UNIVERSITÀ DEGLI STUDI
DI GENOVA

Università degli Studi di Genova

Teoria degli Automi e Calcolabilità

Lorenzo Vaccarecci

Anno Accademico 2024/2025

Capitolo 0: Indice

Capitolo 1: Preliminari

1.1 Alfabeti, stringhe, linguaggi

Definizione 1.1.1: Alfabeto

Insieme finito **non vuoto** di oggetti detti *simboli*.

Definizione 1.1.2: Stringa

Una stringa u su un alfabeto Σ è una funzione totale da $[1, n]$ in Σ, per qualche $n \in \mathbb{N}$. n si dice *lunghezza* di u e si indica con $|u|$.

- $[1, n] \rightarrow$ sono le posizioni dei simboli all'interno della stringa
- Per **funzione totale** intendiamo che per ogni posizione nell'intervallo $[1, n]$ deve avere un simbolo corrispondente

Da ora in poi:

- $\sigma \rightarrow$ simboli generici
- $u, v, w \rightarrow$ stringhe generiche
- $\bigwedge$ o $\varepsilon \rightarrow$ stringa vuota con lunghezza zero ($|\bigwedge| = 0$ o $|\varepsilon| = 0$)

Definizione 1.1.3: Linguaggio L

E' un insieme di stringhe su Σ, ossia un sottoinsieme di Σ^* **infinito e numerabile**.

- **Infinito**: contiene un numero illimitato di elementi
- **Numerabile**: esiste una funzione iniettiva da Σ^* all'insieme dei numeri naturali $\mathbb{N}$

L'insieme $\emptyset$ è un linguaggio che non contiene alcun elemento (neanche la stringa vuota). L'insieme $\{\varepsilon\}$ è composto solo da una stringa di lunghezza 0. *Esempio:*

- $\Sigma = \{a, b\}$
- $L = \Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$

Un linguaggio può non contenere ε.

Definizione 1.1.4: Operazioni su stringhe (concatenazione)

Se u e v sono stringhe di lunghezza n ed m rispettivamente, allora $u \cdot v$ è la stringa di lunghezza $n+m$, definita da

$$(u \cdot v)(k) = \begin{cases} u(k) & \text{se } 1 \leq k \leq n \\ v(k-n) & \text{se } n < k \leq n+m \end{cases}$$

Questa operazione è associativa.

Definizione 1.1.5: Operazioni su linguaggi (concatenazione)

Se L e L' sono linguaggi, $L \cdot L' = \{u \cdot v \mid u \in L, v \in L'\}$. Se si esegue:

$$L \cdot \{\varepsilon\} = L \qquad L \cdot \emptyset = \emptyset$$
$$L^0 = \{\varepsilon\} \qquad L^{n+1} = L \cdot L^n \qquad \text{con } n \geq 0$$

- **Chiusura di Kleene** L^*: $L^* = \cup_{n \geq 0} L^n \rightarrow$ collezione di tutte le sequenze possibili di elementi di L, inclusa la stringa vuota (L^0). *Esempio:*
  - $\Sigma = \{a, b\}$
  - $L = \{a\}$
  - $L^* = \{\varepsilon, a, aa, aaa, \ldots, a^n\}$
- **Chiusura positiva** L^+: $L^+ = \cup_{n > 0} L^n \rightarrow$ differisce da L^* solo per l'esclusione della stringa vuota, a meno che L stesso contenga ε. *Esempio:*
  - $L^+ = \{a, aa, aaa, \ldots, a^n\}$

Questa operazione è associativa.

Capitolo 2: Automi a stati finiti

2.1 Linguaggi regolari

2.1.1 Automi a stati finiti

Definizione 2.1.1: Automa a stato finito deterministico (DFA)

E' una quintupla $\mathcal{M} = \langle Q, \Sigma, \delta, q_0, F \rangle$ dove:

- Q è un insieme finito di stati
- Σ è un alfabeto di input
- $\delta : Q \times \Sigma \to Q$ è una funzione totale detta **funzione di transizione** ovvero stabilisce come l'automa si muove da uno stato all'altro in base al simbolo che legge dall'input
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati finali

La definizione del sistema di transizione associato a un DFA $\mathcal{M}$ è:

- **Configurazioni**: sono rappresentate da una coppia $\langle q, u \rangle$:
  - $q \in Q$ è lo stato corrente in cui si trova l'automa
  - $u \in \Sigma^*$ è la parte della stringa di input che deve ancora essere letta
- **Relazione di Riduzione (o Transizione)**: la relazione di riduzione, indicata con $\to$, definisce come l'automa si muove da una configurazione all'altra:

$$\langle q, \sigma u \rangle \to \langle q', u \rangle \text{ se } \delta(q, \sigma) = q'$$

  - Significa che l'automa è nello stato q e il prossimo simbolo da leggere è σ (con u come resto della stringa) e δ porta dallo stato q allo stato q' leggendo σ, allora l'automa si sposta nello stato q' e il simbolo σ viene "consumato" dall'input
  - Questa relazione di riduzione è **deterministica** (per ogni configurazione, c'è al più una transizione possibile) e **terminante** (ad ogni passo viene consumato un simbolo, quindi la computazione termina sempre)
- **Configurazioni di Arresto (Halting Configurations)**: Le configurazioni di arresto sono quelle in cui non ci sono più simboli da leggere, quindi nella forma $\langle q, \varepsilon \rangle$. Un DFA non si blocca mai prima di aver letto tutto l'input, dato che la sua funzione di transizione δ è totale.
- **Direttive di Input/Output**: Queste definiscono come l'input viene processato e come viene determinato il risultato finale:
  - **Input ($f_{IN}(u)$)**: data la stringa di input u, la configurazione iniziale è $f_{IN}(u) = \langle q_0, u \rangle$

- **Output** ($f_{OUT}(\langle q, u\rangle)$): il risultato di una computazione viene estratto dalla configurazione finale $\langle q, u\rangle$ nel modo seguente:

$$f_{OUT}(\langle q, u\rangle) = \begin{cases} \text{True} & q \in F, u = \varepsilon \Rightarrow \text{ tutta la stringa è stata letta} \\ \text{False} & \text{altrimenti} \end{cases}$$

- **Linguaggio accettato**: Il linguaggio $L(\mathcal{M})$ riconosciuto da un DFA $\mathcal{M}$ è l'insieme di tutte le stringhe u tali per cui, partendo dalla configurazione iniziale $\langle q_0, u\rangle$, l'automa raggiunge una configurazione $\langle q, \varepsilon\rangle$. Formalmente:

$$L(\mathcal{M}) = \{u \mid \langle q_0, u\rangle \to^* \langle q, \varepsilon\rangle, \text{ per qualche } q \in F\}$$

Esiste un modo equivalente per definire il linguaggio accettato, che utilizza una funzione di transizione estesa $\hat{\delta} : Q \times \Sigma^* \to Q$:

  - $\hat{\delta}(q, \varepsilon) = q$ (leggendo la stringa vuota, si rimane nello stato corrente)
  - $\hat{\delta}(q, u\sigma) = \delta(\hat{\delta}(q, u), \sigma)$ (per leggere una stringa u seguita da un simbolo σ, si calcola prima lo stato raggiunto dopo aver letto u, e da quello stato si applica la funzione δ per leggere σ)

Una stringa u è accettata se $\hat{\delta}(q_0, u) \in F$. Il linguaggio accettato (riconosciuto) da $\mathcal{M}$ è quindi:

$$L(\mathcal{M}) = \left\{u \mid \hat{\delta}(q_0, u) \in F\right\}$$

I **linguaggi regolari** sono quelli accettati da qualche DFA.

Esercizio

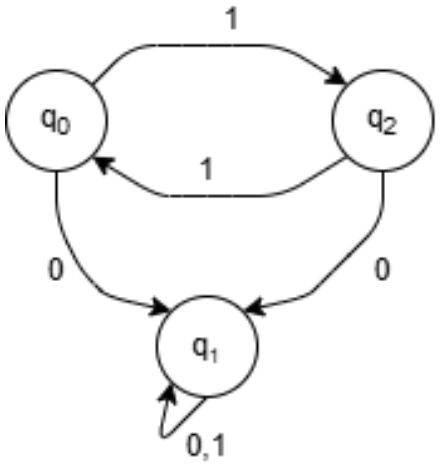
Proviamo che $\emptyset, \{\varepsilon\}$ *e* Σ^* *sono insiemi regolari.*

- $\emptyset$: Per essere accettato abbiamo bisogno che $\mathcal{M} = \langle Q, \Sigma, \delta, q_0, F = \emptyset\rangle$ perchè $\emptyset \in F$ e perchè non viene accettato da quelle $\mathcal{M} \mid F \neq \emptyset$
- $\{\varepsilon\}$: $F = \{q_0\}$ per definizione, quindi:
  - $Q = \{q_0, q_1\}$
  - Σ qualunque ad esempio $\{a\}$
  - δ
    - $\delta(q_0, a) = q_1$
    - $\delta(q_1, a) = q_1$

  Quindi $\{\varepsilon\}$ è accettato perchè la DFA ha come solo stato finale q_0 e se dessimo in input la stringa "aa" porterebbe allo stato q_1 non finale e rimarrebbe bloccato lì ma $q_1 \notin F$ quindi viene rifiutata.
- Σ^*: Possiamo costruire una $\mathcal{M}$ molto semplice
  - $Q = \{q_0\}$
  - $F = \{q_0\}$
  - δ
    - $\delta(q_0, \sigma) = q_0$

  In questo modo qualsiasi sia l'alfabeto, l'automa consuma l'input e quando la stringa è vuota viene accettata.

Un DFA può essere rappresentato come un grafo orientato etichettato detto **grafo di transizione**



Oppure dando una matrice di transizione

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

Dove lo stato iniziale è indicato con $\rightarrow$ e lo stato finale con $*$.

Definizione 2.1.2: Automa a stato finito non deterministico (NFA)

E' una quintupla $\mathcal{M} = \langle Q, \Sigma, \delta, q_0, F \rangle$ dove $\delta : Q \times \Sigma \rightarrow \wp(Q)$.
$\wp(Q)$ mappa un **insieme di stati**, cioè l'insieme di tutti i sottoinsiemi possibili di Q. Ad esempio se abbiamo $Q = \{q_0, q_1, q_2\}$, $\wp(Q) = \{\{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \ldots, \{q_0, q_1, q_2\}\}$

Analogamente ai DFA, un NFA può essere rappresentato in due modi principali:

- **Grafo di transizione**: rimane concettualmente la stessa dei DFA
- **Tabella di transizione**: ogni casella della tabella (corrispondente a una coppia stato/simbolo) può contenere un insieme di stati

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

Il comportamento di un NFA è descritto da un sistema di transizione:

- **Configurazioni**: Sono della forma $\langle q, u \rangle$, dove $q \in Q$ è lo stato corrente e $u \in \Sigma^*$ è la stringa ancora da leggere
- **Relazione di riduzione**: È definita come: $\langle q, \sigma u \rangle \rightarrow \langle q', u \rangle$ se $q' \in \delta(q, \sigma)$. **Non è deterministica** ma comunque **terminante**, poichè a ogni passo viene consumato un simbolo di input (a meno che non si blocchi)
- **Configurazioni di arresto**: Un NFA può bloccarsi in due modi:
  1. Quando ha letto tutta la stringa: $\langle q, \varepsilon \rangle$
  2. Quando si trova in una configurazione $\langle q, \sigma u \rangle$ ma non esistono transizioni possibili per il simbolo σ dallo stato q (cioè $\delta(q, \sigma) = \emptyset$)

- **Direttive di Input/Output**: Le direttive di input/output sono simili a quelle dei DFA:
  - **Input**: $f_{IN}(u) = \langle q_0, u \rangle$
  - **Output**: $f_{OUT} = \begin{cases} \langle q, u \rangle = \text{True} & \text{se } q \in F, u = \varepsilon \\ \text{False} & \text{altrimenti} \end{cases}$
- **Linguaggio Accettato**: Il linguaggio $L(\mathcal{M})$ accettato (o riconosciuto) da un NFA $\mathcal{M}$ è l'insieme delle stringhe u per cui esiste almeno una computazione che, partendo dalla configurazione iniziale $\langle q_0, u \rangle$, raggiunge una configurazione $\langle q, \varepsilon \rangle$ dove q è uno stato finale. Questo è un punto chiave del non determinismo: basta una computazione accettante tra tutte quelle possibili.
  Un modo equivalente per definire il linguaggio accettato usa la funzione estesa $\hat{\delta} : Q \times \Sigma^* \to \wp(Q)$:
  - $\hat{\delta}(q, \varepsilon) = q$
  - $\hat{\delta}(q, u\sigma) = \bigcup_{q' \in \hat{\delta}(q,u)} \delta(q', \sigma)$

  Con questa definizione, una stringa u è accettata se e solo se esiste uno stato q nell'insieme $\hat{\delta}(q_0, u)$ che sia uno stato finale ($q \in F$). In altre parole, $L(\mathcal{M}) = u \mid \hat{\delta}(q_0, u) \cap F \neq \emptyset$

Teorema 2.1.1: Rabin-Scott

Sia $\mathcal{M} = \langle Q, \Sigma, \delta_N, q_0, F_N \rangle$ un NFA. Allora esiste un DFA $\mathcal{M}_D$ tale che $L(\mathcal{M}_D) = L(\mathcal{M})$

Prova: Costruiamo $\mathcal{M}_D$ come la quintupla $\langle \wp(Q), \Sigma, \delta_D, \{q_0\}, F_D \rangle$. L'idea centrale della costruzione è che ogni stato del nuovo DFA $\mathcal{M}_D$ corrisponde a un insieme di stati dell'NFA originale.

- $Q_D = \wp(Q_N)$: questo significa che ogni stato del DFA è un sottoinsieme degli stati dell'NFA. Se l'NFA ha $|Q_N|$ stati, il DFA risultante può avere fino a $2^{|Q_N|}$ stati, sebbene non tutti siano necessariamente raggiungibili.
- δ_D: per ogni stato $q_D \in \wp(Q_N)$ (che è un insieme di stati dell'NFA) e per ogni simbolo $\sigma \in \Sigma$, la transizione $\delta_D(q_D, \sigma)$ porta a un nuovo stato del DFA che è l'unione di tutti gli stati raggiungibili nell'NFA da qualsiasi stato in q_D leggendo σ. Formalmente:

$$\delta_D(q_D, \sigma) = \bigcup_{q \in q_D} \delta_N(q, \sigma)$$

- F_D: Uno stato $q_D \subseteq Q_N$ del DFA è uno stato finale se e solo se contiene almeno uno stato finale dell'NFA. Formalmente:

$$F_D = q_D \subseteq Q_N \mid q_D \cap F_N \neq \emptyset$$

Esempio

- **NFA Originale** ($\mathcal{M}_N$): $\mathcal{M} = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta_N, q_0, q_2)$ con

$$\delta_N = \begin{array}{|c|c|c|} \hline & 0 & 1 \\ \hline \to q_0 & q_0 & \{q_0, q_1\} \\ \hline q_1 & \emptyset & q_2 \\ \hline *q_2 & q_2 & q_2 \\ \hline \end{array}$$

- **DFA Derivante** ($\mathcal{M}_D$): $\mathcal{M}_D = (Q_D, \{0, 1\}, \delta_D, q_0, F_D)$
  - $Q_D = \{\emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$
  - $F_D = \{\{q_2\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$ (tutti gli stati che contengono lo stato finale della NFA)
  - δ_D:
    1. **Da** $\{q_0\}$:
       - $\delta_D(q_0, 0) = \delta_N(q_0, 0) = q_0$
       - $\delta_D(q_0, 1) = \delta_N(q_0, 1) = \{q_0, q_1\}$
    2. **Da** $\{q_0, q_1\}$:
       - $\delta_D(\{q_0, q_1\}, 0) = \delta_N(q_0, 0) \cup \delta_N(q_1, 0) = \{q_0\} \cup \emptyset = q_0$
       - $\delta_D(\{q_0, q_1\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_1, 1) = \{q_0, q_1\} \cup \{q_2\} = \{q_0, q_1, q_2\}$
    3. **Da** $\{q_0, q_1, q_2\}$:
       - $\delta_D(\{q_0, q_1, q_2\}, 0) = \delta_N(q_0, 0) \cup \delta_N(q_1, 0) \cup \delta_N(q_2, 0) = \{q_0\} \cup \emptyset \cup \{q_2\} = \{q_0, q_2\}$
       - $\delta_D(\{q_0, q_1, q_2\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_1, 1) \cup \delta_N(q_2, 1) = \{q_0, q_1\} \cup \{q_2\} \cup \{q_2\} = \{q_0, q_1, q_2\}$
    4. **Da** $\{q_0, q_2\}$:
       - $\delta_D(\{q_0, q_2\}, 0) = \delta_N(q_0, 0) \cup \delta_N(q_2, 0) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
       - $\delta_D(\{q_0, q_2\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_2, 1) = \{q_0, q_1\} \cup \{q_2\} = \{q_0, q_1, q_2\}$

    Quindi

$$\delta_D = \begin{array}{|c|c|c|} \hline & 0 & 1 \\ \hline \to q_0 & q_0 & \{q_0, q_1\} \\ \hline \{q_0, q_1\} & q_0 & \{q_0, q_1, q_2\} \\ \hline *\{q_0, q_1, q_2\} & \{q_0, q_2\} & \{q_0, q_1, q_2\} \\ \hline *\{q_0, q_2\} & \{q_0, q_2\} & \{q_0, q_1, q_2\} \\ \hline \end{array}$$

Abbiamo costruito la tabella includendo unicamente gli stati raggiungibili a partire da $\{q_0\}$. Secondo la definizione di δ_D, non è necessario calcolare le transizioni per stati come $\{q_1\}$, $\{q_1, q_2\}$ e $\{q_2\}$, dato che non vengono mai raggiunti.

Dimostrazione

Dimostrazione per induzione

Base

- Per la definizione della funzione di transizione estesa in un DFA, $\hat{\delta}_D(q_0, \varepsilon) = q_0$
- Per la definizione della funzione di transizione estesa in un NFA, $\hat{\delta}_N(q_0, \varepsilon) = q_0$

Quindi, la condizione è soddisfatta per la stringa vuota: $\hat{\delta}_D(q_0, \varepsilon) = \hat{\delta}_N(q_0, \varepsilon)$.

Passo (Stringa $u\sigma$)

- Assumiamo, per ipotesi induttiva, che per una generica stringa u, valga $\hat{\delta}_D(q_0, u) = \hat{\delta}_N(q_0, u)$, vogliamo dimostrare che la stessa uguaglianza vale per la stringa $u\sigma$
- Applichiamo la definizione di $\hat{\delta}_D$ a $u\sigma$: $\hat{\delta}_D(q_0, u\sigma) = \delta_D(\hat{\delta}_D(q_0, u), \sigma)$, possiamo sostituire $\hat{\delta}_D(q_0, u)$ con $\hat{\delta}_N(q_0, u)$: $= \delta_D(\hat{\delta}_N(q_0, u), \sigma)$
- Ora applichiamo la definizione di δ_D: è l'unione delle transizioni $\delta_N(q', \sigma)$ per tutti gli stati q' nell'insieme $\hat{\delta}_N(q_0, u) := \bigcup_{q' \in \hat{\delta}_N(q_0, u)} \delta_N(q', \sigma)$. Questa è precisamente la definizione di $\hat{\delta}_N(q_0, u\sigma)$

Quindi, $\hat{\delta}_D(q_0, u\sigma) = \hat{\delta}_N(q_0, u\sigma)$.
Avendo dimostrato per induzione che $\hat{\delta}_D(q_0, u) = \hat{\delta}_N(q_0, u)$ per ogni stringa $u \in \Sigma^*$, possiamo concludere sull'accettazione del linguaggio:

- Una stringa u è accettata dal DFA $\mathcal{M}_D$ se e solo se $\hat{\delta}_D(q_0, u) \in F_D$
- Per definizione di F_D, ciò significa che $\hat{\delta}_D(q_0, u)$ deve contenere almeno uno stato finale di $\mathcal{M}_N$, ovvero $\hat{\delta}_D(q_0, u) \cap F_N \neq \emptyset$
- Poiché $\hat{\delta}_D(q_0, u) = \hat{\delta}_N(q_0, u)$, questo è equivalente a $\hat{\delta}_N(q_0, u) \cap F_N \neq \emptyset$
- E quest'ultima è la condizione di accettazione di una stringa per un NFA $\mathcal{M}_N$

Pertanto, u è accettata da $\mathcal{M}_D$ se e solo se è accettata da $\mathcal{M}_N$, il che significa che $L(\mathcal{M}_D) = L(\mathcal{M}_N)$.

Definizione 2.1.3: ε-NFA o NFA con transizioni silenti

E' una quintupla $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ dove la funzione di transizione, che ha la forma

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \wp(Q)$$

La peculiarità di questa definizione è l'inclusione di ε nell'alfabeto di input della funzione di transizione, il che consente transizioni tra stati senza consumare alcun simbolo di input (transizioni silenti). Questo permette, in molti casi, di costruire automi più semplici e leggibili

Il comportamento di un ε-NFA è descritto da un sistema di transizione con le seguenti caratteristiche:

- **Configurazioni**: Sono della forma $\langle q, u \rangle$, dove $q \in Q$ è lo stato corrente e $u \in \Sigma^*$ è la stringa ancora da leggere
- **Relazione di riduzione**: È definita in due modi:
  - $\langle q, \sigma u \rangle \to \langle q', u \rangle$ se $q' \in \delta(q, \sigma)$ (transizione standard, consumando un simbolo)
  - $\langle q, u \rangle \to \langle q', u \rangle$ se $q' \in \delta(q, \varepsilon)$ (transizione silente, senza consumare simboli)

  La relazione di riduzione è non deterministica, anche per via della possibilità di scegliere tra leggere o non leggere un simbolo. È generalmente non terminante, ma le computazioni infinite sono dovute solo a cicli di transizioni ε dallo stesso stato, che possono essere eliminati.

- **Configurazione di arresto**: Oltre alle configurazioni della forma $\langle q, \varepsilon \rangle$, includono anche quelle in cui l'automa si blocca perché non ci sono transizioni possibili (né con simboli di input, né silenti) da un dato stato con il simbolo corrente
- **Linguaggio accettato**: Le direttive di input/output sono le stesse degli NFA. Una stringa u è accettata se esiste almeno una computazione che, partendo dalla configurazione iniziale $\langle q_0, u \rangle$, raggiunge una configurazione $\langle q, \varepsilon \rangle$ dove q è uno stato finale. Un modo equivalente per definire il linguaggio accettato utilizza la funzione di transizione estesa $\hat{\delta} : Q \times \Sigma^* \to \wp(Q)$, che si basa sul concetto di ε-closure:
  - $\varepsilon - closure(q)$: È l'insieme di tutti gli stati raggiungibili da q utilizzando zero o più transizioni ε. Formalmente, include q stesso, tutti gli stati raggiungibili da q tramite $\delta(q, \varepsilon)$, e ricorsivamente tutti gli stati raggiungibili da questi ultimi tramite ulteriori transizioni ε
  - **Funzione di transizione estesa** $\hat{\delta}$:
    * $\hat{\delta}(q, \varepsilon) = \varepsilon\text{-closure}(q)$
    * $\hat{\delta}(q, u\sigma) = \bigcup_{q' \in \hat{\delta}(q,u)} \varepsilon\text{-closure}(\delta(q', \sigma))$

    Una stringa u è accettata se e solo se l'insieme di stati raggiungibili dopo aver letto u (cioè $\hat{\delta}(q_0, u)$) contiene almeno uno stato finale ($\hat{\delta}(q_0, u) \cap F \neq \emptyset$)

Esempio

$\mathcal{M}_N = (\{q_0, q_1, q_2\}, \{a, b\}, \delta_N, q_0, F_N)$

$\delta_N =$

	a	b	ε
$\to q_0$	...	...	q_1
q_1	...	...	q_2
$*q_2$	...	...	q_1

Calcolo le $\varepsilon - closure$:

- $\varepsilon - closure(q_0) = \{q_0, q_1, q_2\}$
- $\varepsilon - closure(q_1) = \{q_1, q_2\}$
- $\varepsilon - closure(q_2) = \{q_2, q_1\}$

Algoritmo

Per trovare un NFA senza transizioni ε equivalente:

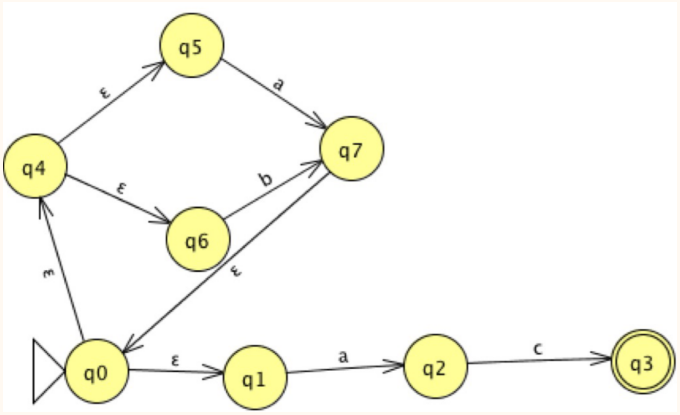
1. Tabelliamo δ (le funzioni di transizione) dell'automa originale
2. Scriviamo il nuovo automa nella forma $\mathcal{M}' = \langle Q', \Sigma, \delta', q_0, F' \rangle$ dove:
   - $Q' = Q$ (solitamente)
   - Σ è l'alfabeto
   - δ' sarà la nuova tabella le cui righe si troveranno facendo:

$$\delta'(q, \sigma) = \bigcup_{p \in \varepsilon\text{-closure}(q)} \bigcup_{s \in \delta(p,\sigma)} \varepsilon\text{-closure}(s)$$

   - q_0 stato iniziale (solitamente è lo stesso)
   - F' è l'insieme di tutti gli stati che hanno nella propria ε-closure uno stato finale dell'automa originale

Esercizio

Si consideri il seguente automa a stati finiti con transizioni silenti:



- *Si trasformi l'automa in un NFA eliminando le transizioni silenti*:
  - Tabelliamo δ

	a	b	c	ε
q_0	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_1, q_4\}$
q_1	$\{q_2\}$	$\emptyset$	$\emptyset$	$\emptyset$
q_2	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
q_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
q_4	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_5, q_6\}$
q_5	$\{q_7\}$	$\emptyset$	$\emptyset$	$\emptyset$
q_6	$\emptyset$	$\{q_7\}$	$\emptyset$	$\emptyset$
q_7	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_0\}$

  - Calcoliamo tutte le ε-closure
    - ε-closure$(q_0) = \{q_0, q_1, q_4, q_5, q_6\}$
    - ε-closure$(q_1) = \{q_1\}$
    - ε-closure$(q_2) = \{q_2\}$
    - ε-closure$(q_3) = \{q_3\}$
    - ε-closure$(q_4) = \{q_4, q_5, q_6\}$
    - ε-closure$(q_5) = \{q_5\}$
    - ε-closure$(q_6) = \{q_6\}$
    - ε-closure$(q_7) = \{q_7, q_0, q_1, q_4, q_5, q_6\}$
  - Costruiamo δ'

	a	b	c
q_0	$\{q_0, q_1, q_2, q_4, q_5, q_6, q_7\}$	$\{q_0, q_1, q_4, q_5, q_6, q_7\}$	$\emptyset$
q_1	$\{q_2\}$	$\emptyset$	$\emptyset$
q_2	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\emptyset$	$\emptyset$
q_4	$\{q_0, q_1, q_4, q_5, q_6, q_7\}$	$\{q_0, q_1, q_4, q_5, q_6, q_7\}$	$\emptyset$
q_5	$\{q_0, q_1, q_4, q_5, q_6, q_7\}$	$\emptyset$	$\emptyset$
q_6	$\emptyset$	$\{q_0, q_1, q_4, q_5, q_6, q_7\}$	$\emptyset$
q_7	$\{q_0, q_1, q_2, q_4, q_5, q_6, q_7\}$	$\{q_0, q_1, q_4, q_5, q_6, q_7\}$	$\emptyset$

- *Si dia un'espressione regolare che denota il linguaggio accettato*: $(a + b)^*ac$

2.2 Minimizzazione di DFA

La minimizzazione di un DFA mira a trovare un automa equivalente, che accetti lo stesso linguaggio, ma con il numero minimo di stati.

- **Stati indistinguibili ($q \sim q'$)**: sono considerati indistinguibili (o equivalenti) se il linguaggio accettato a partire da q è identico al linguaggio accettato a partire da q'. Ciò significa che per qualsiasi stringa $u \in \Sigma^*$, se $\hat{\delta}(q, u) \in F$ allora anche $\hat{\delta}(q', u)$ (deve) $\in F$ e viceversa. Questa relazione di indistinguibilità è una relazione di equivalenza.
- **Classe di equivalenza ($[q]_\sim$)**: è l'insieme di tutti gli elementi di Q che sono in relazione con q ($\forall x \in Q, x \sim q$)
- **Quoziente di Q ($Q/_\sim$)**: è l'insieme i cui elementi sono le classi di equivalenza stesse, $Q/_\sim = \{[q]_\sim \mid q \in Q\}$

Due stati indistinguibili possono intuitivamente essere trasformati in un unico stato.
Quindi:

$$\mathcal{M}' = \langle Q/_\sim, \Sigma, \delta', [q_0]_\sim, F' \rangle$$

2.2.1 Algoritmo

1. Suddividiamo gli stati in due classi di equivalenza: quelli finali e quelli non finali
2. A ogni passo consideriamo le classi di equivalenza ottenute al passo precedente
   (a) Per ognuna di esse immaginiamo che tutti gli stati all'interno leggano un σ
   (b) Se tutti gli stati finiscono in altri stati della stessa classe di equivalenza, allora questi stati sono ancora considerati simili per quel σ
   (c) Altrimenti quei due stati non sono più indistinguibili. A questo punto la classe di equivalenza viene suddivisa in sottogruppi più piccoli, separando gli stati che si comportano in modo diverso
3. Torniamo al punto (2.) e ripetiamo il processo, l'algoritmo si ferma solo quando, dopo aver controllato tutti i grupi con tutti i simboli, non c'è più nulla da dividere

```
ind = {insieme di (q,q') indistinguibili};
old_ind = null;
while(ind != old_ind) {
    old_ind = ind;
    ind = null;
    for each (q,q') in old_ind {
        ok = true;
        for simb in alfabeto {
            if(!indistinguibili(delta(q,simb),delta(q',simb))){
                ok = false;
                break;
            }
        }
        if (ok) {
            ind = ind + (q,q');
        }
    }
}
```

Esempio

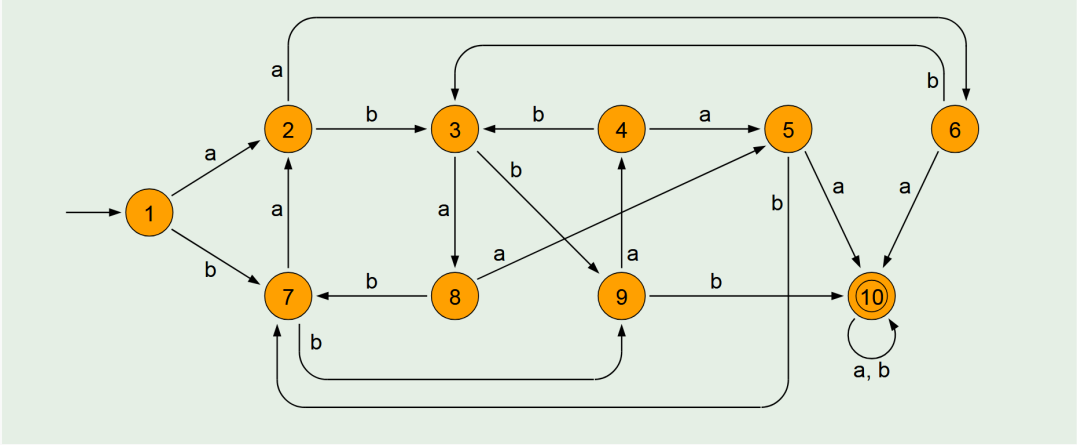
Consideriamo il seguente DFA:

	a	**b**
$\rightarrow q_0$	q_0	q_1
q_1	q_1	q_2
q_2	q_2	q_3
$*q_3$	q_3	q_3

1. $\sim = \{\{q_3\}, \{q_0, q_1, q_2\}\}$
2. **Prima iterazione**:
   (a) Consideriamo $\{q_0, q_1, q_2\}$
   (b) Leggiamo 'a':
      i. $\delta(q_0, a) = q_0 \in \{q_0, q_1, q_2\}$
      ii. $\delta(q_1, a) = q_1 \in \{q_0, q_1, q_2\}$
      iii. $\delta(q_2, a) = q_2 \in \{q_0, q_1, q_2\}$
   (c) Leggiamo 'b':
      i. $\delta(q_0, b) = q_1 \in \{q_0, q_1, q_2\}$
      ii. $\delta(q_1, b) = q_2 \in \{q_0, q_1, q_2\}$
      iii. $\delta(q_2, b) = q_3 \notin \{q_0, q_1, q_2\}$
   (d) Quindi $\sim = \{\{q_3\}, \{q_0, q_1\}, \{q_2\}\}$
3. **Seconda iterazione**:
   (a) Consideriamo $\{q_0, q_1\}$
   (b) Leggiamo 'a':
      i. $\delta(q_0, a) = q_0 \in \{q_0, q_1\}$
      ii. $\delta(q_1, a) = q_1 \in \{q_0, q_1\}$
   (c) Leggiamo 'b':
      i. $\delta(q_0, b) = q_1 \in \{q_0, q_1\}$
      ii. $\delta(q_1, b) = q_2 \notin \{q_0, q_1\}$
   (d) Quindi $\sim = \{\{q_3\}, \{q_0\}, \{q_1\}, \{q_2\}\}$

Quindi il DFA era già minimo.

Esempio



1. Costruiamo la tabella di transizione (non necessaria ma aiuta):

	a	**b**
$\rightarrow 1$	2	7
2	6	3
7	2	9
6	10	3
3	8	9
9	4	10
$*10$	10	10
8	5	7
4	5	3
5	10	7

2. $\sim = \{\{10\}, \{1,2,3,4,5,6,7,8,9\}\}$
3. **Prima iterazione**:
   (a) Consideriamo $\{1,2,3,4,5,6,7,8,9\}$
   (b) Leggiamo 'a':
      i. $\delta(1,a) = 2$
      ii. $\delta(2,a) = 6$
      iii. $\delta(3,a) = 8$
      iv. $\delta(4,a) = 5$
      v. $\delta(5,a) = 10 \notin \{1,2,3,4,5,6,7,8,9\}$
      vi. $\delta(6,a) = 10 \notin \{1,2,3,4,5,6,7,8,9\}$
      vii. $\delta(7,a) = 2$
      viii. $\delta(8,a) = 5$
      ix. $\delta(9,a) = 4$
   (c) Leggiamo 'b':
      i. $\delta(1,b) = 7$
      ii. $\delta(2,b) = 3$
      iii. $\delta(3,b) = 9$
      iv. $\delta(4,b) = 3$
      v. $\delta(7,b) = 9$
      vi. $\delta(8,b) = 7$
      vii. $\delta(9,b) = 10 \notin \{1,2,3,4,5,6,7,8,9\}$
   (d) Quindi $\sim = \{\{10\}, \{5,6\}, \{9\}, \{1,2,3,4,7,8\}\}$
4. Nella seconda (e ultima in questo caso) iterazione avremo che $\sim = \{\{10\}, \{5,6\}, \{9\}, \{1\}, \{2,4,8\}, \{3,7\}\}$

2.2.2 Espressioni regolari

Definizione 2.2.1

Un'espressione regolare è una stringa che descrive schematicamente un insieme di stringhe. I linguaggi denotati dalle RE sono definiti induttivamente:

- $\emptyset$ è una RE che denota il linguaggio vuoto
- ε è una RE che denota il linguaggio contenente solo la stringa vuota $\{\varepsilon\}$
- Per ogni simbolo $\sigma \in \Sigma$, σ è una RE che denota il linguaggio $\{\sigma\}$
- Se r_1 è un'espressione regolare che denota il linguaggio L_1 e r_2 denota L_2, allora:
  - $r_1 + r_2$ (o $r_1 \mid r_2$) denota l'unione dei linguaggi $L_1 \cup L_2$
  - $r_1 r_2$ denota la concatenazione dei linguaggi $L_1 \cdot L_2$
- Se r è un'espressione regolare che denota L, r^* denota la chiusura di Kleene L^*

2.3 Proprietà dei linguaggi regolari

- **Unione**: Se L e L' sono linguaggi regolari, anche la loro unione $(L \cup L')$ è un linguaggio regolare. Questa proprietà è "ovvia per definizione" in relazione alle espressioni regolari.
- **Concatenazione**: Se L e L' sono linguaggi regolari, anche la loro concatenazione $(L \cdot L')$ è un linguaggio regolare. Anche questa è "ovvia per definizione".
- **Chiusura di Kleene**: Se L è un linguaggio regolare, anche la sua chiusura di Kleene (L^*) è un linguaggio regolare. Questa proprietà è anch'essa "ovvia per definizione".
- **Complementazione**: Se L è un linguaggio regolare su un alfabeto Σ, allora anche il suo complementare $(\overline{L} = \Sigma^* \setminus L)$ è regolare. Questo si dimostra prendendo un DFA che riconosce L e scambiando gli stati finali con quelli non finali.
- **Intersezione**: Se L e L' sono linguaggi regolari, allora anche la loro intersezione $(L \cap L')$ è un linguaggio regolare. La chiusura rispetto all'intersezione si deduce facilmente dalla chiusura per complementazione, dato che $L \cap L' = \overline{\overline{L} \cup \overline{L'}}$.

2.3.1 Pumping Lemma

Teorema 2.3.1: Pumping Lemma

Sia L un linguaggio regolare. Allora esiste una costante $n \in \mathbb{N}$ (detta "lunghezza di pompaggio" o "numero di stati del DFA") tale che, per ogni $z \in L$ con $|z| \geq n$, possiamo decomporre z come uvw in modo che:

- $|uv| \leq n$ (la porzione "pompabile" v deve trovarsi entro i primi n simboli della stringa)
- $|v| > 0$ (la porzione "pompabile" non deve essere vuota)
- per ogni $i \geq 0$ si ha che $uv^i w \in L$ (v può essere ripetuta i volte, inclusa l'eliminazione $i = 0$, e la stringa rimarrà nel linguaggio)

Dimostrazione

- Assumiamo L regolare $\Rightarrow$ per il Pumping Lemma, esiste una costante n.
- Si sceglie una stringa $z = \sigma_1\sigma_2\ldots\sigma_{|z|} \in L$ la cui lunghezza $|z| \geq n$.
- Dato che la stringa è accettata dall'automaa, abbiamo $q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_{|z|}} q_{|z|} = q_F \in F$
- Dato che il DFA ha solo n stati in totale, necessariamente si ha che almeno uno stato $\bar{q}$ viene attraversato due volte nei primi n passi, ossia $\bar{q} = q_i = q_j$ per qualche $0 \leq i < j \leq n$
- **Decomposizione di z**:
  - $u = \sigma_1\ldots\sigma_i$
  - $v = \sigma_{i+1}\ldots\sigma_j$
  - $w = \sigma_{j+1}\ldots\sigma_{|z|}$

  si ha $q_0 \xrightarrow{u} \bar{q} \xrightarrow{v} \bar{q} \xrightarrow{w} q_F$, quindi $q_0 \xrightarrow{u} \bar{q} \xrightarrow{v^i} \bar{q} \xrightarrow{w} q_F$ perchè $\bar{q} \xrightarrow{v} \bar{q}$

Il pumping lemma viene tipicamente utilizzato per provare che un linguaggio L **non** è regolare. A tale scopo:

- Si sceglie un numero arbitrario n
- Si trova una stringa $z \in L$, con lunghezza $|z| \geq n$
- Si dimostra che, per ogni modo in cui si può dividere z in tre parti uvw (dove $|uv| \leq n$ e $|v| > 0$), se si ripete la parte v (o la si elimina), la stringa che si ottiene **non** appartiene più al linguaggio

Questo contraddice la definizione di linguaggio regolare, provando così che il linguaggio non lo è.

Esempio

Possiamo dimostrare, utilizzando il pumping lemma, che il linguaggio $\{0^n1^n\}$ non è regolare. Infatti, preso n arbitrario, consideriamo la stringa 0^n1^n. Decomponendo tale stringa in tre parti u, v, w tali che la lunghezza delle prime due sia $\leq n$ e la seconda sia non vuota, si ha chiaramente $u = 0^a$, $v = 0^b$ e $w = 0^c1^n$ con $a + b + c = n$, $b > 0$. Allora, per esempio per $i = 0$, si ha che $uv^0w = 0^a0^c1^n$ non appartiene al linguaggio in quanto $a + c < n$.

2.4 Automi a pila

Definizione 2.4.1: Automa a pila non deterministico (PDA)

E' una tupla

$$\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, Z, F\rangle$$

- Q, Σ, q_0, F come per gli automi a stati finiti
- Γ: alfabeto della pila
- Z: simbolo inziale della pila
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \wp_F(Q \times \Gamma^*)$, dove $\wp_F(X)$ denota l'insieme dei sottoinsiemi *finiti* di X

In termini informali, quando un automa a pila si trova nello stato q, legge un simbolo σ (o esegue una transizione silente) e rimuove il simbolo in cima Z dalla pila, può passare a un nuovo stato q' e inserire una sequenza α (anche vuota) di nuovi simboli in cima alla pila.

L'automa è **non deterministico**, il che significa che per la stessa combinazione di stato, simbolo di input corrente e simbolo in cima alla pila, possono esserci diverse transizioni possibili. Il comportamento di un PDA è descritto da un sistema di transizione con le seguenti caratteristiche:

- **Configurazioni**: sono nella forma $\langle q, u, \alpha \rangle$ dove $q \in Q$ è lo stato corrente, $u \in \Sigma^*$ è la stringa ancora da leggere e $\alpha \in \Gamma^*$ è il contenuto corrente della pila
- **Relazione di riduzione**: Descrive come la macchina si muove da una configurazione all'altra:
  - $\langle q, \sigma u, X\alpha \rangle \to \langle q', u, \gamma\alpha \rangle$ se $\langle q', \gamma \rangle \in \delta\langle q, \sigma, X \rangle$ (lettura di un simbolo di input σ)
  - $\langle q, u, X\alpha \rangle \to \langle q', u, \gamma\alpha \rangle$ se $\langle q', \gamma \rangle \in \delta\langle q, \varepsilon, X \rangle$ (transizione silente ε)

  La relazione di riduzione è non deterministica. Ciò significa che per una data configurazione, potrebbero esserci più transizioni possibili, incluse le transizioni silenti. Le computazioni possono essere non terminanti se ci sono cicli di transizioni ε che non modificano l'input o la pila in modo significativo per il consumo della stringa
- **Configurazioni di arresto**: Si verificano quando non ci sono ulteriori mosse possibili
- **Direttive di Input/Output**:
  - **Input**: $f_{IN}(u) = \langle q_0, u, Z \rangle$
  - **Output**:
    - Riconoscimento per pila vuota (Empty Stack Acceptance):

$$f_{OUT}(\langle q, u, \alpha \rangle) = \begin{cases} \text{True} & \text{se } u = \varepsilon \text{ e } \alpha = \varepsilon \\ \text{False} & \text{altrimenti} \end{cases}$$

    - Riconoscimento per stati finali (Final State Acceptance):

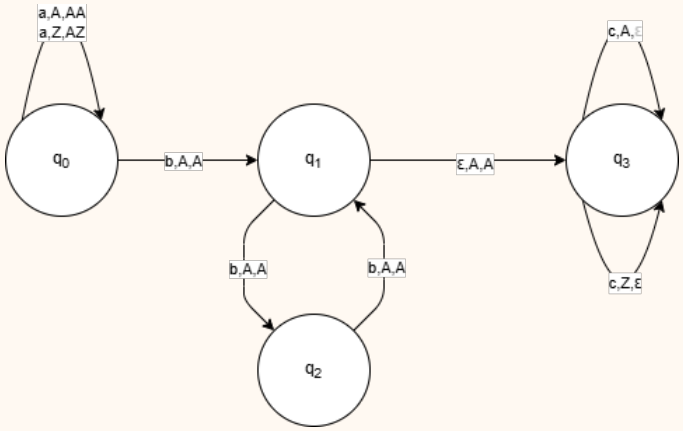
$$f_{OUT}(\langle q, u, \alpha \rangle) = \begin{cases} \text{True} & \text{se } u = \varepsilon \text{ e } q \in F \\ \text{False} & \text{altrimenti} \end{cases}$$

- **Linguaggio accettato**:
  - Per pila vuota: $L(\mathcal{M}) = \{u \in \Sigma^* \mid \langle q_0, u, Z \rangle \to^* \langle \_, \varepsilon, \varepsilon \rangle\}$
  - Per stati finali: $L(\mathcal{M}) = \{u \in \Sigma^* \mid \langle q_0, u, Z \rangle \to^* \langle q, \varepsilon, \_ \rangle, q \in F\}$

  Usiamo la wildcard _ per sottolineare il fatto che lo stato non è rilevante.
  Le due definizioni sono equivalenti: se il linguaggio L è riconosciuto da un qualche PDA secondo la prima definizione, allora esiste anche un PDA che lo riconosce secondo l'altra e viceversa.

Esercizio

Si consideri il linguaggio $L = \{a^n b^m c^n \mid m \text{ dispari}\}$. *Si dia un'automa a pila che riconosce il linguaggio, spiegando su quale idea intuitiva è basato.* L'automa a pila quindi deve accettare stringhe, ad esempio, nel formato: aabcc



L'idea è quella di salvarci soltanto quante a ci sono nella stringa perchè il numero di c è uguale a quello delle a. Le b invece ci importa solo che siano dispari e per fare ciò dobbiamo obbligare l'automa a prendere (oltre alla b di cambio stato che quindi siamo a 1) due b "alla volta" in questo modo avremo sempre b dispari. Per le c invece eseguiamo il pop dalla pila di tutte le a e in questo modo ne consumeremo il numero giusto e poi svuotiamo completamente la pila quando leggiamo l'ultima c.

Definizione 2.4.2: Automa a pila deterministico (DPDA)

E' una tupla $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, Z, F \rangle$ dove:

- Q, Σ, q_0 e F sono come per gli automi a stati finiti
- Γ è l'alfabeto della pila
- Z è il simbolo iniziale nella pila
- $\delta : Q \times (\Sigma \cup \varepsilon) \times \Gamma \to \mathcal{P}_F(Q \times \Gamma^*)$ soddisfa la seguente condizione di determinismo:
  Per ogni $\langle q, \sigma, X \rangle \in Q \times \Sigma \times \Gamma$:

$$|\delta\langle q, \sigma, X\rangle| + |\delta\langle q, \varepsilon, X\rangle| \leq 1$$

Questa condizione garantisce che per ogni stato corrente q, simbolo di input corrente σ (o transizione silente ε), e simbolo in cima alla pila X, esista al più una transizione possibile. Non è possibile, ad esempio, leggere un simbolo di input e contemporaneamente avere una transizione silente dallo stesso stato con lo stesso simbolo in cima alla pila.

2.5 Linguaggi Context-Free

2.5.1 Grammatiche Context-Free

Definizione 2.5.1: Grammatica CF

Una CFG è definita da una quadrupla $\langle T, N, P, S \rangle$:

- T è l'alfabeto dei terminali (i simboli che compongono le stringhe del linguaggio)
- N è l'alfabeto dei non terminali (simboli intermedi usati nelle derivazioni), con $T \cap N = \emptyset$
- P è un insieme finito di produzioni (o regole), della forma $A ::= \alpha$, dove $A \in N$ e $\alpha \in (T \cup N)^*$
- $S \in N$ è il simbolo iniziale (o assioma), da cui partono le derivazioni

Esempio

- $T = \{0, 1, \ldots, 9\}$
- $N = \{Exp, Num\}$
- $P = \{Exp := Num, Exp := (Exp + Exp), Num := T\}$
- $S = Exp$

Definizione 2.5.2: Derivazione in un passo

Questa nozione è fondamentale per comprendere come una grammatica context-free (CFG) genera le stringhe di un linguaggio.
Si considerano una grammatica $G = \langle T, N, P, S \rangle$. Si prendono due stringhe, α e β, che possono contenere sia simboli terminali che non terminali (ovvero $\alpha, \beta \in (T \cup N)^*$).
Si dice che β è derivabile da α in un passo se valgono le seguenti condizioni:

- La stringa α ha la forma $\alpha_1 A \alpha_2$. Questo significa che α contiene un non terminale A, circondato da due (eventualmente vuote) sottostringhe α_1 e α_2, che possono essere composte da terminali e non terminali ($\alpha_1, \alpha_2 \in (T \cup N)^*$)
- La stringa β ha la forma $\alpha_1 \gamma \alpha_2$
- Esiste una produzione (regola) $A ::= \gamma$ nell'insieme P delle produzioni della grammatica. Qui A è il non terminale che viene espanso e γ è la stringa (di terminali e/o non terminali) con cui A viene sostituito

La derivazione in un passo è indicata con $\alpha \to \beta$. Questa relazione è una relazione su $(T \cup N)^*$.

Chiusura Riflessiva e transitiva ($\to^*$):

- La chiusura riflessiva e transitiva di $\to$ è indicata con $\to^*$
- Si dice che β è derivabile da α (in uno o più passi, o anche zero passi se $\alpha = \beta$) se $\alpha \to^* \beta$
- Questa significa che si può ottenere β partendo da α attraverso una sequenza di zero o più applicazioni delle regole di derivazione in un passo

Definizione 2.5.3: Linguaggio generato da una grammatica

Data una grammatica Context-Free (CF) $G = \langle T, N, P, S \rangle$, il linguaggio generato da G, denotato $L(G)$, è definito come l'insieme di tutte le stringhe u che appartengono all'alfabeto dei terminali T^* e che possono essere derivate dal simbolo iniziale S:

$$L(G) = \{u \in T^* \mid S \to^* u\}$$

Un linguaggio L è definito Context-Free (CF) se esiste una grammatica CF G che lo genera, ovvero tale che $L(G) = L$.
Talvolta, l'interesse può estendersi non solo al linguaggio generato dall'assioma (S), ma a una famiglia di linguaggi generati a partire da ciascun non terminale della grammatica (ad esempio, $L_A(G) = u \in T^* \mid A \to^* u$). In tal caso, la scelta di un assioma specifico non è l'unico aspetto significativo.

2.5.2 Proprietà

Definizione 2.5.4: Pumping Lemma

Se L è un linguaggio CF, esiste una costante $n \in \mathbb{N}$ (lunghezza di pompaggio) tale che, per ogni $z \in L$ con $|z| \geq n$ può essere decomposta in $uvwxy$ in modo che:

- $|vwx| \leq n$
- $|vx| > 0$
- $\forall i \geq 0$ si ha che la stringa $uv^iwx^iy \in L$

L'idea fondamentale di questo lemma è che, se un linguaggio è CF, allora ogni sua stringa che sia "sufficientemente lunga" può essere divisa in cinque parti. Due di queste parti possono essere ripetute (o eliminate) un numero qualsiasi di volte, e la stringa risultante apparterrà ancora al linguaggio.

Dimostrazione

Prova: Qualsiasi linguaggio CF (che non includa la stringa vuota ε, per semplicità) può essere generato da una grammatica CF che sia in Forma Normale di Chomsky (CNF). Una grammatica in CNF ha solo due tipi di produzioni (regole di riscrittura):

- Un non-terminale che produce due altri non-terminali (es. $A ::= BC$)
- Un non-terminale che produce un singolo simbolo terminale (es. $A ::= \sigma$)

Proviamo che, per una grammatica in CNF, se un albero di derivazione ha altezza m, allora la stringa finale che genera (la "frontiera" dell'albero) ha una lunghezza massima di $2^{(m-1)}$.
Base: Un albero di altezza 1 significa che il simbolo di partenza ha prodotto direttamente un terminale (es. $S ::= \sigma$). La stringa ha lunghezza 1. E $2^{(1-1)} = 2^0 = 1$. Quindi, il caso base funziona.
Passo: Un albero di altezza $m + 1$ significa che il simbolo di partenza ha prodotto due altri non-terminali di altezza m (es. $A ::= BC$) le cui stringhe saranno di lunghezza al più $2^{(m-1)}$. Quindi:

$$|A| = 2^{(m-1)} + 2^{(m-1)} = 2^m$$

Prendiamo m come il numero di non-terminali della grammatica CNF che genera il linguaggio L e stabiliamo $n = 2^m$. Data una stringa $z \in L \mid |z| \geq n$, l'altezza del suo albero di derivazione deve essere **almeno** $m + 1$. Visto che l'altezza dell'albero è maggiore del numero di non-terminali, sappiamo sicuramente che un non-terminale compaia due volte in un cammino dalla radice alle foglie (**Principio dei Cassetti**).

- Indichiamo con $A^{(1)}$ e $A^{(2)}$ la prima e seconda occorrenza del non-terminale ripetuto
- Supponiamo di avere $z = uvwxy$ dove:
  - $A^{(1)}$ genera la sottostringa vwx
  - $A^{(2)}$, contenuta in $A^{(1)}$, genera solo w
  - Le condizioni del lemma specificano:
    - $|vwx| \leq n$ perché abbiamo scelto la ripetizione più vicina alle foglie (cioè quella con altezza minore).
    - $|vx| > 0$ perché in una grammatica CNF non è possibile che entrambi v e x siano vuoti.

Poiché A può generare sia vwx che w, è possibile ripetere o eliminare v e x. Questo significa che le stringhe uv^iwx^iy (per $i \geq 0$) apparterranno tutte al linguaggio.

Come nel caso dei regolari, il pumping lemma viene utilizzato per provare che un linguaggio L non è CF.
Per farlo, bisogna scegliere un numero n qualunque, e poi trovare una stringa z che:

- Appartiene al linguaggio L
- Ha lunghezza almeno n, cioè $|z| \geq n$

A questo punto, mostriamo che qualunque modo si scelga di dividere z nella forma

$$z = uvwxy$$

rispettando le due condizioni:

- $|vwx| \leq n$
- $|vx| > 0$

allora esiste almeno un valore di i per cui la stringa "pompata"

$$uv^iwx^iy$$

non appartiene al linguaggio L.

Esempio

Provare che $L = \{a^nb^nc^n\}$ *non è CF.*

1. Per assurdo consideriamo L CF
2. Scegliamo la stringa "più lunga": $z = a^nb^nc^n$, la lunghezza sarà $|z| = 3n \geq n$
3. Sappiamo che z può essere decomposta come $uvwxy$
4. Applichiamo il lemma:
   (a) $|vwx| \leq n$
   (b) $|vx| > 0$
   (c) $\forall i \geq 0, uv^iwx^iy$
   (d) Per certo possiamo dire che vwx non può includere a e c perchè dovrebbe includere tutto b^n e quindi la lunghezza sarebbe in ogni caso $> n$
   (e) Impostiamo $i = 0$ (solitamente si usa $i = 0$ o $i = 2$): $uv^0wx^0y = uwy$ con $|uwy| = |u| + |w| + |y|$
   (f) $|z| = |u| + |v| + |w| + |x| + |y| \Rightarrow |uwy| = |z| - (|v| + |x|)$
   (g) Poichè $|vx| > 0$, ne consegue che $|uwy| < |z|$
5. L non è CF

Proprietà di Chiusura

I linguaggi CF risultano chiusi rispetto a:

- **Unione**
- **Concatenazione**
- **Chiusura di Kleene**

I linguaggi CF non risultano chiusi rispetto a:

- **Intersezione**
- **Complementazione**

Capitolo 3: Teoria della calcolabilità

3.1 Macchine di Turing

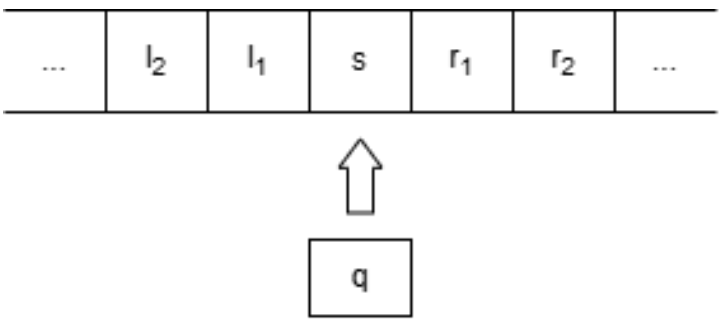
Definizione 3.1.1: Macchina di Turing deterministica

E' una tupla $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, B, F \rangle$ dove:

- Q è un insieme finito di stati
- Σ è un alfabeto di input
- Γ è un alfabeto del nastro che include l'alfabeto di input ($\Sigma \subseteq \Gamma$)
- δ è una funzione di transizione parziale della forma $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$. Questa funzione specifica l'azione successiva della macchina in base allo stato corrente e al simbolo letto sulla cella del nastro sotto la testina. L'azione consiste nel cambiare stato, scrivere un simbolo su quella cella (scrivere B equivale a cancellare) e spostare la testina a sinistra (L) o a destra (R). Se per una data combinazione di stato e simbolo non è specificata alcuna azione, la macchina si ferma.
- q_0 è lo stato iniziale
- B è un simbolo speciale che appartiene a Γ ma non a Σ ($B \in \Gamma \setminus \Sigma$). Rappresenta una cella vuota o "blank" sul nastro.
- $F \subseteq Q$ è l'insieme degli stati finali

La scelta di avere due alfabeti e un insieme di stati finali è legata ala visione di una macchina di Turing come **riconoscitore**.
Una macchina di Turing può essere immaginata come:

- Un nastro, illimitato nei due sensi e suddiviso in celle. Sul nastro è presente un numero finito di simboli diversi da B
- Una testina (di lettura e scrittura) posizionata su una cella del nastro
- La testina si trova in un certo stato $q \in Q$



Il funzionamento è determinato dalla funzione di transizione: in ogni istante, in dipendenza deterministica dello stato della testina e del simbolo letto nella cella corrente, la macchina effettua un'azione che comporta: un cambiamento di stato, la scrittura di un simbolo di Γ nella cella corrente e lo spostamento della testina a sinistra (L) o a destra (R) o rimane ferma (N).

- **Configurazioni**: sono rappresentate come $\langle \alpha, q, \beta \rangle$ dove:
  - α sono i simboli a sinistra della testina
  - q è lo stato corrente della testina
  - β sono i simboli a destra della testina

  Se β è vuota, la testina punta a una cella B. Viene rappresentata solo la porzione significativa del nastro.
- **Relazione di riduzione**: descrive come le configurazioni cambiano passo dopo passo. Per esempio, se $\delta(q, X) = \langle q', Y, R \rangle$, allora $\langle \alpha, q, X\beta \rangle \to \langle \alpha Y, q', \beta \rangle$ (spostamento a destra). Questa relazione è deterministica e, in generale, non terminante (possono esserci computazioni infinite)
- **Configurazioni di arresto**: sono quelle per cui la funzione di transizione è indefinita (cioè $\delta(q, X) \uparrow$ o $\delta(q, B) \uparrow$)
- **Direttive di Input/Output**
  - **Input**: $f_{IN}(u) = \langle \epsilon, q_0, u \rangle$
  - **Output**:
$$f_{OUT}(\langle \alpha, q, \beta \rangle) = \begin{cases} \text{True} & \text{se } q \in F \\ \text{False} & \text{altrimenti} \end{cases}$$
- **Linguaggio accettato**: è l'insieme di tutte le stringhe u che, partendo dalla configurazione iniziale, possono portare a una configurazione di accettazione.
$$L(\mathcal{M}) = \{u \in \Sigma^* \mid \langle \varepsilon, q_0, u \rangle \to^* \langle \alpha, q, \beta \rangle, q \in F\}$$
  Il contenuto finale del nastro è irrilevante ai fini dell'accettazione

Definizione 3.1.2: Linguaggi ricorsivamente enumerabili

Un linguaggi è **ricorsivamente enumerabile** se è accettato da una macchina di Turing

Definizione 3.1.3: Linguaggi ricorsivi

Un linguaggio è **ricorsivo** se è accettato da una macchina di Turing che termina su ogni input

3.1.1 Sintassi del simulatore

```
<current state> <read> <write> <dir> <next state>
```

Dove:

- `<current state>`: è lo stato corrente (dove si trova la testina)
- `<read>`: è il simbolo che viene letto
- `<write>`: è il simbolo che viene scritto
- `<dir>`: è la direzione L (Sinistra), R (Destra), $*$ (Nessuna)
- `<next state>`: è in quale stato deve andare

Esempio

Definire la TM che riconosce il linguaggio regolare $L = \{0^n 1^m \mid n, m \geq 0\}$

- Leggiamo tutti i simboli 0 spostandoci a destra
- Quando troviamo un 1 cambiamo stato e leggiamo tutti i simboli 1 spostandoci a destra
- Se troviamo B la stringa è accettata, se troviamo 0 la macchina si blocca in uno stato di non accettazione

Usando la sintassi del simulatore:

```
0 0 0 r 0
0 1 1 r 1
0 _ _ * halt-accept
1 1 1 r 1
1 _ _ * halt-accept
```

"_" è Blank

Esercizio

Si dia una macchina di Turing che, data in input una stringa di 0 e 1, produca in output la stringa privata degli 0. Quindi, per esempio, data la stringa 01100100, produce in output la stringa 111. ***E' assolutamente necessario dare prima una descrizione a parole dell'algoritmo****, e solo successivamente la matrice di transizione, preferibilmente usando nomi significativi per gli stati.*

- **Descrizione algoritmo**: L'idea è quella di spostare a sinistra tutti gli 1 e a destra avere tutti gli 0, una volta che incontreremo un blank a destra sappiamo che siamo arrivati alla fine della stringa, quindi torniamo indietro e poi cancelliamo tutti gli 0 fino a riincontrare il blank.
- **MT**:

```
s0   0 0 r gor
s0   1 1 r gor
gor  1 0 l gol // Facciamo il bit-flip
gor  0 0 r gor // Andiamo a destra finchè non c'è 1
gor  _ _ l goll // Blank fine stringa torniamo indietro
gol  0 0 l gol // Andiamo a sinistra finchè non c'è 1 o _
gol  1 1 r cs // Ultimo 1 spostato, concateniamo 1
gol  _ _ r cs // Inizio stringa raggiunto, concateniamo 1
cs   0 1 r gor // "Concatenazione"
goll 0 0 l goll // Sinistra finchè non c'è 1
goll 1 1 r cb // 1 trovato, iniziamo a cancellare 0
goll _ _ r cb // _ trovato, iniziamo a cancellare 0
cb   0 _ r cb // se 0 cancelliamo con _
cb   _ _ * halt // fine stringa raggiunta
```

3.2 Funzioni T-calcolabili

Le funzioni T-calcolabili sono quelle funzioni che possono essere calcolate da una Macchina di Turing (TM).
Consideriamo una definizione di TM semplificata:

$$\mathcal{M} = \langle Q, \Sigma, \delta, q_0, B \rangle$$

dove $\Sigma = \Gamma$ include almeno un altro simbolo oltre al blank B, e la funzione di transizione $\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, R\}$ è una funzione parziale.
Per calcolare una funzione $f : Input \to Output$, è necessario stabilire un modo per codificare gli elementi del dominio e del codominio come stringhe sul nastro della TM. Questo si realizza tramite due funzioni iniettive: $c_{IN} : Input \to \Sigma^*$ per l'input e $c_{OUT} : Output \to \Sigma^*$ per l'output.

- **Direttive di Input/Output**
  - **Input**: La testina è inizialmente posizionata sul primo simbolo della stringa che rappresenta l'input. La configurazione iniziale per un input i è $f_{IN}(i) = \langle \varepsilon, q_0, c_{IN}(i) \rangle$
  - **Output**: Il risultato della funzione è rappresentato dalla stringa che rimane sul nastro quando l'esecuzione della macchina termina. Se la stringa sul nastro rappresenta effettivamente un risultato valido (secondo la codifica c_{OUT}), allora la funzione è definita per quell'input; altrimenti, la funzione risulta indefinita (o "malformata" se si ferma in una configurazione non valida). Formalmente

$$f_{OUT}(\langle \alpha, q, \beta \rangle) = \begin{cases} o & \text{se } c_{OUT}(\alpha\beta) = o \\ \text{Indefinito} & \text{altrimenti} \end{cases}$$

Esempio

Definire una TM che calcola la sommaa di due numeri codificati in unario separati da #, quindi $1^n\#1^m$ *deve dare come output* 1^{n+m}. L'algoritmo sposta semplicemente tutti gli 1 prima di # alla fine del secondo argomento cancellando alla fine il separatore.

```
0    1 _ r go_r
0    # _ r halt
go_r 1 1 r go_r
go_r # # r go_r
go_r _ 1 l go_l
go_l 1 1 l go_l
go_l # # l go_l
go_l _ _ r 0
```

3.3 Tesi di Church-Turing

Sebbene le MT non abbiano istruzioni come `goto` o la possibilità di accedere alla memoria tramite indici come in un linguaggio di programmazione convenzionale, dimostrano una sorprendente potenza espressiva. Concetti come i "salti" possono essere realizzati tramite sequenze di spostamenti elementari della testina sul nastro. Le MT possono essere considerate una versione embrionale dell'architettura di Von Neumann.

3.3.1 La tesi

La Tesi di Church-Turing afferma che una funzione è calcolabile con un algoritmo se e solo se è ricorsiva (parziale). Ciò significa che la classe delle funzioni T-calcolabili (quelle calcolabili da una Macchina di

Turing) è identificata con la classe intuitiva delle funzioni "calcolabili con un algoritmo". Non può essere dimostrata formalmente. È una congettura che collega una nozione informale e intuitiva ("calcolabile con un algoritmo") a una nozione matematica precisa (funzione ricorsiva o T-calcolabile) però ci sono diversi argomenti significativi che ne rafforzano la validità:

- **Equivalenza dei Formalismi**: Tutti i formalismi di calcolo proposti indipendentemente (come le funzioni μ-ricorsive, il lambda-calcolo, le Random Access Machine, le macchine a contatori) si sono dimostrati equivalenti alle Macchine di Turing. Tutti definiscono la stessa classe di funzioni calcolabili, chiamata la classe delle funzioni ricorsive (parziali)
- **Potere Espressivo**: Ogni funzione che può essere ragionevolmente considerata "algoritmica" si è rivelata appartenere a questa classe
- **Algoritmi di Trasformazione**: Le dimostrazioni di equivalenza tra formalismi spesso mostrano che esiste un algoritmo che, dato un programma in un formalismo, produce un programma equivalente nel secondo formalismo. Questo implica che ogni formalismo offre tutti i possibili algoritmi

3.4 Nozione di algoritmo e funzioni ricorsive primitive

3.4.1 Funzioni Ricorsive Primitive $\mathcal{PR}$

Per tentare di formalizzare la nozione di funzione calcolabile, vengono introdotte le funzioni ricorsive primitive ($\mathcal{PR}$). Si tratta di funzioni da $\mathbb{N}^k$ in $\mathbb{N}$ (con $k \geq 0$) definite induttivamente a partire da un insieme di funzioni base e schemi di costruzione:

- **Funzioni base**:
  - Funzione zero ($Z(x_1, \ldots, x_n) = 0$): Restituisce sempre 0, indipendentemente dagli input
  - Funzione successore ($S(x) = x + 1$): Aggiunge 1 al suo argomento
  - Funzione proiezione ($\Pi_i^n(x_1, \ldots, x_n) = x_i$): Restituisce il valore del i-esimo argomento di una tupla di n argomenti
- **Schemi di costruzione**:
  - **Composizione**: Se $h : \mathbb{N}^k \to \mathbb{N}$ e $g_i : \mathbb{N}^n \to \mathbb{N}$ (per $i \in [1, k]$) sono $\mathcal{PR}$, allora anche $f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$ è $\mathcal{PR}$
  - **Ricorsione primitiva**: Se $g : \mathbb{N}^{n-1} \to \mathbb{N}$ e $h : \mathbb{N}^{n+1} \to \mathbb{N}$ sono $\mathcal{PR}$, allora anche $f : \mathbb{N}^n \to \mathbb{N}$ è $\mathcal{PR}$ se definita come:
    - Caso base: $f(x_1, \ldots, x_{n-1}, 0) = g(x_1, \ldots, x_{n-1})$
    - Chiamata ricorsiva: $f(x_1, \ldots, x_{n-1}, y + 1) = h(x_1, \ldots, x_{n-1}, y, f(x_1, \ldots, x_{n-1}, y))$

Una funzione è ricorsiva primitiva se può essere costruita a partire dalle funzioni banalmente calcolabili di base, utilizzando la composizione (generalizzata) e un meccanismo di ricorsione molto semplice nel quale il caso base è zero e la chiamata ricorsiva è sempre effettuata sul predecessore, in modo che la ricorsione sia sempre terminante.
Introduciamo un semplice linguaggio di programmazione:

- **Struttura del programma**:
  - Un programma è una sequenza di dichiarazioni di funzione
  - Si assume l'esistenza di un insieme di nomi di funzione (es. f) con una certa arità (numero di parametri) e un insieme di variabili (es. x, y, z)
- **Formati delle Dichiarazioni di Funzione**:

- Formato generale: $f(x_1, \ldots, x_n) = e$. Questa è la forma per le funzioni definite tramite le funzioni base (zero, successore, proiezione) o composizione
- Formato per ricorsione primitiva (se $n > 0$):
  - $f(x_1, \ldots, x_{n-1}, Z) = e$ (caso base)
  - $f(x_1, \ldots, x_{n-1}, S(y)) = e'$ (passo ricorsivo) In questo caso, la funzione f è definita per ricorsione primitiva.

- Definizione delle Espressioni (e):

$$e ::= x|Z|S(e)|f(e_1, \ldots, e_n)$$

- Vincoli Contestuali: Questo linguaggio include delle regole per la correttezza del codice:
  - Un nome di funzione non può essere dichiarato più volte
  - Una variabile non può apparire più di una volta come parametro di una funzione
  - Ogni dichiarazione di funzione può contenere chiamate solo a funzioni dichiarate precedentemente
  - Nel caso di una funzione definita per ricorsione primitiva, l'espressione e' (del passo ricorsivo) può contenere chiamate ricorsive a f, ma solo nella forma $f(x_1, \ldots, x_{n-1}, y)$ (cioè, la chiamata ricorsiva è sempre sul predecessore dell'ultimo argomento)
  - In ogni chiamata di funzione, il numero degli argomenti deve corrispondere all'arità (il numero di parametri attesi)
  - L'ultima funzione dichiarata in un programma è considerata la funzione principale

- **Direttive di Esecuzione (Sistema di Transizione)**:
  - Le configurazioni sono rappresentate da espressioni senza variabili
  - Le direttive consistono nel procedere per sostituzioni successive fino a raggiungere il risultato finale
  - Le regole di riduzione specificano come le espressioni vengono semplificate:
    - Se c'è una dichiarazione $f(x_1, \ldots, x_n) = e$, allora $f(e_1, \ldots, e_n)$ si riduce a e con i parametri sostituiti dagli argomenti ($e[e_1/x_1 \ldots e_n/x_n]$)
    - Per le funzioni definite per ricorsione primitiva ($f(x_1, \ldots, x_{n-1}, Z) = e$ e $f(x_1, \ldots, x_{n-1}, S(y)) = e'$):
      - $f(e_1, \ldots, e_{n-1}, Z)$ si riduce a e con i parametri sostituiti
      - $f(e_1, \ldots, e_{n-1}, S(e_n))$ si riduce a e' con i parametri sostituiti
  - Le riduzioni possono essere applicate anche all'interno di espressioni più complesse: se e si riduce a e', allora $S(e)$ si riduce a $S(e')$, e $f(..., e_i, ...)$ si riduce a $f(..., e_i', ...)$

- **Configurazioni di Arresto**:
  - Le configurazioni di arresto sono quelle della forma $S^x(Z)$, che rappresentano i numeri naturali (es. $S(S(Z))$ rappresenta 2). Queste sono abbreviate come $\bar{x}$ (es. $\bar{2}$)

- **Direttive di Input/Output**:
  - **Input**: L'input $x_1, \ldots, x_n$ per la funzione principale f viene codificato come $f(\bar{x}_1, ..., \bar{x}_n)$
  - **Output**: L'output $\bar{x}$ (una rappresentazione di numero naturale) viene decodificato nel numero x

Le direttive di esecuzione sono non deterministiche (più scelte possibili, ad esempio nell'ordine di valutazione di sotto-espressioni), ma il risultato finale è sempre lo stesso. Questo è perché, come provato per induzione, tutte le funzioni ricorsive primitive sono totali, ovvero definite su ogni possibile argomento e garantiscono sempre la terminazione. Questo è il motivo per cui il meccanismo di ricorsione è descritto come "molto semplice" e sempre terminante, poiché il caso base è zero e la chiamata ricorsiva è sempre effettuata sul predecessore.

Definizione 3.4.1

I programmi che calcolano funzioni ricorsive primitive possono essere numerati *effettivamente*.

"Numerare effettivamente" significa che esiste un algoritmo in grado di elencare questi programmi uno dopo l'altro. Inoltre, la corrispondenza è bidirezionale: dato un programma, si può calcolare il suo numero d'ordine, e viceversa, dato un numero n, si può costruire l'n-esimo programma.

- Un programma è considerato una stringa su un alfabeto (un insieme finito di simboli)
- È possibile ordinare queste stringhe, ad esempio per lunghezza e poi lessicograficamente per quelle di uguale lunghezza
- Questo ordinamento permette di elencare i programmi ben formati in una sequenza, associando a ciascuno un numero naturale (il suo posto nell'elenco)

Teorema 3.4.1

Esistono funzioni calcolabili con un algoritmo che non sono ricorsive primitive

Il teorema afferma che la classe delle funzioni ricorsive primitive ($\mathcal{PR}$) non è sufficiente a rappresentare tutte le funzioni che possono essere calcolate tramite un algoritmo. In altre parole, esistono funzioni che sono evidentemente calcolabili da un algoritmo, ma che non rientrano nella definizione di funzione ricorsiva primitiva. La prova di questo teorema si basa su un metodo chiamato **diagonalizzazione**:

- Questo metodo sfrutta il fatto che i programmi (o algoritmi) che calcolano funzioni ricorsive primitive possono essere numerati effettivamente. Questo significa che è possibile associare a ogni funzione ricorsiva primitiva a una variabile un indice naturale, creando un elenco enumerabile: $f_x | x \geq 0$
- Si definisce quindi una nuova funzione f come $f(x) = f_x(x) + 1$
- Questa funzione f è ovviamente calcolabile con un algoritmo. Il motivo è che la numerazione f_x è algoritmica (cioè, possiamo generare l'x-esimo programma) e ogni f_x è a sua volta calcolabile con un algoritmo
- Tuttavia, questa funzione f non può appartenere alla classe delle funzioni ricorsive primitive. Se f fosse ricorsiva primitiva, esisterebbe un indice $\bar{x}$ tale che $f = f_{\bar{x}}$. Ma allora, applicando f a $\bar{x}$, si otterrebbe la contraddizione $f(\bar{x}) = f_{\bar{x}}(\bar{x}) = f_{\bar{x}}(\bar{x}) + 1$. Questa contraddizione dimostra che f non può essere ricorsiva primitiva

Il nome "diagonalizzazione" deriva dall'idea di una matrice dove le righe sono gli indici (x) e le colonne sono gli argomenti (y), e la funzione f è costruita per essere diversa, per ogni argomento x, dal valore $f_x(x)$ che appare sulla diagonale della matrice.
Requisiti:

- Esiste una numerazione algoritmica degli algoritmi che definiscono le funzioni della classe
- Le funzioni della classe sono totali (cioè, definite su tutti gli input). Poiché le funzioni ricorsive primitive sono per definizione totali e i loro programmi possono essere numerati effettivamente, la diagonalizzazione è applicabile

3.5 Numerazione algoritmica delle funzioni ricorsive

Analogamente a quanto visto per le funzioni ricorsive primitive, è possibile codificare qualsiasi Macchina di Turing come una stringa su un alfabeto fissato Σ. Questo è possibile perché il funzionamento di una macchina è indipendente dai nomi scelti per stati e simboli, quindi si possono sempre considerare stati e simboli numerati in $\mathbb{N}$. Le quintuple che descrivono le transizioni di una MT (stato corrente, simbolo letto, nuovo stato, simbolo scritto, direzione di spostamento) possono essere rappresentate come sequenze di numeri d'ordine e poi codificate come stringhe. L'intera funzione di transizione di una MT può essere codificata come la concatenazione di queste stringhe, usando un separatore specifico (ad esempio, 11). In questo modo, si ottiene una codifica unica ($\langle\mathcal{M}\rangle$) per ogni macchina $\mathcal{M}$

Definizione 3.5.1

Le macchine di Turing possono essere numerate *effettivamente*.

Questo significa che, una volta che tutte le macchine sono rappresentate come stringhe su un alfabeto, si può procedere per elencazione (generare una stringa dopo l'altra in un ordine prestabilito), proprio come avviene per i programmi che calcolano funzioni ricorsive primitive.

Definizione 3.5.2

Le funzioni ricorsive ($\mathbb{N} \to \mathbb{N}$) possono essere numerate effettivamente.

Questa è una conseguenza diretta della definizione precedente e della Tesi di Church-Turing.

Glossario:

- $\mathcal{M}_x$: la macchina di Turing con indice x nella numerazione
- ϕ_x: la funzione $\mathbb{N} \to \mathbb{N}$ calcolata da $\mathcal{M}_x$

3.5.1 Osservazioni

- La numerazione può essere estesa a funzioni ricorsive da qualsiasi insieme A a qualsiasi insieme B, purché vengano utilizzate codifiche "ragionevoli"
- È cruciale notare che $n \neq m$ non implica necessariamente $\phi_n \neq \phi_m$. Questo significa che algoritmi diversi (con indici diversi) possono calcolare la stessa identica funzione
- Conoscere un algoritmo che calcola una funzione ricorsiva f è equivalente a conoscere almeno un numero n tale che $f = \phi_n$. Viceversa, dato un indice n, si può costruire la rappresentazione della macchina corrispondente e quindi un algoritmo
- Vi è una differenza significativa tra affermare che una funzione "è ricorsiva" (proprietà della funzione stessa) e affermare che "si conosce n tale che $f = \phi_n$" (si ha un'implementazione specifica)

3.5.2 Corollari

- Le funzioni ricorsive (e anche le funzioni ricorsive totali) hanno la cardinalità del numerabile. Tuttavia, le funzioni ricorsive totali non possono essere numerate effettivamente (decisivamente), poiché non è possibile stabilire algoritmicamente se un algoritmo calcola una funzione totale
- Esistono funzioni, sia parziali che totali, che non sono ricorsive. Ciò si deduce dal fatto che l'insieme delle funzioni caratteristiche dei sottoinsiemi di N ha una cardinalità strettamente maggiore
- Ogni funzione ricorsiva possiede un'infinità numerabile di indici nella numerazione effettiva. Questo perché è sempre possibile creare varianti di una MT (ad esempio, aggiungendo stati o istruzioni inutili) che calcolano la stessa funzione ma hanno un indice diverso

Esiste una corrispondenza biunivoca tra tutte le possibili macchine di Turing (algoritmi), tutte le stringhe (su un certo alfabeto Σ) che le rappresentano (rappresentazioni degli algoritmi), e i numeri naturali (indici degli algoritmi). **Essenziale è la possibilità di vedere le stringhe (o, ancora più astrattamente, i numeri naturali) al tempo stesso come dati(argomenti del calcolo) e programmi (macchine che eseguono il calcolo)**. Questo permette di applicare ai formalismi analizzati finora uno dei principi base dell'informatica, ossia quello di poter passare programmi come argomenti ad altri programmi.

3.6 Macchina di Turing universale e calcolatore

Una Macchina di Turing (MT) può essere vista come un programma, ma la sua funzione di transizione (il suo "programma") è fissa.
Un calcolatore moderno, invece, riceve programmi come dati in ingresso, insieme ai dati stessi da elaborare. Questo suggerisce che un calcolatore è a sua volta un programma e in accordo con la Tesi di Church-Turing (che identifica le funzioni calcolabili con gli algoritmi formalizzabili), se un calcolatore è un programma, allora deve corrispondere a una Macchina di Turing.

Definizione 3.6.1

La funzione universale $f_U : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ è definita come

$$f_U(x, y) = \begin{cases} \phi_x(y) & \text{se } \phi_x(y) \downarrow \text{ (è definita)} \\ \text{indefinito} & \text{altrimenti} \end{cases}$$

Il teorema afferma che f_U è ricorsiva, il che significa che esiste una Macchina di Turing in grado di calcolarla.
Prova: La funzione f_U può essere calcolata mediante un algoritmo che prende in input x e y, genera la macchina $\mathcal{M}_x$ (corrispondente all'indice x), e poi esegue/simula $\mathcal{M}_x$ sull'input y. La Tesi di Church-Turing giustifica il fatto che f_U sia ricorsiva.

3.7 Risolvibilità di problemi

3.7.1 Problema dell'arresto

Il Problema dell'arresto (o Halting Problem) ha come obiettivo l'identificazione della classe dei problemi risolvibili tramite un elaboratore.

- **Definizione**: Il Problema dell'arresto si chiede se esista un programma (o algoritmo) che, dati in input un qualunque programma P e un suo input y, sia in grado di stabilire se il programma P eseguito su y termina o meno.

Teorema 3.7.1: Indecidibilità del problema dell'arresto

La funzione $f_{\mathcal{H}} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ è definita da:

$$f_{\mathcal{H}}(x, y) = \begin{cases} 1 & \text{se } \phi_x(y) \downarrow \\ 0 & \text{altrimenti} \end{cases}$$

La funzione $f_{\mathcal{H}}$ non è ricorsiva. Questo significa che il Problema dell'arresto è indecidibile, ciò implica che non è possibile dare un algoritmo generale che, esaminando unicamente il codice del programma P e l'input y, determini con certezza se P terminerà o meno su y. Simulando l'esecuzione del programma (cosa possibile con una Macchina di Turing Universale), si ottiene una risposta positiva solo se il programma

termina; se non termina, la simulazione potrebbe proseguire all'infinito senza fornire una risposta. Prima di dimostrare il teorema introduciamo un lemma

Lemma 3.7.1

La funzione $f_{\mathcal{K}} : \mathbb{N} \to \mathbb{N}$ definita da:

$$f_{\mathcal{K}}(x) = \begin{cases} 1 & \text{se } \phi_x(x) \downarrow \\ 0 & \text{altrimenti} \end{cases}$$

Non è ricorsiva.

Dimostrazione

- Si assume che $f_{\mathcal{K}}$ sia ricorsiva, ovvero che esista una MT $\mathcal{M}_{\mathcal{K}}$ che la calcola
- Si costruisce una nuova funzione g (e il suo algoritmo):
  1. input x
  2. se $\mathcal{M}_{\mathcal{K}}(x) = 1$ (cioè $\phi_x(x)$ termina), allora g non termina
  3. altrimenti ($\mathcal{M}_{\mathcal{K}}(x) = 0$, cioè $\phi_x(x)$ non termina), g restituisce 1
- Per la Tesi di Church-Turing, se g è algoritmica, allora deve esistere un indice $\bar{z}$ tale che $g = \phi_{\bar{z}}$ e considerando il comportamento di $\phi_{\bar{z}}(\bar{z})$:
  - $\phi_{\bar{z}}(\bar{z}) \uparrow \iff g(\bar{z}) \uparrow \iff f_{\mathcal{K}}(\bar{z}) = 1 \iff \phi_{\bar{z}}(\bar{z}) \downarrow$
  - $\phi_{\bar{z}}(\bar{z}) = 1 \iff g(\bar{z}) = 1 \iff f_{\mathcal{K}}(\bar{z}) = 0 \iff \phi_{\bar{z}}(\bar{z}) \uparrow$

Il lemma implica il teorema in quanto, se la funzione $f_{\mathcal{H}}$ fosse ricorsiva, lo sarebbe anche $f_{\mathcal{K}}$ poichè $f_{\mathcal{K}}(x) = f_{\mathcal{H}}(x, x)$, contraddicendo il lemma.

3.7.2 Insiemi ricorsivi e ricorsivamente numerabili

Possiamo esprimere il teorema 3.7.1 dicendo che:

- $\mathcal{H} = \{\langle x, y\rangle \mid x, y \in \mathbb{N}, \phi_x(y) \downarrow\}$ (le coppie programma-input per cui il programma termina) è un sottoinsieme di $\mathbb{N} \times \mathbb{N}$ che non è ricorsivo
- $\mathcal{K} = \{x \mid x \in \mathbb{N}, \phi_x(x) \downarrow\}$ (i programmi che terminano su se stessi) non è ricorsivo. $\mathcal{K}$ è un caso particolare di $\mathcal{H}$, poiché $f_{\mathcal{K}}(x) = f_{\mathcal{H}}(x, x)$

Le nozioni di insieme ricorsivo e ricorsivamente enumerabile posso essere anche formulate nel modo seguente:

- Un insieme $A \subseteq \mathbb{N}$ è ricorsivamente enumerabile se esiste una funzione ricorsiva $f : \mathbb{N} \to \{0, 1\}$ tale che:

  $$f(x) = \begin{cases} 1 & \text{se } x \in A \\ 0 \text{ oppure indefinito} & \text{altrimenti} \end{cases}$$

  Questo significa che esiste una Macchina di Turing (o un algoritmo) che, per ogni input $x \in \mathbb{N}$, termina sempre e restituisce 1 se $x \in A$ oppure 0 se $x \notin A$. Gli insiemi ricorsivi corrispondono ai problemi decidibili. Tutti gli insiemi finiti e quelli il cui complementare è finito sono esempi di insiemi ricorsivi.

- Un insieme $A \subseteq \mathbb{N}$ è ricorsivamente enumerabile se la sua funzione semicaratteristica ψ_A è ricorsiva:

$$\psi_A(x) = \begin{cases} 1 & \text{se } x \in A \\ \text{indefinito} & \text{altrimenti} \end{cases}$$

  Questo significa che esiste una Macchina di Turing (o un algoritmo) che, per ogni input $x \in \mathbb{N}$, restituisce 1 se $x \in A$, ma potrebbe non terminare (o restituire 0) se $x \notin A$. Gli insiemi r.e. corrispondono ai problemi semidecidibili. Un problema è semidecidibile se esiste un algoritmo che, per le istanze per cui la risposta è "vero", fornisce una risposta positiva in un numero finito di passi, mentre per le istanze per cui la risposta è "falso" potrebbe non terminare. L'insieme $\mathcal{H}$ (e $\mathcal{K}$) è un esempio di insieme r.e. ma non ricorsivo.

Teorema 3.7.2: Il teorema di Post

A ricorsivo $\iff$ $\bar{A}$ ricorsivo $\iff$ A e $\bar{A}$ ricorsivamente enumerabili

Dimostrazione

- ($\Rightarrow$) Se A è ricorsivo, esiste un algoritmo $\mathcal{M}_A$ che termina sempre e restituisce 1 per $x \in A$ e 0 per $x \notin A$. Per ottenere un algoritmo per A, basta invertire l'output di $\mathcal{M}_A$ (restituire 0 se $\mathcal{M}_A(x)$ è 1 e viceversa). Entrambi gli insiemi sono quindi ricorsivi e di conseguenza r.e.
- ($\Leftarrow$) Se A e $\bar{A}$ sono entrambi r.e., esistono due algoritmi che terminano con 1 rispettivamente se l'input appartiene ad A o a $\bar{A}$. Per decidere se $x \in A$, basta eseguire i due algoritmi in interleaving, alternando passi di esecuzione dell'uno e dell'altro. Poiché ogni x appartiene o ad A o ad $\bar{A}$, una delle due esecuzioni terminerà, fornendo una risposta definita (1 o 0), garantendo la terminazione e quindi la ricorsività dell'insieme

Dal Teorema di Post e dal fatto che $\mathcal{K}$ è r.e. ma non ricorsivo, si deduce che il suo complemento $\mathcal{K} = \{x | \phi_x(x) \uparrow\}$ (l'insieme dei programmi che non terminano su se stessi) non è ricorsivamente enumerabile.

Teorema 3.7.3: Caratterizzazione degli insiemi r.e.

Questo teorema fornisce formulazioni equivalenti per la definizione di insieme r.e.:

1. Un insieme A è ricorsivamente enumerabile
2. A è l'immagine (l'insieme dei valori) di una funzione ricorsiva
3. A è l'insieme vuoto, oppure è l'immagine di una funzione ricorsiva totale

Dimostrazione

- **(1)⇒(2)**: Un insieme A è r.e. se la sua funzione semicaratteristica ψ_A è ricorsiva. Questo significa che esiste un algoritmo (una Macchina di Turing) che, per ogni input x:
  - Restituisce 1 se $x \in A$
  - Non termina o restituisce 0 altrimenti (ovvero, se $x \notin A$)

  Possiamo modificare l'algoritmo in modo tale che, per ogni $x \in \mathbb{N}$, invece di restituire 1 restituisca x. Questo algoritmo definisce una funzione ricorsiva f il cui insieme di valri è A
- **(2)⇒(3)**: Il punto precedente non fornisce un algoritmo (o MT) per generare tutti gli elementi di A, in quanto il calcolo di $f(x)$ potrebbe non terminare su qualche x. Per superare questo problema, utilizziamo una tecnica di enumerazione nota come "**zig-zag**". Questa tecnica ci permette di esplorare tutte le possibili computazioni di f per tutti i possibili input in un modo che garantisce che, se una computazione termina, essa verrà trovata in un tempo finito.
- **(3)⇒(1)**: Se A è l'insieme vuoto ($\emptyset$), è per definizione ricorsivamente enumerabile. Altrimenti sia f la funzione totale il cui insieme di valori è A. Definiamo allora la funzione semicaratteristica g nel modo seguente:

$$g(x) = \begin{cases} 1 & \text{se } f(y) = x \text{ per qualche } y \\ \text{indefinito} & \text{altrimenti} \end{cases}$$

  Questo significa che, dato un input x, l'algoritmo per $g(x)$ deve eseguire l'algoritmo che genera successivamente tutti gli elementi di A. Pertanto, questa funzione g è ricorsiva, e di conseguenza, A è ricorsivamente enumerabile

Approfondimento: Tecnica a "zig-zag"

Immagina una griglia infinita:

- Le righe rappresentano i possibili input per la funzione f
- Le colonne rappresentano il numero di passi di calcolo che concediamo a f per un dato input
- Ogni "cella" `(input, passi)` della griglia rappresenta "cosa succede se eseguo f(`input`) per passi istruzioni"

Invece di percorrere la griglia per righe o per colonne, la tecnica a zig-zag ci suggerisce di percorrerla in diagonale. Questo garantisce che, se un calcolo di $f(x)$ termina per qualche x, lo troveremo prima o poi.

3.7.3 Problemi decidibili e indecidibili

Un problema di decisione è una domanda che può avere come risposta "sì" o "no" per ogni istanza data. Per formalizzare un problema di decisione, lo si può esprimere in due modi equivalenti:

- Come una funzione $\mathcal{P}$ che prende un'istanza del problema come input e restituisce 0 o 1 (dove 1 sta per "vero" e 0 per "falso"). Questo è anche chiamato un **predicato**
- Come l'insieme di tutte le istanze per cui la risposta è "sì" ($\{a \in A \mid \mathcal{P}(a) = 1\}$)

A partire da questa formalizzazione, vengono definite due proprietà chiave per i problemi di decisione:

- **Problema decidibile**: Un problema di decisione $\mathcal{P}$ è decidibile se l'insieme delle sue istanze per cui $\mathcal{P}$ è vero è ricorsivo. Questo significa che esiste un algoritmo che, per ogni istanza del problema, fornisce una risposta (**positiva o negativa**) in un numero finito di passi. L'algoritmo termina sempre, indipendentemente dalla risposta. Esempi:

- Determinare se un grafo è aciclico
- Determinare se un grafo è aciclico
- Determinare se una sequenza compare come sottosequenza di un'altra
- Decidere se una stringa su $\{0, 1\}$ ha un numero pari di 1
- Decidere se una stringa u appartiene a un linguaggio L (se L è decidibile)

- **Problema semidecidibile**: Un problema di decisione $\mathcal{P}$ è semidecidibile se l'insieme delle sue istanze per cui $\mathcal{P}$ è vero è ricorsivamente enumerabile. Questo significa che esiste un algoritmo che, per le istanze per cui $\mathcal{P}$ è vero, fornisce una risposta **positiva** in un numero finito di passi; tuttavia, per le istanze per cui $\mathcal{P}$ è falso, l'algoritmo potrebbe anche non terminare

3.7.4 Riducibilità

Definizione 3.7.1

Dati due problemi $\mathcal{P}$ e $\mathcal{Q}$, diciamo che $\mathcal{P}$ è riducibile a $\mathcal{Q}$, e scriviamo $\mathcal{P} \leq \mathcal{Q}$, se esiste una funzione totale ricorsiva $f : \mathbb{N} \to \mathbb{N}$, detta funzione di riduzione, tale che, per ogni $x \in \mathbb{N}, x \in \mathcal{P}$ se e solo se $f(x) \in \mathcal{Q}$

Questa definizione può essere applicata anche a sottoinsiemi di N^k o Σ^*, e i problemi possono essere sottoinsiemi di insiemi diversi.
Intuitivamente, se $\mathcal{P}$ è riducibile a $\mathcal{Q}$, significa che $\mathcal{Q}$ è "più difficile" (o, più precisamente, "non più facile") di $\mathcal{P}$. Questo perché, dato un algoritmo $M_\mathcal{Q}$ che decide $\mathcal{Q}$, è possibile ottenere un algoritmo $M_\mathcal{P}$ che decide $\mathcal{P}$. $M_\mathcal{P}$ si ottiene eseguendo, a partire dall'input x, prima l'algoritmo M_f che calcola $f(x)$, e poi l'algoritmo $M_\mathcal{Q}$ con input $f(x)$. Questo algoritmo $M_\mathcal{P}$ è garantito essere un algoritmo che termina sempre e restituisce 1 se $x \in \mathcal{P}$ e 0 se $x \notin \mathcal{P}$. Una proprietà analoga vale anche per la semidecidibilità.
La riducibilità è uno strumento potente per dimostrare proprietà sui problemi:

- Se $\mathcal{Q}$ è ricorsivo (o ricorsivamente enumerabile), allora anche $\mathcal{P}$ è ricorsivo (o ricorsivamente enumerabile)
- Se $\mathcal{P}$ non è ricorsivo (o non è ricorsivamente enumerabile), allora non può esserlo neanche $\mathcal{Q}$ per il teorema di Post

La relazione di riducibilità è riflessiva e transitiva.

Esempio

Consideriamo l'insieme $\mathcal{K} = \{k \mid \phi_x(x) \downarrow\}$ e l'insieme $\mathcal{Z} = \{\phi_x \text{ è la funzione costante } 0\}$. Mostriamo che $\mathcal{K} \leq \mathcal{Z}$. Questa implicazione mostra che anche l'insieme $\mathcal{Z}$ è non ricorsivo, ossia non è possibile decidere se un algoritmo restituisce 0 su ogni input.La riduzione consiste nel costruirre una funzione totale e ricorsiva $g : \mathbb{N} \to \mathbb{N}$ che trasforma un input x in un nuovo algoritmo $g(x)$. La funzione $g(x)$ viene costruita come segue:

- Esegue $\mathcal{M}_x(x)$
- Se $\mathcal{M}_x(x)$ termina, $g(x)$ restituisce 0
- Se $\mathcal{M}_x(x)$ non termina, $g(x)$ non termina

In questo modo si ha:

$$\phi_{g(x)}(y) = \begin{cases} 0 & \text{se } x \in \mathcal{K} \Rightarrow g(x) \in \mathcal{Z} \\ \text{indefinito} & \text{altrimenti} \Rightarrow g(x) \notin \mathcal{Z} \end{cases}$$

Dato che $\mathcal{K} \leq \mathcal{Z}$ e $\mathcal{K}$ è noto essere un problema non ricorsivo (indecidibile), la riduzione dimostra che anche $\mathcal{Z}$ non è ricorsivo. Questo implica che non è possibile avere un algoritmo che decida se un dato programma restituirà sempre 0 come output per qualsiasi input.

3.7.5 Teorema di Rice

Il teorema di Rice afferma che ogni proprietà "semantica" (estensionale) dei programmi (algoritmi) non è decidibile.

- **Proprietà Estensionali** Π: una proprietà dei programmi (o algoritmi) è detta estensionale se dipende esclusivamente dal **comportamento input/output** del programma, e non dalla sua implementazione interna o da suo codice specifico. In altre parole, se due programmi calcolano la stessa funzione (cioè hanno lo stesso comportamento), allora decono o entrambi soddisfare la proprietà o entrambi non soddisfarla. Formalmente:

$$\forall x, y \in \mathbb{N} \mid (x \in \Pi \land \phi_x = \phi_y) \Rightarrow y \in \Pi$$

Teorema 3.7.4: Teorema di Rice

Una proprietà estensionale Π è ricorsiva $\iff$ è banale.

Dimostrazione

- L'implicazione $\Leftarrow$ è ovvia
- Per l'implicazione $\Rightarrow$: consideriamo una proprietà Π
  - Supponiamo che Π non banale
  - Mostriamo che non può essere ricorsiva attraverso una riduzione dal problema dell'arresto $\mathcal{K} = \{x \mid \phi_x(x) \downarrow\}$, che è noto per essere non ricorsivo (indecidibile), a Π
  - L'idea è costruire un algoritmo $g(x)$ che trasforma un programma x in un nuovo programma $g(x)$ tale che x termina su se stesso ($x \in \mathcal{K}$) se e solo se $g(x)$ soddisfa la proprietà estensionale Π. Se fosse possibile decidere Π, allora si potrebbe decidere anche $\mathcal{K}$, il che è un assurdo. Questo implica che Π non è ricorsiva

3.8 Altri modelli di calcolo

3.8.1 Funzioni μ-ricorsive

Le funzioni μ-ricorsive rappresentano un'estensione del formalismo delle funzioni ricorsive primitive, introdotto per catturare l'intera classe delle funzioni calcolabili, comprese quelle parziali, e a supportare la tesi di Church-Turing.

- **Operatore μ (Minimizzazione)**:
  - Data una funzione totale $f : \mathbb{N}^{(n+1)} \to \mathbb{N}$, si definisce la funzione $g : \mathbb{N}^n \to \mathbb{N}$ tramite l'operatore μ come: $g(x_1, \ldots, x_n) = \mu 0 y(f(x_1, \ldots, x_n, y))$
  - Il valore di g è il più piccolo y per cui $f(x_1, \ldots, x_n, y) = 0$, se tale y esiste
  - Se non esiste un y per cui $f(x_1, \ldots, x_n, y) = 0$, allora $g(x_1, \ldots, x_n)$ è indefinito (non termina). Questo rende le funzioni μ-ricorsive capaci di esprimere funzioni parziali, a differenza delle primitive ricorsive che sono sempre totali

Definizione 3.8.1

La classe delle funzioni μ-ricorsive è la più piccola classe di funzioni che include le funzioni ricorsive primitive (zero, successore, proiezione) e che è chiusa rispetto alle operazioni di composizione, ricorsione primitiva, e all'operatore μ (applicato, come detto, a funzioni totali).

Capitolo 4: Esercizi d'esame

Esercizio

Dire se le seguenti affermazioni sono vere o false motivando la risposta.

1. La proprietà dei programmi corrispondente all'insieme $\{x \mid x \leq 100\}$ non è estensionale. $\rightarrow$ **Vero** perchè una proprietà è estensionale se dipende solo dal comportamento del programma.
2. La proprietà dei programmi corrispondente all'insieme $\{x \mid x \leq 100\}$ è ricorsiva. $\rightarrow$ **Vero**, basta verificare se $x \leq 100$
3. La proprietà dei programmi corrispondente all'insieme $\{x \mid \phi_x(0) \leq 100\}$ è ricorsivamente enumerabile. $\rightarrow$ **Vero** perchè il programma termina per quelle x tali per cui $\phi_x(0)$ termina e restituisce un valore ≤ 100 altrimenti non termina.

Esercizio

Supponiamo di avere un predicato decidibile Q su coppie di numeri naturali $Q(x, y)$. Si consideri il predicato P sui numeri naturali tale che $P(x)$ è vero se e solo se esiste y tale che $Q(x, y)$ è falso. Si descriva un algoritmo di semidecisione per P.

```
input x;
y = 0
while(true) {
    if(Q(x,y)==false) return true;
    y++
}
```

Esercizio

Siano P e Q due insiemi r.e., e indichiamo con A_P^k l'esecuzione di k passi dell'algoritmo che semidecide P, e analogamente per Q. Si descriva in pseudocodice un algoritmo che semidecide $\{x \mid x \notin P \Rightarrow x+1 \in Q\}$.

```
input x;
k = 0
while(true) {
    if(Akp(x)==1) return 1
    if(Akq(x+1)==1) return 1
    k++
}
```

Esercizio

Sia $f : \mathbb{N} \to \mathbb{N}$ una funzione ricorsiva e A un insieme ricorsivamente enumerabile, allora $\{x \mid f(x) \in A\}$ è ricorsivamente enumerabile? Giustificare la riposta.
Sappiamo che f termina o può non terminare (definizione di ricorsivo), A è r.e. quindi per definizione termina o non termina. Quindi se $f(x)$ termina, bisogna controllare se il valore restituito fa terminare o no A. Nel caso in cui $f(x)$ non termini, anche A non termina. Quindi si, l'insieme dato è ricorsivamente enumerabile.

Esercizio

Sia $f : \mathbb{N} \to \mathbb{N}$ una funzione ricorsiva e A un insieme ricorsivo, allora $\{f(x) \mid x \in A\}$ è ricorsivamente enumerabile? Giustificare la risposta.
Si perchè per definizione l'immagine di f è ricorsivamente enumerabile

Esercizio

L'insieme $A = \{x \mid \phi_x(x) \uparrow$ oppure $\phi_x(x) = 5\}$, ossia l'insieme dei programmi che su se stessi non terminano oppure restituiscono 5, è ricorsivamente enumerabile? Giustificare accuratamente la risposta.
Usiamo il teorema di Post quindi:

- $\bar{A} = \{x \mid \phi_x(x) \downarrow$ e $\phi_x(x) \neq 5\}$ è ricorsivamente enumerabile
- Per determinare se $\bar{A}$ è ricorsiva, troviamo un insieme contenuto in $\bar{A}$ ovvero $K = \{x \mid \phi_x(x) \downarrow \}$, per definizione sappiamo che l'insieme K non è ricorsivo perchè termina sempre. Dobbiamo determinare se c'è una funzione di riducibilità $f(x)$ da K a $\bar{A}$ e per fare ciò costruiamo un algoritmo $M_{f(x)} = \{x \in K \iff f(x) \in \bar{A}\}$ e sappiamo che se termina $\phi_x(x) \neq 5$ altrimenti non termina. In questo modo dimostriamo che K è riducibile a $\bar{A}$ e visto che K non è ricorsivo anche $\bar{A}$ non lo è e quindi, per definizione A non è ricorsivamente enumerabile.