi teoremi portano alle seguenti conseguenze pratiche:

- Lo schema di dimostrazione usato per le *TM* vale anche per il modello di calcolatore di Von Neumann
    - → È possibile avere accelerazioni lineari arbitrariamente grandi **aumentando il parallelismo fisico**
- Miglioramenti più che lineari nel tempo di calcolo possono essere ottenuti solo **cambiando algoritmo**

→ concepire ed utilizzare algoritmi **efficienti** è molto più efficiente di procedere di *forza bruta*

- Un calcolatore è in grado di eseguire operazioni aritmetiche su tipi a dimensione finita in tempo costante, mentre la *TM* richiede di propagare gli effetti al singolo `bit`, uno alla volta
  - → il calcolatore opera su un alfabeto più vasto, di dimensione $|I| = 2^w$, con w dimensione della parola
  - → un calcolatore può accedere direttamente ad una cella di memoria, mentre una *TM* impiega $\Theta(n)$, con n pari alla distanza di quest'ultima dalla testina

10.4 La macchina *RAM*

L'introduzione della macchina *RAM* consente di fare un passo per arrivare ad un livello di astrazione più vicino alla realtà. È infatti un modello classico, ispirato all'architettura di *Von Neumann*.
La **macchina *RAM*** è dotata di un nastro di **lettura** (`IN`) e uno di **scrittura** (`OUT`), similarmente ad una *TM*. Il programma è cablato all'interno dell'organo di controllo tramite una serie *(finita)* di istruzioni in un certo linguaggio. Esse **non sono alterabili** durante il funzionamento della macchina. L'indice dell'istruzione da eseguire in ogni dato momento è indicato dal *program counter*.
L'accesso alla memoria avviene tramite indirizzamento diretto, in cui ogni cella (contenente un intero) viene letta o scritta dalla *unità aritmetica*. Ogni cella di memoria è quindi associata ad un intero e definita come `N`$[n]$, $n \in \mathbb{N}$. Grazie alla sua struttura, l'accesso ad una cella di memoria non implica uno scorrimento delle stesse. Le istruzioni del programma caricato nell'organo di controllo usano come primo operando sorgente o destinazione il primo elemento della memoria, `N`$[0]$, detto **accumulatore**.

La struttura di una macchina *RAM* è mostrata nella Figura 37.

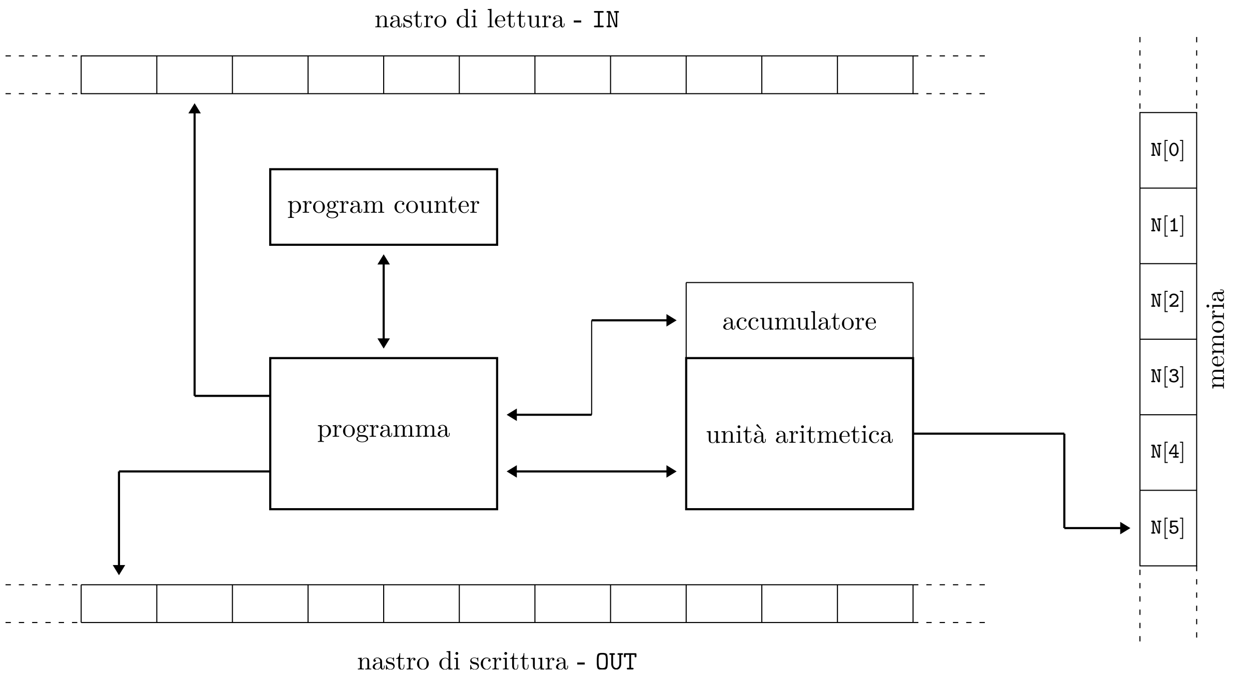


Figura 37: Struttura di una macchina *RAM*

I programmi all'interno dell'unità di controllo sono codificati tramite una semantica simile al codice `Assembly`, seppur di molto semplificato. Viene definita una specifica dei comandi (un *Instruction Set*) che permette operazioni matematiche, logiche e di salto.
Un'esempio di tale linguaggio è mostrato nella Tabella 7.

10.4.1 Criterio di costo logaritmico

L'approssimazione di costo tramite Ω, Θ funziona ma presenta molti limiti.

istruzione	*semantica*
`LOAD X`	$N[0] \leftarrow N[X]$
`LOAD= X`	$N[0] \leftarrow X$
`LOAD* X`	$N[0] \leftarrow N[N[X]]$
`STORE X`	$N[X] \leftarrow N[0]$
`STORE* X`	$N[N[X]] \leftarrow N[0]$
`ADD X`	$N[0] \leftarrow N[0] + N[X]$
`SUB X`	$N[0] \leftarrow N[0] - N[X]$
`MUL X`	$N[0] \leftarrow N[0] \times N[X]$
`DIV X`	$N[0] \leftarrow N[0] \div N[X]$
`HALT`	-

istruzione	*semantica*
`READ X`	$N[[X] \leftarrow IN]$
`READ* X`	$N[N[X]] \leftarrow IN$
`WRITE X`	$OUT \leftarrow N[X]$
`WRITE= X`	$OUT \leftarrow X$
`WRITE* X`	$OUT \leftarrow N[N[X]]$
`JUMP l`	$PC \leftarrow l$
`JZ l`	$PC \leftarrow l$ se $N[0] = 0$
`JGZ l`	$PC \leftarrow l$ se $N[0] > 0$
`JZZ l`	$PC \leftarrow l$ se $N[0] < 0$

Tabella 7: Instruction set e semantica pseudo-*RTL*

Infatti non si tiene conto di una limitatezza della memoria: una singola parola di memoria non può sempre contenere in un solo simboli tutti gli interi usati in un determinato algoritmo. Poiché non si considera il numero di cifre necessarie a rappresentare un numero, il calcolo della complessità può risultare molto sbagliato.
Per effettuare un'approssimazione più simile alla realtà, è necessario tenere conto del numero di cifre necessarie a rappresentare un intero. Così facendo risulterà che caricare e salvare interi in memoria non sarà più a costo costante (ma varierà in funzione della lunghezza del numero), con effetti analoghi sulle operazioni algebriche.
Alla luce di questi ragionamenti, sarà possibile calcolare la complessità per ogni categoria di operazione.

- **Copiare**, **spostare**, **scrivere**, **leggere** un intero i avrà complessità pari al suo numero di cifre in base b:
  - → $log_b(i) = \Theta(\log_2(i))$
  - → con $b = 2$ il costo corrisponde al numero di `bit` usati per rappresentare i
- Il costo delle operazioni **aritmetiche** e **logiche** elementari dipende dall'operazione. Ponendo $d = \log_2(i)$:
  - → **addizioni**, sottrazioni saranno $\Theta(d)$
  - → **divisioni** e moltiplicazioni con **metodo scolastico** $\Theta(d^2)$
  - → **divisioni** e moltiplicazioni con **algoritmi più efficienti** $\Theta(n \log(n))$
- Operazioni di **salto**:
  - → il costo è **costante**, quindi $\Theta(1)$

10.4.2 Scelta del criterio di costo

La scelta del criterio di costo (tra logaritmico o costante) dipende da una serie di fattori. Infatti, se:

- L'elaborazione **non altera l'ordine di grandezza** dei dati di ingresso
- La memoria allocata inizialmente **non varia durante l'esecuzione** del programma
  - la memoria **non dipende** dai dati
  - di conseguenza **una singola cella è elementare** e con essa le operazioni relative
  - la dimensione di ogni singolo elemento in ingresso **non varia significativamente** nel tempo

allora il criterio di costo costante è adeguato. In caso contrario è indispensabile il criterio di costo logaritmico, l'unico in grado di approssimare con correttezza il costo della funzione.

La conseguenza immediata di questa differenza di complessità implica che con macchine diverse lo stesso algoritmo può dare luogo a complessità diverse. Non è possibile identificare un modello *"migliore"* tra i vari formalismi studiati e non esiste un'analogo alla Tesi di Church (Sezione 9.4), però è possibile stabilire una relazione di maggiorazione a priori tra le complessità dei diversi modelli di calcolo. Essa prende il nome di *Tesi di correlazione polinomiale.*

10.5 Tesi di correlazione polinomiale

La Tesi di correlazione polinomiale, in analogia con la Tesi di Church, prova una relazione tra le complessità di diverse modelli di automi. Essa afferma che:

> *"Se un problema è risolvibile mediante il modello $\mathscr{M}_1$ con complessità spaziale o temporale $C_1(n)$, allora è risolvibile da un qualsiasi altro modello Turing completo $\mathscr{M}_2$ con complessità $C_2(n) \leq \pi\left(C_1(n)\right)$, con π polinomio "*

10.5.1 Dimostrazione della Tesi di correlazione polinomiale

L'obiettivo di questa sezione sarà dimostrare la Tesi di correlazione polinomiale *(che d'ora in poi prenderà quindi la definizione di Teorema).*
Essa sarà strutturata in due parti, che mostreranno rispettivamente la simulazione di una *TM* tramite *RAM* e viceversa.

10.5.1.1 Simulazione di una *TM* a k nastri tramite *RAM*

Verrà ora dimostrato il funzionamento della simulazione delle azioni una *TM* a k nastri tramite macchina *RAM*. Inizialmente, sarà necessario compiere questi passi per inizializzare la memoria della *RAM*

- La *TM* viene mappata sulla *RAM*:
  - lo stato della *TM* verrà posto sulla prima cella di memoria della *RAM*
  - verrà usata una cella di memoria della *RAM* per ogni cella del nastro
  - la memoria rimanente viene divisa in blocchi da 4 celle
- I blocchi vengono riempiti con la seguente strategia:
  - Il blocco 0 conterrà le posizioni delle k testine
  - I blocchi n-esimi, $n > 0$, conterranno l'n-esimo simbolo di ognuno dei k nastri
- La *RAM* emula la lettura di un carattere sotto la testina con un accesso indiretto, usando l'indice contenuto nel blocco 0

Una volta inizializzata la memoria, sarà possibile simulare propriamente le azioni della *TM*. In particolare:

- Lettura dei nastri
  - lettura del blocco 0 e dello stato ($\Theta(k)$ mosse)
  - lettura dei valori sui nastri in corrispondenza delle testine ($\Theta(k)$ accessi indiretti)
- Scrittura dei nastri
  - scrittura dello stato ($\Theta(1)$)
  - scrittura della celle dei nastri ($\Theta(k)$ accessi indiretti)
  - scrittura nel blocco 0 per aggiornare le posizioni delle k testine ($\Theta(k)$ mosse)

Per valutare la complessità, bisogna considerare che la *RAM* emula una mossa della *TM* con $\Theta(k)$ mosse:

- $T_{RAM}(n) = \Theta\left(T_{\mathscr{M}}(n)\right)$ per il costo costante
- $T_{RAM}(n) = \Theta\left(T_{\mathscr{M}}(n)\right) \log\left(T_{\mathscr{M}}(n)\right)$ per il costo logaritmico

10.5.1.2 Simulazione di una *RAM* tramite *TM* a k nastri

In questo paragrafo verrà dimostrata la simulazione di una *RAM* tramite *TM* a k nastri, omettendo le operazioni `MUL` e `DIV` per semplicità.

- Un nastro di memoria della *TM* viene inizializzato come in Figura 38

...	\$	\$	i_1	\$	$M[i_1]$	\$	\$	$\cdots$	i_k	\$	$M[i_k]$	\$	\$	b	...

Figura 38: Inizializzazione della memoria *RAM*

- le celle di memoria indicizzate da i sono state coinvolte da almeno una operazione di `STORE`
- il simbolo \$ è usato come delimitatore tra indice della cella e contenuto della cella
- il simbolo \$\$ è usato per separare tra di loro le celle di memoria

- Un secondo nastro di memoria ospita il contenuto dell'accumulatore $\mathtt{N}[0]$
  - il valore viene memorizzato con codifica binaria
- Un terzo nastro viene utilizzato come memoria temporanea
  - viene usato per spostare i dati qualora sia necessario memorizzare $\mathtt{N}[i_j]$ laddove $\mathtt{N}[i_k]$, $\mathtt{N}[i_l]$, $i_k < i_j < i_l$

Una volta inizializzata la memoria, sarà possibile simulare propriamente le azioni della macchina *RAM*. In particolare:

- L'operazione `LOAD x` avverrà come:
  1. ricerca di `x` sul nastro principiale
  2. copia di dati letti sulla cella corrispondente a $\mathtt{N}[0]$ usando il nastro di supporto
- L'operazione `STORE x` avverrà come:
  1. ricerca di `x` sul nastro principiale
  2. se esso non viene trovato, è necessario creare dello spazio usando il nastro di servizio *(se necessario)*
  3. valore di $\mathtt{N}[0]$ viene salvato in $\mathtt{N}[\mathtt{x}]$
- L'operazione `ADD x` avverrà come:
  - ricerca di `x`
  - copia di $\mathtt{N}[x]$ sul nastro di supporto
  - calcolo della somma
  - scrittura del valore risultante in $\mathtt{N}[0]$

In generale, la simulazione di una mossa della *RAM* richiede alla *TM* un numero di mosse n inferiore alla lunghezza del nastro principale per una costante opportuna. In simboli:

$$n \leq c \cdot \text{ lunghezza nastro principale}$$

10.5.2 Lemma della Tesi di correlazione polinomiale

In seguito all'enunciazione e alla dimostrazione del Teorema di correlazione polinomiale, risulta che:

> *"Lo spazio occupato sul nastro principale è $\mathcal{O}\left(T_{RAM}(n)\right)$"*

10.5.2.1 Dimostrazione del lemma

- Ogni cella i_j-esima della *RAM* occupa $\log(i_j) + \log(\mathtt{N}[i_j])$
- Ogni cella della *RAM* viene materializzata solo se la *RAM* effettua una operazione di `STORE`
- L'operazione `STORE` ha un costo per la *RAM* pari a $\log(i_j) + \log(\mathtt{N}[i_j])$
- Per riempire r celle, la *RAM* impiega $\displaystyle\sum_{j=1}^{r} \log(i_j) + \log(\mathtt{N}[i_j])$ unità di tempo, la stessa quantità di spazio occupata sul nastro

10.5.3 Conclusioni sulla Tesi di correlazione

A valle delle dimostrazioni, si può concludere che:

- La *TM* impiega al più $\Theta(T_{\text{RAM}}(n))$ per simulare una mossa della *RAM*
- Se la *RAM* ha complessità $T_{\text{RAM}}(n)$, essa effettua al più $T_{\text{RAM}}(n)$ mosse (ogni mossa costa almeno 1)
- La simulazione completa della *RAM* da parte della *TM* costa al più $\Theta\left((T_{\text{RAM}}(n))^2\right)$
- Il legame tra $T_{\text{RAM}}(n)$ e $T_{\text{TM}}(n)$ è polinomiale

Inoltre, è possibile fare le seguenti osservazioni:

- Per quanto grande possa essere il grado di un polinomio, sarà sempre inferiore ad una funzione esponenziale

  $\rightarrow$ è sufficiente confrontare l'andamento delle funzioni n^k e 2^n
- Grazie al teorema di correlazione polinomiale è possibile parlare dei problemi risolvibili in tempo e/o spazio polinomiale astraendo dalla classe del calcolatore
- classe dei problemi trattabili in pratica $\equiv$ classe dei problemi risolvibili in tempo polinomiale, detta $\mathcal{P}$
- I problemi di interesse pratico che sono in $\mathcal{P}$ hanno anche grado del polinomio *"accettabile"*

11 Concetti base di complessità degli algoritmi

TODO

12 Conclusioni sparse delle precedenti Sezioni

L'obiettivo di questa Sezione sarà ricapitolare delle conclusioni tratte nelle precedenti sezioni, con particolare riguardo agli esercizi ed alle applicazioni pratiche.

12.1 Scala di potenza delle classi di automi

Le classi di automi possono essere rappresentati in una scala, in ordine dal **più** al **meno** potente. Ciò avviene nella Figura 39.
Si noti che *NFA* e *FSA* hanno potenza equivalente in quanto esiste un algoritmo per convertire i primi nei secondi.

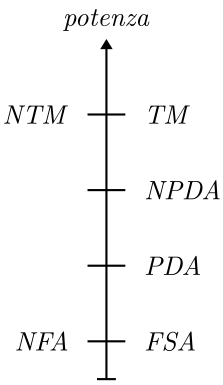


Figura 39: Scala di potenza degli automi

12.2 Chiusura degli automi rispetto alle operazioni

La chiusura degli automi rispetto alle operazioni è mostrata all'interno della Tabella 8.

classe di automi	*unione* $\cup$	*intersezione* $\cap$	*complemento* A^c	*differenza* $\setminus$	*stella di Kleene* A^*	*concatenazione* $\cdot$
FSA, NFA	✔	✔	✔	✔	✔	✔
PDA	✖	✖	✔	✖	✖	✖
NPDA	✔	✖	✖	✖	✔	✔
TM, NTM	✔	✔	✖	✖	✔	✔

Tabella 8: Chiusura degli automi rispetto alle operazioni

12.3 Automi e grammatiche di Chomsky

All'interno della Tabella 9 è mostrata la relazione tra grammatiche secondo la caratterizzazione di Chomsky e corrispondente formalismo a potenza minima che la riconosce.

12.4 Pattern tipici delle grammatiche

Si riconoscono pattern tipici per generare delle strutture note all'interno di grammatiche. Essi sono mostrati in Tabella 10.

grammatica	*formalismo*	*note*
0	*TM*	*grammatiche **generali** e **dipendenti** dal contesto*
1	*linear bounded automaton*	*non trattati nel corso*
2	*NPDA*	*grammatiche **non dipendenti** dal contesto*
3	*FSA, NFA*	*grammatiche **regolari***

Tabella 9: Relazione tra grammatiche e formalismi

linguaggio	*pattern*	*note*
$a^n b^n, n \geq 0$	$S \to aSb \mid \epsilon$	*stella di Kleene*
$a^n b^n, n \geq 1$	$S \to aSb \mid ab$	*più di Kleene*
$ww^r,\ w = (a \mid b)^*$	$S \to aSa \mid aBb \mid a \mid b \mid \epsilon$	*stringhe palindrome*
$(ab^+)^*$	$\begin{cases} S \to ah \mid \epsilon \\ h \to bh \mid bS \end{cases}$	

Tabella 10: Pattern tipici

12.4.1 Sintetizzazione di grammatiche di tipo 0

Idea generale: simulare i nastri di una *TM* tramite le regole della grammatica. In dettaglio:

- I nastri memorizzano i caratteri non terminali
- Altri caratteri non terminali simulano le testine
- I simboli nei nastri non mossi attraverso regole di *"swap"*

12.5 Linee guida sulla decidibilità

12.5.1 Casi immediati

- C'è una domanda di tipo **booleano** la cui risposta non dipende da alcun parametro esterno?

  $\Rightarrow$ La domanda è **chiusa** e il problema è **decidibile**
- La funzione in questione consiste di un **numero finito di casi**, tutti singolarmente decidibili e calcolabili?

  $\Rightarrow$ La funzione è **calcolabile** e il problema è **decidibile**

12.5.2 Caso del programmatore

È possibile scrivere un programma in un generico linguaggio (sia esso `C`, `Java`, `...`) che risolve il problema dato?

$\Rightarrow$ Se **sì**, la funzione è **calcolabile** e il problema è **decidibile**

Ciò implica che sia possibile scrivere un programma che per ogni possibile ingresso sia in grado di calcolare il valore corretto dell'uscita. Se per qualche valore dell'ingresso l'uscita non è definita, è sufficiente far entrare il programma in un loop infinito.

12.5.3 Riduzioni

- Esiste un problema **indecidibile** che è un caso particolare del problema in analisi
  ⇒ Il problema in analisi è **indecidibile**
- Esiste un problema **decidibile** che è un caso particolare del problema in analisi
  ⇒ Il problema in analisi è **indecidibile**

12.5.4 Applicazione del teorema di Rice

Il teorema di Rice può essere applicato nei casi in cui si deve verificare se un programma *(o analogamente una TM o un algoritmo)*:

- Ha una data **proprietà** relativa alla funzione da esso calcolata
- Calcola una **funzione** tra quelle di un insieme dato

Infatti, se:

- L'insieme di funzioni identificato **non è banale**
  ⇒ Il problema in analisi è **indecidibile**
- L'insieme di funzioni identificato **è banale**
  ⇒ Il problema in analisi è **decidibile**

Si ricordi che un insieme di funzioni è banale se è **vuoto** o se è l'insieme di **tutte le funzioni computabili**.

12.5.5 Ricorsività di un insieme S di numeri naturali

- S è **finito**
  ⇒ S è **ricorsivo**
- S è **infinito**
  - La funzione caratteristica di S è **computabile**
    ⇒ S è **ricorsivo**
  - S può essere espresso come insieme di indici di TM con una **proprietà comune relativa alla funzione che calcolano**
    ⇒ **Si può** usare il teorema di Rice
  - S può essere espresso come insieme di indici di TM con una **proprietà comune non relativa alla funzione che calcolano**
    ⇒ **Non si può** usare il teorema di Rice