Fondamenti di Informatica

Università di Verona
Imbriani Paolo - VR500437
Professor Isabella Mastroeni

December 1, 2025

Contents

1 Cosa è l'informatica?

La domanda chiave di questo corso è: *"Cosa è l'informatica?"*. Ci sono diversi definizioni a seconda del contesto, ma in generale l'informatica è lo studio dei processi che trasformano l'informazione. Possiamo vedere, storicamente, diverse definizioni come quella in inglese come "Computer Science" che vede l'informatica come studio della calcolabilità, della computazione e dell'informazione.

1.1 Perché la calcolabilità

Si studia la calcolabilità perché ci aiuta a capire cosa possiamo fare con gli strumenti che abbiamo. Quando descriviamo un programma, quanto tempo ci mette e quanto spazio utilizza è una domanda importante per capire se il programma è efficiente o meno. Anche in senso dei linguaggi di programmazione per capire se usiamo quello giusto per il problema che stiamo cercando di risolvere. Chiaramente è un qualcosa che in continuo sviluppo perché si evolve in base alla tecnologia che abbiamo a disposizione.

Uno dei pionieri è stato **Hilbert** che si chiedeva se la matematica fosse formalizzabile come insieme finito (non contraddittorio) di assiomi? Godel dimostrò che non è possibile rappresentare la matematica come un insieme finito di assiomi in maniera non contradditoria, dicendo che in ogni sistema formale ci sono proposizioni vere che non possono essere dimostrate all'interno del sistema.

Nel tentativo di rispondere a queste (ed altre) domande si è costruito un modello che permette di comprendere profondamente ilr agionamento computazionale, permettendo di applicarlo ad ogni disciplina. Cosa è calcolabile e cosa non lo è?

1.1.1 Macchina di Turing

Anche Turing si pose questa domanda e propose la **Macchina di Turing** come modello di calcolo. Una sola macchina (programmabile) per tutti i problemi. La macchina è universale (interprete):

$$Init(P,x) = \begin{cases} P(x) & \text{se } P \text{ è un programma che termina} \\ \uparrow & \text{se } P \text{ è un programma che non termina} \end{cases}$$

Se un problema è intuitivamente calcolabile, allora esisterà una macchina di Turing (o un dispostivo equivalente, come il computer) in grado di risolverlo, cioè calcolarlo. I modelli equivalenti possono essere:

- Lambda calcolo
- Funzioni ricorsive

- Linguaggi di programmazione (Turing completi)

I problemi non calcolabili sono infinitamente più numerosi di quelli calcolabili.

1.1.2 Limiti dell'informatica

L'informatica ha più limiti di quanto si possa pensare, definiti dall'equivalenza di Turing e l'incompletezza di Godel che ci dicono che non possiamo risolvere tutti i problemi. Ci sono anche limiti fisici e tecnologici come:

- Dati non osservabili (teorema di Shannon)
- Dati non controllabili (velocità della luce)

1.2 Basi di logica

Alcune nozioni di logica che ci serviranno in seguito:

- **Linguaggio del primo ordine:**
  - Simboli relazionali $(p, q, ...)$
  - Simboli di funzione $(f, g, ...)$
  - Simboli di costante $(c, d, ...)$
- **Simboli logici:**
  - Parentesi (,) e virgola
  - Insieme numerabile di variabili $(v, x, ...)$
  - Connettivi logici $(\neg, \wedge, \vee, \rightarrow, \leftrightarrow)$
  - Quantificatori $(\forall, \exists)$
- **Termini:**
  - Variabili
  - Costanti
  - f simbolo di funzione m-ario $t_1, t_2, \ldots, t_m$ termini, allora $f(t_1, t_2, \ldots, t_m)$ è un termine.
- **Formula atomica:** p simbolo di relazione n-ario, $t_1, t_2, \ldots, t_n$ termini, allora $p(t_1, t_2, \ldots, t_n)$ è una formula atomica.
- **Formula:**
  - Formula atomica
  - φ formula, allora $\neg\varphi$ è una formula
  - φ e ψ formule, allora $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$, $(\varphi \leftrightarrow \psi)$ sono formule.
  - φ formula e v variabile, alloar e $\forall v.\varphi$ e $\exists v.\varphi$ sono formule.

1.3 Nozioni sugli insiemi

- $x \in A$ signfiica che x è un elemento dell'insieme A
- $\{x|P(x)\}$ si identifica insieme costuito dagli x che soddisfano la proprietà (o predicato) $P(x)$
- $A \subseteq B$ significa che A è un sottoinsieme di B se ogni elemento di A è anche in B
- $\mathcal{P}(S)$ denota l'insieme delle parti di S, ovvero l'insieme di tutti i sottoinsiemi di S ($\mathcal{P}(S) = \{X|X \subseteq S\}$)
- $A \backslash B = \{x|x \in A \wedge x \notin B\}, A \cup B = \{x|x \in A \vee x \in B\}, A \cap B = \{x|x \in A \wedge x \in B\}$
- $|A|$ denota la cardinalità di A, ovvero il numero di elementi in A.
- $\bar{A}$ denota il complemento di A, ovvero $x \bar{\in} A \leftrightarrow x \notin A$

1.4 Nozioni sulle relazioni

- Prodotto cartesiano: $A_1 \times A_2 \times \cdots \times A_n = \{\langle a_1, a_2, \ldots, a_n\rangle | a_1 \in A_1, \ldots, a_n \in A_n\}$
- Una **RELAZIONE** (binaria) è un sottoinsieme del prodotto cartesiano di (due) insiemi; dati A e B, $R \subseteq A \times B$ è una relazione su A e B
  - **Riflessiva:** $\forall a \in S$ si ha che aRa
  - **Simmetrica:** $\forall a, b \in S$ se aRb allora bRa
  - **Antisimmetrica:** $\forall a, b \in S$ se aRb e bRa allora $a = b$
  - **Transitiva:** $\forall a, b, c \in S$ se aRb e bRc allora aRc
- Per ogni relazione $R \subseteq S \times S$ la chiusura transitiva di R è il più piccolo insieme R^* tale che $\langle a, b\rangle \in R \wedge \langle b, c\rangle \in R \rightarrow \langle a, c\rangle \in R^*$
- Una relazione è detta **totale** su S se $\forall a, b \in S$ si ha che $aRb \vee bRa$
- Una relazione R di *di equivalenza* è una relazione binaria riflessiva, simmetrica e transitiva.
- Una relazione binaria $R \subseteq S \times S$ è un **pre-ordine** se è riflessiva e transitiva.
- R è un ordine parziale se è un pre-ordine antisimmetrico.
- $x \in S$ è **minimale** rispetto a R se $\forall y \in S.y \not R x$ (ovvero $\neg(yRx)$)
- $x \in S$ è **minimo** rispetto a R se $\forall y \in S.xRy$
- $x \in S$ è **massimale** rispetto a R se $\forall y \in S.x \not R y$ (ovvero $\neg(xRy)$)
- $x \in S$ è **massimo** rispetto a R se $\forall y \in S.yRx$

1.5 Nozioni sulle funzioni

- Una relazione f è una **funzione** se $\forall a \in A$ esiste uno ed un solo $b \in B$ tale che $(a, b) \in f$
- A dominio e B codominio di f. Il range di f è l'insieme di tutti i valori che f può assumere.
- f è **iniettiva** se $\forall a_1, a_2 \in A$ se $a_1 \neq a_2$ allora $f(a_1) \neq f(a_2)$
- Se $f : A \mapsto B$ è sia iniettiva che suriettiva allora è **biiettiva** e quindi esiste $f^{-1} : B \mapsto A$

2 Funzioni calcolabili

Un insieme è a tutti gli effetti una proprietà che dato un oggetto stabilisce se esso all'interno di insieme o no.

$$\text{Problemi} \equiv \text{Funzioni} \quad f : \mathbb{N} \to \mathbb{N}$$

Ci chiediamo se questa funzioni siano tutte calcolabili (intuitivamente). Da il teorema che abbiamo citato nei precedenti paragrafi (quello dell'incompletezza) sappiamo che non è così. Cerchiamo di vedere insiemisticamente perché questo è giustificato.

Definizione 2.1: Intuitivamente Calcolabile

Qualcosa che è **intuitivamente calcolabile** è qualcosa è che riusciamo a descrivere attraverso un algoritmo, ovvero una sequenza finita di passi discreti elementari.

La funzione di tipo:

$$f : \mathbb{N} \to \mathbb{N}$$

è un **insieme** di associazioni input-output.

Esempio 2.1

$$\begin{aligned} f = \text{quadrato} &= \{(0,0), (1,1), (2,4), (3,9), (4,16), ...\} \\ &= \{(x, x^2) \mid n \in \mathbb{N}\} \end{aligned}$$

Quindi f è un insieme di coppie in $\mathbb{N} \times \mathbb{N}$. Quindi $|\mathbb{N} \times \mathbb{N}| = |\mathbb{N}|$ (cardinalità di $\mathbb{N} \times \mathbb{N}$ Quindi

$$f \subseteq \mathbb{P}(\mathbb{N} \times \mathbb{N}) = \mathbb{P}(\mathbb{N})$$

Per esempio se:

$$A = \{1, 2, 3\} \text{ allora } \mathbb{P}(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}\}$$

dove $|\mathbb{P}(A)| = 2^{|A|}$.

$$|\mathbb{N}| = \omega < |P(\mathbb{N})|) = |\mathbb{R}|$$

e quindi l'insieme delle funzioni **non è numerabile**.

2.1 Quale funzioni numerabili ci sono?

Σ =alfabeto finito di simboli che uso per il programma/algoritmo

$$\Sigma = s_1, s_2, s_3, \dots$$

quindi un programma non è nient'altro che un sottoinsieme finito di Σ^* (tutte le stringhe finite che posso formare con l'alfabeto).

$$\Sigma = a, b, c$$

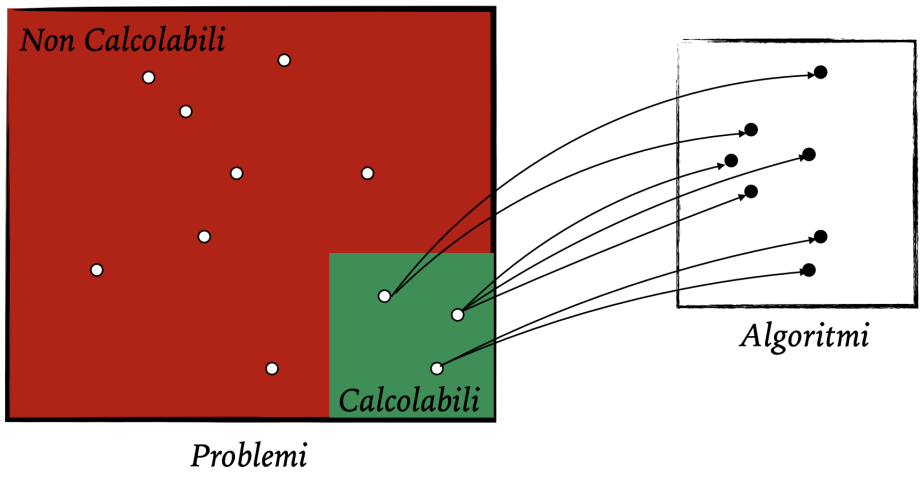
$$\Sigma^* = \{\emptyset, a, b, c, ab, ba, ac, cd, bc, cb, \dots\}$$

in questo caso, la sequenza di simboli in Σ^* è numerabile, perché

$$|\Sigma^*| = |\mathbb{N}|$$

$$|\text{Programmi in } \mathbb{N}| = |\Sigma^*| = |\mathbb{N}|$$

e di conseguenza l'insieme dei programmi è numerabile. Una veloce constatazione che possiamo fare è vedere quindi che l'insieme delle funzioni calcolabili è numerabile, perché ogni funzione calcolabile è associata ad almeno un programma che la calcola.



3 Principio di induzione

Il principio di induzione ha senso solo su insieme infiniti e serve per dimostrare che una proprietà vale per tutti gli elementi. Esistono due metodi di induzione:

- Induzione matematica
- Induzione strutturale

Tratteremo nello specifico caso in questo corso **l'induzione matematica.** Partiamo da un insieme A infinito con una relazione $< : (A, <)$ è una relazione d'ordine non riflessiva perché non è minore stretto. Utilizziamo l'induzione matematica e quindi $A = \mathbb{N}$ e $<$ è l'ordinamento stretto tra numeri. Una relazione d'ordine deve essere *ben fondata* vuol dire che non esistono catene discendenti infinite. Una catena discendente è una sequenza infinita di elementi

$$a_0 > a_1 > a_2 > a_3 > \dots$$

In un tipo di relazione riflessiva è sicuramente non ben fondata perché posso fare continuare ad inserire lo stesso numero all'infinito.

$$m \text{ minimale in } A \ : \ b \in A \text{ è minimale se } \forall b' < b . b' \notin A$$

Esempio 3.1

Se prendiamo $\{1, 2, 3\}$ con relazione d'ordine di contenimento allora esistono più minimali come $\{1, 2\}$ o $\{2, 3\}$. Quindi se $A = \mathbb{N} \implies \exists b$ minimo $\forall x \subseteq \mathbb{N}$.

Quindi preso A insieme ben fondato con ordinamento $<$ allora: π proprietà definita sugli elementi di

$$A : \pi \subseteq A \text{ allora } \forall a \in A . \pi(a) \iff \forall a \in A . [[\forall b < a . \pi(b)] \to \pi(a)]$$

Se dimostriamo π per ogni elemento più piccolo di a allora π vale per a.

$$Base_A = \{a \in A | a \text{ minimale}\}$$

Quindi

$$\overbrace{\forall A \in Base_A \ . \ \pi(a)}^{\text{Base}} \ \underbrace{\forall a \in A . Base_A}_{\text{passo induttivo}} . \ \overbrace{\forall b < a . \pi(b)}^{\text{ipotesi induttiva}} \ \to \underbrace{\pi(a)}_{\text{tesi}}$$

Esempio 3.2

Prendiamo come esempio il seguente enunciato:

$$\forall n \in \mathbb{N}, \ \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Base: $n = 1$

$$A = \mathbb{N}\backslash\{0\} \;\; Base_A = \{1\} \rightarrow \sum_{i=1}^{1} i = 1$$

$$\sum_{i=1}^{1} i = 1$$
$$= n(n+1)/2$$
$$= \frac{1(1+1)}{2} = 1$$

Caso base dimostrato.

Passo induttivo: prendo $n \in \mathbb{N}$ e applico l'ipotesi induttiva: per ogni $k < n$ vale la tesi.

$$\text{Tesi da dimostrare: } \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n-1} i + n$$

Sappiamo sicuramente che $n - 1 < n$ e per ipotesi induttiva:

$$\sum_{i=1}^{n-1} i + n = \frac{(n-1)(n-1+1)}{2} + n = \frac{n(n-1)}{2} + n =$$

$$\frac{n(n-1) + 2n}{2} = \frac{n(n-1+2)}{2} = \frac{n(n+1)}{2} \quad \square$$

e quindi la tesi vale perché siamo arrivati alla stessa espressione che volevamo dimostrare.

3.1 Linguaggi formali

Definizione 3.1: Linguaggio formale

Un linguaggio formale è un insieme di stringhe composte da simboli in un alfabeto finito Σ.

Σ^* denota il linguaggio di tutte le stringhe dell'alfabeto Σ, se Σ non è vuota allora Σ^* è infinito e numerabile. Solitamente un linguaggio formale $\mathcal{L}$ è un sottoinsieme di Σ^* tipicamente infiniti ma non è necessario:

$$\mathcal{L} \subseteq \Sigma^*$$

I linguaggi finiti sono sicuramente regolari perché *posso sempre costruire un automa a stati finiti* che li riconosce.

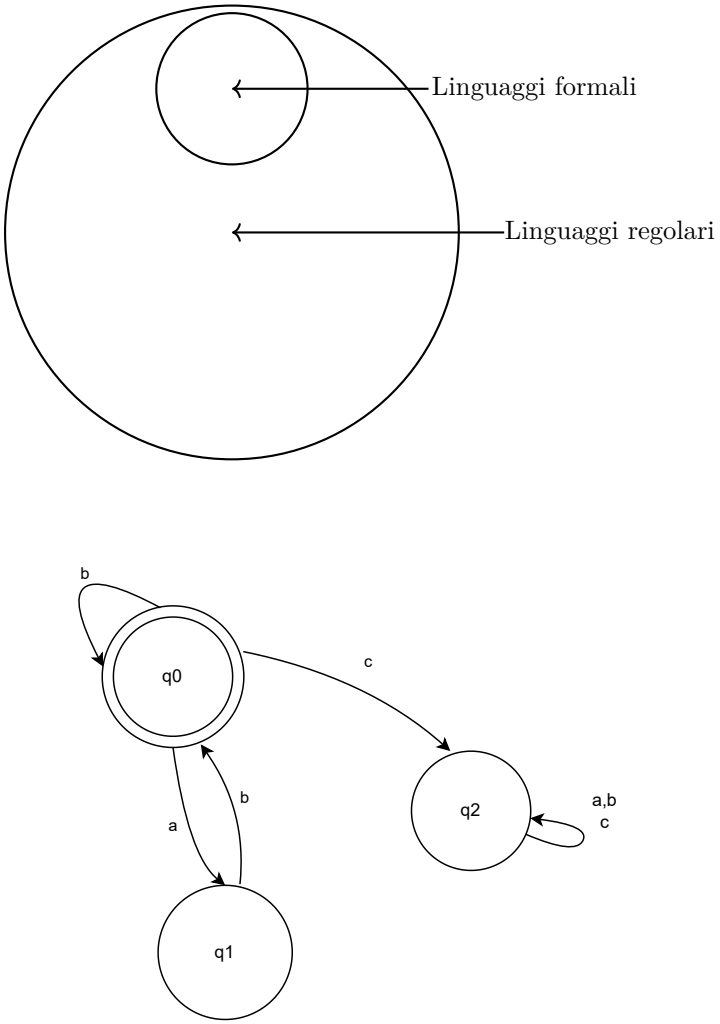


Figure 1: Esempio di automa a stati finiti

Come abbiamo detto in precedenza, una funzione è *calcolabile* se possiamo pensare ad un algoritmo per calcolarla. Tra le funzioni calcolabili, ci sono quelle totali, ovvero quelle che terminano per ogni input.

Esempio 3.3

Pensiamo ad una funzione $f \subseteq \mathbb{N} \times \mathbb{N}$, la possiamo descrivere come associazione di coppie:

$$f(0) = 1, f(1) = 1, f(2) = 2, f(3) = 3, f(4) = 5, f(5) = 8, \ldots$$

Tuttavia servirebbe scrivere una quantità infinita di coppie per descrivere la funzione.

Quindi proviamo a farlo ma stavolta ricorsivamente.

$$\begin{cases} f(0) = 1 = f(1) \\ f(x+2) = f(x+1) + f(x) \end{cases}$$

3.2 Funzioni e insiemi

In realtà ci sta un forte legame tra funzioni e insiemi perché per esempio dovessimo prendere una funzione $f : \mathbb{N} \to \mathbb{N}$, questa funzione può essere vista come un linguaggio $L_f = \{1^{f(x)} | x \in \mathbb{N}\}$ che è l'insieme delle stringhe composte da $f(x)$ simboli 1. Il nostro linguaggio $\Sigma = \{1\}$ e quindi il linguaggio è un sottoinsieme di Σ^*. Il nostro linguaggio ci può dire se un oggetto in input fa parte o no dell'insieme. Parliamo di **insiemi** invece che di funzioni dove gli elementi dell'insieme dipendono dal calcolo della funzione.

Esempio 3.4

Se io dovessi scrivere una funzione costante del tipo $f(x) = 2$. In questo caso riconosciamo che il linguaggio L_f è finito perché

$$L_f = \{1\}$$

L'unica combinazione possibile è una singola stringa. Il linguaggio si dice finito che è sottoinsieme di linguaggi totali.

Esempio 3.5

In questo caso la funzione $f(x) = 2x$ è una funzione lineare. Ho bisogno di una memoria finita. Mi sposto tra queste informazioni per determinare se la stringa in input fa parte o no del linguaggio. Questa funzione è anche detta **regolare**.

Esempio 3.6

$f(\sigma) = \sigma_{\text{reverse}}$ quindi per esempio:

$$f(abc) = cba$$

Tuttavia per questa funzione non mi basta più una memoria finita ma illimitata essendo che non posso sapere a priori quanta memoria abbia bisogno la funzione (è sufficiente uno stack). Questo tipo di funzione è detta anche **context free**.

Esempio 3.7

Nel caso di $f(x) = x^2$ sappiamo sicuramente che possiede un output preciso ma avrei bisogno di una memoria illimitata per rappresentarla.

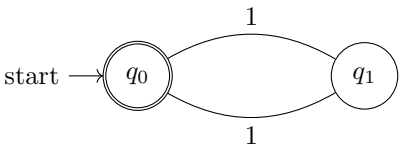
4 Linguaggi Regolari

4.1 Automa a stati finiti determinsitici (DFA)

Quando parliamo di memoria è facilmente codificabile in termini di stati. Dal punto di vista grafico possiamo rappresentarli come nodi collegati da archi. Prendiamo per esempio il linguaggio L_f di prima:

$$L_f = \{1^{2n} \mid n \in \mathbb{N}\}$$

$$\Sigma = 1$$



Definizione 4.1: Automa a stati finiti

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

è un automa a stati finiti deterministico dove:

- Q è un insieme finito di stati (ogni stato rappresenta "un informazione")
- Σ è un alfabeto finito di simboli
- $\delta : Q \times \Sigma \to Q$ è la funzione di transizione (descrive come evolve il calcolo a partire dallo stato raggiunto e dal simbolo letto)
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme di stati finali (o di accettazione)

$\Sigma \backslash Q$	q_0	q_1
1	q_1	q_0

Ci sta poi la funzione $\hat{\delta} : Q \times \Sigma^* \to Q$ che estende la funzione di transizione e descrive lo stato

che raggiungiamo leggendo una sequenza di simboli.

$$\begin{cases} \hat{\delta}(q, \epsilon) = q \\ \hat{\delta}(q, wa) = \hat{\delta}(\delta(q, w), a) \end{cases} \qquad w \in \Sigma^*, a \in \Sigma$$

anche chiamata chiusura transitiva di δ. Quindi mi permette di calcolare lo stato raggiunto leggendo una stringa di simboli.

4.2 Come si dimostra che un linguaggio è regolare?

Esempio 4.1

Prendiamo come esempio il linguaggio:

$$L = \{\sigma \mid \ \sigma \text{ contiene almeno due } 1\}$$

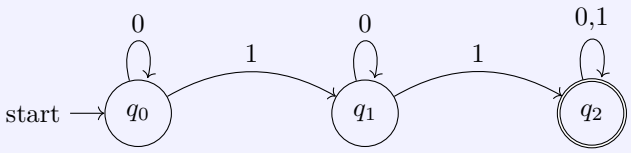
$$\Sigma = \{0, 1\}$$

Per esempio:

$$110, 101, 111, 011, 1001 \in L$$

$$0, 00, 000, 01, 10, 0000 \notin L$$

Quindi intuitivamente come possiamo definire l'automa?



Un linguaggio L è riconosciuto da M (Automa a stati finiti deterministico o DFA) se $L = L(M)$ dove $L(M)$ è il linguaggio di n definito come:

$$L(M) = \{\sigma \in \Sigma^* \mid \hat{\delta}(q_0, \sigma) \in F\}$$

Sono tutte le stringhe che partendo da q_0 (stato iniziale) raggiunge uno stato finale.

Definizione 4.2

Per dimostrare che L è regolare dobbiamo costruire M (almeno un M) e dimostrare che $L(M) = L$.

$$L = L(M) \equiv L \subseteq L(M) \text{ e } L(M) \subseteq L$$

1.

$$L \subseteq L(M) \equiv \sigma. \in L \to \sigma \in L(M)$$

$$\sigma \in L \to \hat{\delta}(q_0, \sigma) \in F$$

2.

$$L(M) \subseteq L \equiv \sigma \in L(M) \to \sigma \in L$$

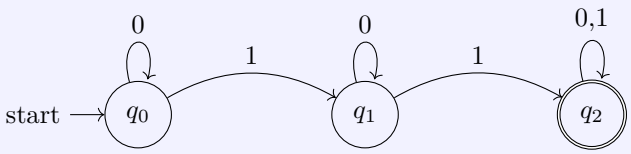
$$\hat{\delta}(q_0, \sigma) \in F \to \sigma \in L \equiv$$

$$\sigma \notin L \to \hat{\delta}(q_0, \sigma) \notin F$$

Dimostriamo per induzione sulla lunghezza delle stringhe $\sigma \in \Sigma^*$ che se $\sigma \in L$ allora $\hat{\delta}(q_0, \sigma) \in F$ e $\sigma \notin L$ allora $\hat{\delta}(q_0, \sigma) \notin F$.

Esempio 4.2

$$L = \{\sigma \mid \ \sigma \text{ contiene almeno due } 1\}$$



Dimostriamo per induzione sulla lunghezza delle stringhe $\sigma \in \Sigma^*$ che se $x \in L$ allora $\hat{\delta}(q_0, x) \in F$ e se $x \notin L$ allora $\hat{\delta}(q_0, x) \notin F$.

$|\sigma| = 0$ non è **mai** sufficiente come base, ma è eventualmente la base **solo** per una delle due dimostrazioni. Bisogna quindi prendere la lunghezza più piccola che permette di avere sia $\sigma \in L$ che $\sigma \notin L$, in questo caso è $|\sigma| = 2$. Per ogni σ tale che $|\sigma| < 2 \quad \sigma \notin L$ perchè non può contenere due 1 e non è riconosciuta da M dove il primo stato finale è raggiunto leggendo almeno due simboli.

$$\varepsilon \in L \quad \varepsilon \notin L$$

- **Base**: Controlliamo ogni stringa di lungheezza minima nel linguaggio per provare il caso base

$$\begin{cases} \sigma & = 11 \in L \ \text{ e } \ \hat{\delta}(q_0, 11) = q_2 \in F \\ \sigma & = 10 \notin L \ \text{ e } \ \hat{\delta}(q_0, 10) = q_1 \notin F \\ \sigma & = 01 \notin L \ \text{ e } \ \hat{\delta}(q_0, 01) = q_1 \notin F \\ \sigma & = 00 \notin L \ \text{ e } \ \hat{\delta}(q_0, 00) = q_0 \notin F \end{cases}$$

- **Passo induttivo**: Dimostriamo l'**ipotesi induttiva**, cioè la tesi con un limite

fissato:

$$\forall \sigma \in \Sigma^* \ . \ |\sigma| \leq n \ . \ \begin{cases} \sigma \in L & \Rightarrow \hat{\delta}(q_0, \sigma) \in F \\ \sigma \notin L & \Rightarrow \hat{\delta}(q_0, \sigma) \notin F \end{cases}$$

Vogliamo dimostrare se $|\sigma| = n + 1$ (la successiva stringa che posso considerare) allora vale $\begin{cases} \sigma \in L \Rightarrow \hat{\delta}(q_0, \sigma) \in F \\ \sigma \notin L \Rightarrow \hat{\delta}(q_0, \sigma) \notin F \end{cases}$

Dimostrazione:

$$|\sigma| = n + 1 \rightarrow \sigma = \sigma' 1 \ \vee \ \sigma = \sigma' 0$$

- Supponiamo che σ appartiene al linguaggio e termini con 1:

$$\sigma \in L \wedge \sigma = \sigma' 1 \rightarrow$$

  * Se $\sigma' \in L$ applico l'ipotesi induttiva:

$$\hat{\delta}(q_0, \sigma') = q_2$$

$$\begin{aligned} \hat{\delta}(q_0, \sigma) &\overset{\sigma = \sigma' 1}{=} \hat{\delta}(q_0, \sigma' 1) \\ &= \delta(\hat{\delta}(q_0, \sigma'), 1) \\ &= \delta(q_2, 1) = q_2 \in F \end{aligned}$$

  * Se $\sigma' \notin L$ allora σ' contiene esattamente un 1:

$$\hat{\delta}(q_0, \sigma') = q_1$$

$$\hat{\delta}(q_0, \sigma' 1) = \delta(q_1, 1) = q_2$$

- Supponiamo che σ appartiene al linguaggio e termini con 0:

$$\sigma \in L \wedge \sigma = \sigma' 0$$

  Per definizione di L abbiamo che

$$\sigma \in L \wedge \sigma = \sigma' 0 \Rightarrow \sigma' \in L$$

  Dimostriamo l'ipotesi induttiva:

$$\hat{\delta}(q_0, \sigma') = q_2$$

allora

$$\begin{aligned}\hat{\delta}(q_0, \sigma) &= \hat{\delta}(q_0, \sigma' 0) \\ &= \delta(\hat{\delta}(q_0, \sigma'), 0) \\ &= \delta(q_2, 0) = q_2 \in F\end{aligned}$$

– Supponiamo che σ non appartiene al linguaggio e termini con 1:

$$\sigma \notin L \land \sigma = \sigma' 1 \text{ (ha esattamente un 1)}$$

$$\Downarrow$$

$$\sigma \notin L \text{ non ha 1}$$

– Supponiamo che σ non appartiene al linguaggio e termini con 0:

$$\sigma \notin L \land \sigma = \sigma' 0$$

$$\Downarrow$$

$$\sigma' \notin L$$

Esempio 4.3 (Esercizio da esame)

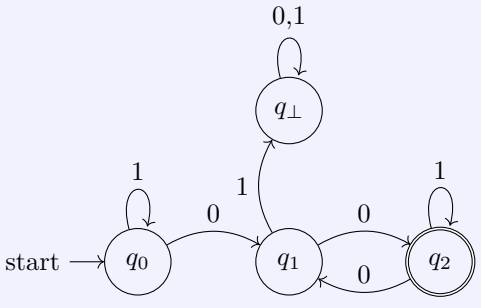
$$L = \{\sigma \in \Sigma^* \mid \exists n \geq 1 \,.\, \sigma = 0^n \to n \geq 2\}$$

$$\Sigma = \{0, 1\}$$

Ovvero ogni sequenza di 0 nella stringa deve essere di lunghezza pari. Per esempio:

$$101 \notin L, 1111 \in L$$

$$1001 \in L, 000 \notin L, 0000 \in L, 010101 \notin L$$



- q_0: non abbiamo letto 0
- q_1: rappresenta una sequenza di 0 consecutivi di lunghezza dispari
- q_2: sequenza di 0 di lunghezza pari.

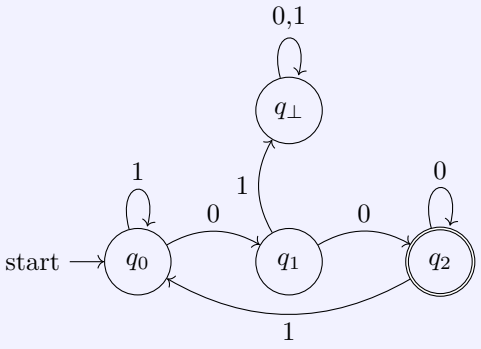
Tesi: $\begin{cases} \sigma \in L \text{ e non contiene } 0 \ \rightarrow \hat{\delta}(q_0, \sigma) = q_0 \\ \sigma \in L \text{ e contiene } 0 \ \rightarrow \hat{\delta}(q_0, \sigma) = q_2 \\ \sigma \notin L \text{ e contiene una sequenza finali di } 0 \ \rightarrow \hat{\delta}(q_0, \sigma) = q_1 \\ \sigma \notin L \text{ e contiene una sequenza dispari di } 0 \text{ seguiti da un } 1 \ \rightarrow \hat{\delta}(q_0, \sigma) = q_\perp \end{cases}$

Esempio 4.4

$$L = \{\sigma \in \Sigma^* \mid \exists n \geq 1 \,.\, \sigma = 0^n \rightarrow n \geq 2\}$$

$$\Sigma = \{0, 1\}$$

Se σ non contiene 0 allora $\sigma \in L$.



- q_0: non contiene 0 oppure tutte le sequenze di 0 sono lunghe almeno 2.
- q_1: esattamente uno 0.
- q_2: almeno due 0.

Tesi: $\begin{cases} \sigma \in L \text{ e } \sigma = \sigma'1 \rightarrow \hat{\delta}(q_0, \sigma) = q_0 \\ \sigma \in L \text{ e } \sigma = \sigma'0 \rightarrow \hat{\delta}(q_0, \sigma) = q_2 \\ \sigma \notin L \text{ e } \sigma = \sigma'0 \text{ dove l'ultima seq di 0 è esattamente lunga 1} \rightarrow \hat{\delta}(q_0, \sigma) = q_1 \\ \sigma \notin L \text{ e } \sigma \text{ contiene una sequenza lunga 1 di 0} \rightarrow \hat{\delta}(q_0, \sigma) = q_\perp \end{cases}$

4.3 Automi a stati finiti non deterministici (NFA)

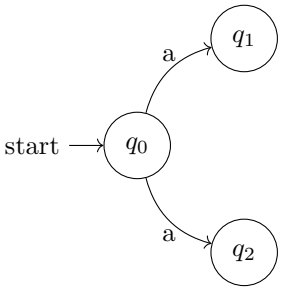
Definizione 4.3

Un automa a stati finiti non deterministico è una 5-upla:

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

dove:

- Q è un insieme finito di stati
- Σ è un alfabeto finito di simboli
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme di stati finali (o di accettazione)
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ è la funzione di transizione dove per ogni coppia di stato-simbolo ho un insieme di stati potenzialmente raggiungibili.



$$\delta(q_0, a) = \{q_1, q_2\} \subseteq Q$$

$$\varnothing \in \mathcal{P}(Q)$$

- È possibile che non esistano coppie associate al vuoto
- Non è obbligatorio avere un arco uscente per ogni simbolo di Σ.

Definiamo $\hat{\delta}$: $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$

$$\begin{cases} \hat{\delta}(q, \epsilon) = \{q\} \\ \hat{\delta}(q, wa) = \bigcup_{p \in \hat{\delta}(q,w)} \delta(p, a) \end{cases}$$

Linguaggio riconosciuto:

$$L(N) = \{\sigma \in \Sigma^* \mid \hat{\delta}(q_0, \sigma) \cap F \neq \varnothing\}$$

Teorema 4.3.1 Teorema di Rabin-Scott

Se abbiamo $N = (Q, \Sigma, \delta, q_0, F)$ un NFA allora esiste M DFA tale che $L(N) = L(M)$.

Quindi se M DFA allora N è un particolare NFA.

Proof. Preso $N = (Q, \Sigma, \delta, q_0, F)$ costruiamo $M = (Q', \Sigma, \delta', q_0', F')$ dove:

$$Q' = \mathcal{P}(Q)$$

$$Q = \{q_0, q_1, q_2\}$$

$$Q' = \{\varnothing, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$$

$$q_0' = \{q_0\}$$

$$F = \{q_2\}$$

$$F' = \{\underbrace{P \subseteq Q}_{P \in \mathcal{P}(Q)} \mid P \cap F \neq \varnothing\}$$

$$F' = \{\{q_2\}, \{q_1 q_2\}, \{q_0 q_2\}, \{q_0 q_1 q_2\}\}$$

$$\delta'(P, a) = \bigcup_{q \in P} \delta(q, a) \in \mathcal{P}(Q)$$

$$\begin{aligned} \delta'(q_5', 1) &= \delta(q_1, 1) \cup \delta(q_2, 1) \\ &= \{q_0, q_2\} \cup \{q_0, q_1, q_2\} \\ &= \{q_0, q_1, q_2\} \end{aligned}$$

Dimostriamo:

1. $\hat{\delta}' q_0', \sigma = \hat{\delta}(q_0, \sigma)$
2. $\sigma \in L(N)$ sse $\sigma \in L(M)$

1. Per induzione su $|\sigma|$:
   Se $\sigma = \epsilon$ allora: $\hat{\delta}'(q_0', \epsilon) = q_0' = \{q_0\} = \hat{\delta}(q_0, \epsilon)$ per le definizioni.
   Se $\sigma = \sigma' a$: $\hat{\delta}'(q_0', \sigma' a) = \hat{\delta}(\hat{\delta}'(q_0, \sigma'), a)$. Per ipotesi induttiva:

$$\hat{\delta}'(q_0', \sigma') = \hat{\delta}(q_0, \sigma')$$

2. $\sigma \in L(N)$ sse $\sigma \in L(M)$

$$\begin{aligned}\sigma \in L(N) &\Leftrightarrow \hat{\delta}(q_0, \sigma) \cap F \neq \varnothing \\ &\Leftrightarrow \hat{\delta}'(q_0', \sigma) \cap F \neq \varnothing \\ &\Leftrightarrow \hat{\delta}'(q_0', \sigma) \in F' \\ &\Leftrightarrow \sigma \in L(M) \text{ def. linguaggio accettato in DFA} \quad \square\end{aligned}$$

Una volta aver raggiunto un risultato di automa a stati finiti deterministico non è detto che sia il migliore che possiamo trovare. Possiamo *eliminare gli stati non raggiungibili* e ottenere un automa equivalente più piccolo (**DFA minimo**).

4.3.1 Automi non deterministici con ε-transizioni (ε-NFA)

Gli ε-NFA sono una variante degli NFA in cui possiamo cambiare stato senza leggere simboli.

$$q_1 \xrightarrow{\varepsilon} q_2$$

Definizione 4.4

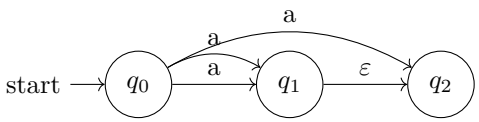
Un ε-NFA è una 5-upla:

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

dove:

- Q è un insieme finito di stati
- Σ è un alfabeto finito di simboli
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme di stati finali (o di accettazione)
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ è la funzione di transizione dove per ogni coppia di stato-simbolo ho un insieme di stati potenzialmente raggiungibili. La ε mi dice che possiamo cambiare stato senza leggere simboli.

Per esempio:



Definiamo ora:

$$\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$$

$$\varepsilon - \text{closure} : Q \to \mathcal{P}(Q)$$

La ε-closure(q) di uno stato q è l'insieme di stati raggiungibili da q seguendo archi etichettati con ε.

$$\varepsilon - \text{closure} : \mathcal{P}(Q) \to \mathcal{P}(Q)$$

$$\begin{cases} \hat{\delta}(q, \epsilon) = \varepsilon\text{-closure}(q) \\ \hat{\delta}(q, wa) = \varepsilon\text{-closure}\left(\bigcup_{p \in \hat{\delta}(q,w)} \delta(p, a)\right) \end{cases}$$

Esempio 4.5

La ε-closure di q' nell'automa sopra è:

$$\varepsilon\text{-closure}(q') = \{q', q''\}$$

Il riconoscimento di un linguaggio è analogo a quello di un NFA:

$$L(N) = \{\sigma \in \Sigma^* \mid \hat{\delta}(q_0, \sigma) \cap F \neq \varnothing\}$$

Teorema 4.3.2

Sia $N = (Q, \Sigma, \delta, q_0, F)$ un ε-NFA. Allora esiste un NFA N' tale che $L(N) = L(N')$.

Ciò vuol dire che l'insieme dei linguaggi riconosciuti da ε-NFA coincide con quello degli NFA, che a sua volta coincide con i linguaggi regolari. Prendiamo $N = (Q, \Sigma, \delta, q_0, F)$ un ε-NFA. Costruiamo NFA $N' = (Q', \Sigma', \delta, q_0', F')$ dove $\delta'(q, a) = \hat{\delta}(q, a)$ e $F' = \begin{cases} F \cup \{q_0\} & \text{se } \varepsilon\text{-closure}(q_0) \cap F \neq \varnothing \\ F & \text{altrimenti} \end{cases}$.

4.4 Espressioni Regolari

Le espressioni regolari sono un modo compatto per rappresentare i linguaggi. Le opreazioni che si possono fare sono:

- Unione: Siano $L_1, L_2 \subseteq \Sigma^*$ linguaggi : $L_1 \cup L_2 = \{\sigma \mid \sigma \in L_1 \text{ oppure } \sigma \in L_2\}$
- Concatenazione: Siano $L_1, L_2 \subseteq \Sigma^*$ linguaggi : $L_1 \cdot L_2 = \{\sigma_1\sigma_2 \mid \sigma_1 \in L_1, \sigma_2 \in L_2\}$ che permette la concatenazione di tutte le stringhe di L_1 con tutte le stringhe di L_2.
- Stella di Kleene: Sia $L \subseteq \Sigma^*$ un linguaggio : $L^* = \bigcup_{n \in \mathbb{N}} L^n$ dove L^n è la concatenazione di L con sè stesso n volte.
$$\begin{cases} L^0 = \{\varepsilon\} \\ L^{n+1} = L^n \cdot L \end{cases}$$

$$L = \{000, 111\}$$

$$L^0 = \{\varepsilon\}$$
$$L^1 = \{000, 111\} = L$$
$$L^2 = \{000000, 000111, 111000, 111111\}$$
$$L^3 = \{000000000, 000000111, 000111000, 000111111, 111000000, 111000111, 111111000, 111111111\}$$
$$\vdots$$
$$L^* = \bigcup_{n \in \mathbb{N}} L^n$$

$$L^+ = \bigcup_{n \geq 0} L^n$$

Definizione 4.5

Σ alfabeto, definiamo per induzione:

- **Caso base:**
  - $\varnothing \subseteq \Sigma^*$ è espressione regolare che rappresenta il linguaggio vuoto.
  - ε è espressione regolare che rappresenta il linguaggio $\{\varepsilon\} \subseteq \Sigma^*$.
  - $a \in \Sigma$ è espressione regolare che rappresenta il linguaggio $\{a\} \subseteq \Sigma^*$.
- **Passo induttivo:**
  - r, s sono espressioni regolari che rappresentono il linguaggio $R \subseteq \Sigma^\alpha$ e $S \subseteq \Sigma^*$ rispettivamente.
    - $(r) + (s)$ è espressione regolare che rappresenta il linguaggio $R \cup S$.
    - $(r)(s)$ è espressione regolare che rappresenta il linguaggio $R \cdot S$.
    - $(r)^*$ è espressione regolare che rappresenta il linguaggio R^*.

Esempio 4.6

Prendiamo come esempio la seguente espressione:

$$1^* + 0^* + (10)^*$$

equivale:

$$\{1^n \mid n \in \mathbb{N}\} \cup \{0^n \mid n \in \mathbb{N}\} \cup \{(10)^n \mid n \in \mathbb{N}\}$$

Un'altra equivalenza interessante è:

$$L = 000, 111 \to (000 + 111)$$

$$L^* = (000 + 111)^*$$

Teorema 4.4.1 Teorema di equivalenza

Dato M DFA $(Q, \Sigma, \delta, q_0, F)$ allora esiste r espressione regolare t.c $L(N) = L(r)$.

$$\text{L regolare} \overset{def}{\Leftrightarrow} \exists \underbrace{M\ DFA}_{L(n)=L} \overset{thm}{\to} \exists r \in ER \mid L(r) = L$$

Teorema 4.4.2

Data r espressione regolare (ER) allora esiste un ε-NFA M tale che $L(r) = L(N)$.

- $\varepsilon \rightsquigarrow q_0, L(N) = \{\varepsilon\}$
- $\varnothing \rightsquigarrow q_0, q_1, L(N) = \varnothing$
- $a \rightsquigarrow q_0 \to q_1, L(N) = \{a\}$
- r, s espressioni regolari per ipotesi induttiva $\exists N_1.L(n_1) = L(r)$ e $\exists N_2.L(n_2) = L(s)$.

$$r\ ER \Rightarrow M\ \varepsilon\text{-closure} \iff M'\ \text{DFA} \iff L(M') \text{ è regolare}$$

Esempio 4.7

$$L = \{\sigma \in \Sigma^* \mid \sigma \text{ contiene almeno due } 1\}$$

per dimostrare che è regolare si costruisce il DFA M e si dimostra ceh $L = L(M)$.

$$r = 0^*10^*10^*$$

dove r non dimostra che L è regolare, perché dovremmo dimostrare $L = L(r)$. $L(r)$ è sicuramente regolare ma dobbiamo **comunque** dimostrare l'uguaglianza insiemistica $L = L(r)$.

4.5 Proprietà di LR (linguaggi regolari)

4.5.1 Proprietà di chiusura

Rispetto a quali operazioni due linguaggi regolari sono chiusi? Questa proprietà è utile perché a volte mi serve studiare un linguaggio complesso come intersezioni di linguaggi più semplici. Operazioni: $*, \cup, \cdot, \cap, \hat{}$.

Teorema 4.5.1

I linguaggi sono chiusi rispetto a stella di Kleene, unione (finita) e concatenazione.

Quindi

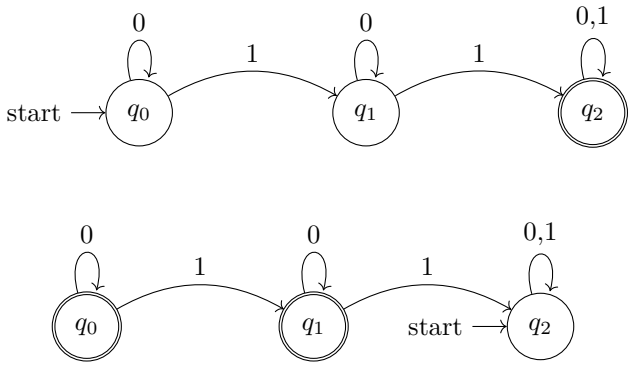
$$L_1, L_2 \text{ reg}$$

$$L^* \text{ è reg. } L_1 \cup L_2 \text{ reg} \quad L_1 L_2 \text{ è regolare}$$

$$\underbrace{a^3}_{\text{reg. finito}} \cdot \{b^n c^m \mid n, m \in \mathbb{N}\}$$

Teorema 4.5.2

I linguaggi regolari sono chiusi rispetto alla complementazione.



L'automa sopra disegnato riconosce il complemento.

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

per le leggi di Morgan, i linguaggi regolari sono chiusi per intersezione (finita).

4.5.2 Proprietà di decidibilità

L'insieme di stringhe accettate da un DFA (linguaggio regolare) DFA con M stati con n stati.

1. $L(M)$ è $\neq \varnothing$ sse accetta almeno una stringa di lunghezza $\leq n$.
2. $L(M)$ è infinito se accetta almeno una stringa lunga l con $n \leq l \leq 2n$.
3. $L(M_1) = L(M_2)$ cazzo piccolissimooo... 8=======D

4.5.3 Esistenza dell'automa minimo

Fornisce strategie per costruire un automa minimo. Sia R relazione su $\Sigma : R \subseteq \Sigma \times \Sigma$. R è una relazione di equivalenza se:

- Riflessiva: $\forall a \in \Sigma . aRa$
- Simmetrica: $\forall a, b \in \Sigma . aRb \Rightarrow bRa$
- Transitiva: $\forall a, b, c \in \Sigma . aRb \wedge bRc \Rightarrow aRc$

Una relazione di equivalenza induce una partizione. $R \subseteq S \times S$ induce una partizione di S dove S_i sono una partizione di S se

$$S = S_1 \cup S_2 \cup \cdots \cup S_n$$

e

- $\forall i, j \; S_i \cap S_j = \varnothing$
- $\forall a, b \in S_i . aRb$
- $\forall a \in S \, \forall b \in S_j . \neg(aRb)$

S_i sono detti classi di equivalenza di R:

$$a \in R \Rightarrow [a]_R = \{b \mid aRb\} \text{ classe di equivalenza di a}$$

$$cRa \Rightarrow [a]_R \equiv [c]_R$$

Torniamo ora ai linguaggi regolari. Consideriamo un linguaggio $L \subseteq \Sigma^*$. Possiamo definire $R_L \subseteq \Sigma^* \times \Sigma^*$ come:

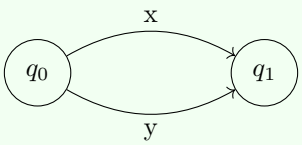
$$x, y \in \Sigma^* \quad xR_Ly \text{ sse } \forall z \in \Sigma^* . xz \text{ sse } yz \in L$$

Quello che diciamo quindi è che due stringhe sono in relazione se per ogni possibile estensione z la loro appartenenza a L è la stessa.

Definizione 4.6: Classe di equivalenza definita per gli automi

Consideriamo $M = (Q, \Sigma, \delta, q_0, F)$ un DFA. Possiamo definire $R_M \subseteq \Sigma^* \times \Sigma^*$ come:

$$x, y \in \Sigma^*.xR_m y \iff \hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$$



Allora R_L **e** R_M **sono relazioni di equivalenza.**

Definizione 4.7: Relazione invariante destra

Una relazione di equivalenza R su Σ^* è invariante a destra se:

$$\forall x, y, z \in \Sigma^* \quad xRy \Rightarrow xzRyz$$

Quindi essere in relazione è invariante rispetto all'estensione della stringa verso destra. R_L **e** R_M **sono relazioni invarianti destra.**

Definizione 4.8: Raffinamento

Siano R e S due relazioni di equivalenza su Σ^*. Diciamo che R raffina S se:

$$\forall x, y \in \Sigma^* \quad xRy \Rightarrow xSy$$

Quindi ogni classe di equivalenza di R è contenuta in una classe di equivalenza di S. Il numero di classi di equivalenza di R è maggiore del numero di classi di equivalenza di S.

Definizione 4.9

Dato un DFA $M = (Q, \Sigma, \delta, q_0, F)$ ogni stato definisce un linguaggio.

$$q \in Q : L_q = \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) = q\}$$

R si dice di indice finito se il numero di classi di equivalenza indotte è finito.

Teorema 4.5.3 Teorema di Nyhill-Nerode

I seguenti enunciati sono equivalenti:

1. $L \subseteq \Sigma^*$ è regolare ovvero esiste un DFA M t.c. $L(M) = L$.
2. L è l'unione di classi di equivalenza indotte da una relazione di equivalenza R invariante destra con indice finito.
3. R_L è di indice finito.

Dimostriamo che:

$$(1) \overset{R_m}{\Longrightarrow} (2) \overset{R \text{ raffina } R_L}{\Longrightarrow} (3) \overset{\text{costruisce } M}{\Longrightarrow} (1)$$

Proof. (1) $\Rightarrow$ (2)
Ipotesi: L linguaggio riconosciuto da $M = (Q, \Sigma, \delta, q_0, F)$ DFA.

$$L = L(M)$$

Tesi: $\exists R$ relazione di equivalenza invariante destra di indice finito t.c. L è l'unione di classi di equivalenza di R. Prendiamo $R = R_M \quad xR_m y$ sse $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$.

1. Il numero di classi di equivalenza di R_M è uguale al numero di stati $|Q|$ ma per definizione Q è finito quindi R_M è di indice finito.
2. R_M è invariante destra:

$$\begin{aligned} L = L(M) &= \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) \in F\} \\ &= \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) = q \wedge q \in F\} \\ &= \bigcup_{q \in F} \{x \mid \hat{\delta}(q_0, x) = q\} \\ &= \bigcup_{q \in F} L_q \text{ dove } L_q \text{ è la classe di equivalenza di } R_M \end{aligned}$$

Quindi L è l'unione di classi di equivalenza di R_M.

Proof. (2) $\Rightarrow$ (3)
Ipotesi: L è l'unione di classi di equivalenza indotte da una relazione di equivalenza R di indice finito.
Tesi: R_L è di indice finito. Possiamo dimostrare che R raffina R_L. Perché se fosse così allora il numero di classi di R sarebbe maggiore del numero di classi di R_L. Quindi se il numero di classi di R è finito allora sicuramente lo sarà anche R_L. Per dimostrare R raffinamento

di R_L dobbiamo dimostrare che $\forall x, y \in \Sigma^* \quad xRy \Rightarrow xR_Ly$. In maniera equivalente:

$$y \in [x]_R \Rightarrow y \in [x]_{R_L} \equiv [x]_R \subseteq [x]_{R_L}$$

Prendiamo xRy sapendo che R è invariante destra.

$$\forall z \in \Sigma^* \quad xRy \Rightarrow xzRyz$$

Quindi se L è l'unione di classi di equivalenza di R:

$$xRy \quad x \in L \text{ sse } y \in L$$

$$[x]_R \subseteq L \text{ oppure } [x]_R \text{ fuori da } L$$

Quindi:

$$\begin{aligned} xRy &\Longrightarrow x \in L \Longleftrightarrow y \in L \\ &\Longrightarrow \forall z \,.\, xzRyz \Longrightarrow xz \in L \Longleftrightarrow yz \in L \\ &\overset{\text{def } R_L}{\Longrightarrow} xR_Ly \end{aligned}$$

Queste dimostrazioni dato M generico, R_M è un raffinamento di R_L.

Proof. (3) $\Rightarrow$ (1)
Ipotesi: R_L è di indice finito.
Tesi: L è riconosciuto da un DFA M.
Costruiamo M:

- $Q = \{[x]_{R_L} \mid x \in \Sigma^*\}$ è l'insieme delle classi di equivalenza di R_L (sono finite)
- $\Sigma =$ quello di L
- $q_0 = [\varepsilon]_{R_L}$
- $F = \{[x]_{R_L} \mid x \in L\}$
- $\delta(q, a) = \delta([x]_{R_L}, a) = [xa]_{R_L}$ questo è vero perché R_L è invariante destra. δ è una buona definizione perché indipendente dall'elemento che rappresenta la classe.

Ora dobbiamo dimostrare che $L = L(M)$. Dimostrare per induzione che $\hat{\delta}([x], y) = [xy]$:

$$\hat{\delta}(q_0, x) = \hat{\delta}([\varepsilon]_{R_L}, x) = [x]_{R_L}$$

$$x \in L(M) \text{ sse } \hat{\delta}(q_0, x) \in F$$
$$\text{sse } \hat{\delta}([\varepsilon]_{R_L}, x) \in F$$
$$\text{sse } [x]_{R_L} \in F$$
$$\text{sse } x \in L \quad \square$$

Quindi $L(M) = L$.

Teorema 4.5.4

Dato L esiste M DFA t.c. $L(M) = L$ e M ha il numero minimo di stati (quello costruito da R_L).

4.5.4 Pumping Lemma per linguaggi regolari

Questo risultato è importante perché fornisce una condizione π necessaria alla regolarità.

$$L \text{ regolare} \implies \pi$$

Questa forma si può riscrivere come:

$$\neg\pi \implies L \text{ non è regolare}$$

Teorema 4.5.5 Pumping Lemma

Sia $L \subseteq \Sigma^*$ un linguaggio regolare. Allora esiste $k \in \mathbb{N}$ t.c. per ogni stringa $s \in L$ con $|s| \geq k$ può essere scritta come:

$$s = uvw$$

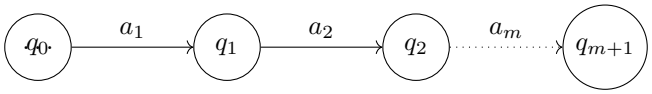
con:

- $|v| > 0$
- $|uv| \leq k$
- $\forall i \in \mathbb{N} \,.\, uv^i w \in L$

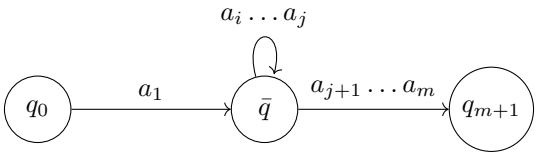
Proof. L regolare se esiste M DFA t.c $L = L(M)$ dove $M = (Q, \Sigma, \delta, q_0, F)$.

$$|Q| = n \in \mathbb{N}, k = n \quad z \in L \text{ t.c. } |z| \geq k$$

$$z = a_1 a_2 \dots a_m \quad m \geq k$$

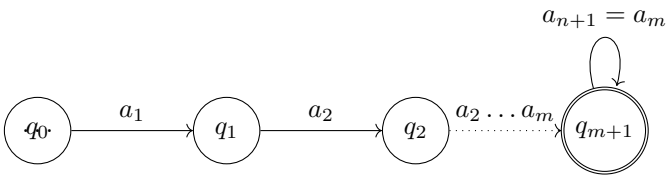


Per leggere m simboli si attraverso $m+1$ stati. Ma $m \geq n \Rightarrow$ attraverso almeno $n+1$ stati. Attraversiamo più stati di quelli in Q ovvero almeno uno stato è ripetuto nel riconoscimento di z. Suppiniamo $\bar{q} \in Q$ sia il primo stato leggendo z che viene ripetuto, in cui si torniamo per riconoscere z.



Non ci sono stati ripetuti perché $\bar{q}$ è il primo per costruzione. u da q_0 a $\bar{q}$, v da $\bar{q}$ a $\bar{q}$, w da $\bar{q}$ a q_m finale. Questa suddivisione è accettabile per il lemma?

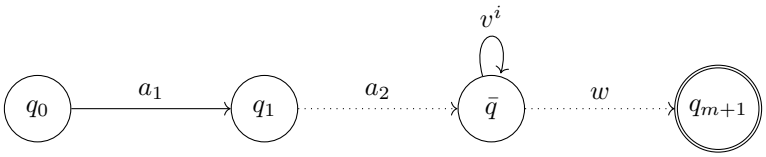
- $|uv| \leq k$



  Per arrivare allo stato finale q_m si attraversano $m-1$ simboli.

$$Q = \{q_0, q_1, \dots, q_{m-1}\} \rightarrow |Q| = m$$

- $|v| > 0$ perché altrimenti non ci sarebbe stato ripetuto, sappiamo che per forza di cose almeno un arco deve uscire dallo stato e ritornare nello stesso stato.
- $\forall i \in \mathbb{N} \,.\, uv^i w \in L$



  perché essendo v un ciclo, ripeterlo i volte mi riporta sempre nello stato $\bar{q}$. $\square$

Ora proviamo a negare il pumping lemma per dimostrare che un linguaggio non è regolare.

- $\exists k \rightsquigarrow \forall k$ quindi la dimostrazione non deve imporre vincoli su k.

- $\forall z \rightsquigarrow \exists z \in L \,.\, |z| \geq k$ quindi costruiamo noi la $z \in L$ di lunghezza k.
- $\exists uvw = z \rightsquigarrow \forall uvw = z$ quindi dobbiamo dimostrare che per ogni possibile suddivisione $(|v| > 0, |uv| \leq k)$
- $\forall i \in \mathbb{N} \rightsquigarrow \exists i \in \mathbb{N}$ quindi dobbiamo trovare un i che rispetti la condizione $uv^i w \notin L$.

Esempio 4.8

Consideriamo il seguente linguaggio:

$$L = \{0^n 1^n \mid n \in \mathbb{N}\}$$

Dimostriamo che L non è regolare tramite il pumping lemma.

$$\forall k \in \mathbb{N} \,.\, \exists z \in L \; |z| \geq k \quad \forall uvw = z \;\; |uv| \leq k \;\; |v| > 0 \quad \exists i \,.\, uv^i w \notin L$$

Fissiamo $k \in \mathbb{N}$ (k non deve avere nessun vincolo):

$$z = 0^k 1^k \in L \quad |z| \geq k$$

Condizioni di appartenenza a L:

$$0^a 1^b \in L \text{ sse } a = b$$

Gli 0 e gli 1 devono avere cardinalità uguale. Quindi una singola suddivisione che va bene posso solo prendere $uv \in 0^k$ perché per ipotesi $|uv| \leq k$. Andiamo ora a definire

$$z_i = uv^i w = 0^{k+(i-1)|v|} 1^k$$

Prendiamo per esempio $i = 0$:

$$0^{k+(0-1)|v|} 1^k = 0^{k-|v|} 1^k$$

e quindi abbiamo effettivamente tolto l'interezza di v.
Per $i = 1$:

$$0^{k+(1-1)|v|} 1^k = 0^k 1^k$$

Per $i = 3$ (ripete tre volte v):

$$0^{k+(3-1)|v|} 1^k = 0^{k+2|v|} 1^k$$

$$z_i = 0^{k+(i-1)|v|} 1^k$$

In questo caso, dobbiamo rompere la condizione di appartenenza per cui $a \neq b$. Basterebbe ripetere v almeno una volta, quindi prendiamo $i = 2$:

$$z_2 = 0^{k+(2-1)|v|}1^k = 0^{k+|v|}1^k \notin L$$

Perché $k + |v| = k$ solo se $|v| = 0$ ma questo non è possibile perché per definizione $|v|$ deve essere maggiore di zero e quindi $z_2 \notin L$.

Riassumendo il metodo per dimostrare che un linguaggio è o non è regolare:

- L Reg $\rightarrow$ Costruisci automa e si dimostra che $L = L(M)$
- L non Reg $\rightarrow$ usa il Pumping Lemma
  - Scrivere le condizioni di appartenenza
  - Fissa k generico e si sceglie z facendo attenzione ai vincoli di k
  - Cerchiamo $i \in \mathbb{N}$ t.c. $uv^iw \notin L$ (condizioni di appartenenza violate)

5 Linguaggi Context Free

5.1 Grammatiche Context Free

Definizione 5.1

Una grammatica è una quadrupla $G = (V, T, R, S)$ dove:

- V è un insieme finito di variabili (non terminali)
- T è un insieme finito di simboli terminali (disgiunto da V)
- R è un insieme finito di regole di produzione della forma $A \rightarrow \beta$ dove $A \in V$ e $\beta \in (T \cup \Sigma)^*$
- $S \in V$ è il simbolo iniziale

Teorema 5.1.1

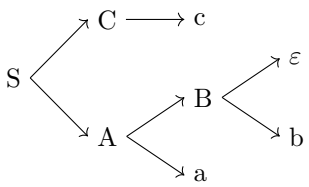
L è un linguaggio CF se esiste una grammatica G t.c. $L = L(G)$.

5.2 Alberi di derivazione

Sia $G = (V, \Sigma, R, S)$ una grammatica CF. Un albero di derivazione (*Parse Tree*) per G:

- Ogni vertice ha un etichetta in $V \cup T \cup \{\varepsilon\}$

- L'etichetta della radice è in V
- Ogni vertice interno (non foglia) ha etichetta in V
- Se un vertice n etichettato con $A \in V$ e i vertici $n_1, n_2, \ldots, n_k$ sono i figli etichettati con $X_1, X_2, \ldots, X_k \in V \cup T \cup \{\varepsilon\}$ allora $A \to X_1 X_2 \ldots X_k \in P$.



Se un vertice ha etichetta ε allora è una foglia ed è l'unico figlio del padre.

Esempio 5.1

Consideriamo la grammatica:

$$E \to 0|1|(\text{E or E})|(\text{E and E})|(\text{not E})$$

Consideriamo la seguente derivazione:

$$E \Rightarrow ((0 or 1) and (not 0))$$

che è la stringa generata dalla grammatica descritta dall'albero

Teorema 5.2.1

Sia $G = (V, T, P, S)$ CF. Allora $S \Rightarrow^* \alpha \in T^*$ sse esiste un albero di derivazione con radice etichettata con S e foglie etichettate (da sx verso dx) con α

5.3 Ambiguità

Nelle grammatiche a volte, possono capitare delle ambiguità in base a come è scritta.

Esempio 5.2

$$E \to E + E|E * E|0|1|2$$

Fingiamo di prendere la stringa:

$$2 * 0 + 1$$

Il problema della grammatica ha radici aritmiche perché non sappiamo quale operazione fare prima, quindi in base a quale scegliamo, avremo due risultati diversi (1 o 2).

Quindi è utile evitare questa proprietà. Una grammatica è ambigua se esiste almeno una parola α con più di un albero di derivazione con radice S. Il linguaggio generato da una grammatica ambigua è detto inerentemente *ambiguo*.

5.4 Forme normali e minimizzazione delle grammatiche CF

Le grammatiche le cui produzioni rispettono vincoli "di forma" specifici.

- **Forma normale di Chomsky:** Tutte le produzioni hanno una forma: $A \to a$ o $A \to BC$ con $A, B, C \in V$ e $a \in T$
- **Forma di Greibach**: Tutte le produzioni hanno la forma: $A \to a\alpha$ con $a \in T$ e $\alpha \in V^*$

Ogni grammatica può essere riscritta in modo tale che:

1. Ogni simbolo non terminale genera simboli terminale e ogni terminale è generato da almeno un terminale (eliminazione dei simboli inutili)
2. Ogni produzione non è della forma $A \to B$ con $A, B \in V$ (eliminazione delle produzioni unitarie).
3. Se ε non appartiene al linguaggio allora non ci devono essere $A \to \varepsilon$ con $A \in V$ (Se $\varepsilon \in L$ allora $S \to \varepsilon$) (eliminazione ε-produzioni)

5.4.1 Eliminazione dei simboli inutili

$X \in V \cup T$ è utile se essite una derivazione $S \to^* \alpha x \beta \to^* w \in T^*$
Metodo algoritmico per eliminare i simboli che non arrivano a sequenze terminali. $\forall A \in V \ . \ \exists v \in T^* \ . \ A \to^* w$ allora $L(G) \neq 0$ tutti i simboli sono inutili. Quindi prima andremo a togliere i simboli che non portano a simboli terminali. Una volta trovati quelli, andremo a togliere quelli che non sono raggiungibili da S. Tutto il resto sono simboli inutili.
Eliminiamo i simboli non raggiungibili da S:

$$\Gamma(W) = \{X \in V \cup T \mid \exists A \in W \ . \ A \to \alpha x \beta \in P\} \cup S$$

$$\Gamma(\varnothing) = \{S\}$$

Esempio 5.3

Aggiungiamo ciò che possiamo raggiungere da S in un certo numero di passi.

- $S \to a|bC$

- $C \to d|dE$
- $E \to \varepsilon$
- $F \to f$

$$\Gamma(\varnothing) = \{\}$$

$$\Gamma(S) = \{X \in V \cup T \mid S \to \alpha x \beta \in P\} \cup \{S\} = \{a, b, c, S\}$$

$$\Gamma(a, b, c, S) = \{X \in V \cup T \mid C \to \alpha x \beta \in P \text{ o } S \to \alpha x \beta \in P\} = \{a, b, c, S, d, E\}$$

$$\Gamma(a, b, c, d, E, S) = \{a, b, c, d, E, S, \varepsilon\}$$

F è inutile.

Teorema 5.4.1

$\forall G = (V, T, P, S)$ CF esiste una grammatica $G' = (V', T, P', S)$ tale che

$$L(G') = L(G)$$

e G' è senza simboli inutili.

G' si ottiene applicando in sequenza le due trasformazioni viste (nell'ordine visto).

5.4.2 Eliminazione delle ε-produzioni

L'idea è quella di sostituire la transizione con tutto quello che può generare ε ad esempio:

Esempio 5.4

Consieriamo la grammatica:

$$S \to \varepsilon|A|B$$

$$A \to 0|0A0|B$$

$$B \to 1|1B1|A|\varepsilon$$

$A \to B \to \varepsilon$ tutti i simboli non terminali che in un certo nnumero di passi generano ε ovvero $\{S, B, A\}$

$$S \to \varepsilon|A|B$$

$$A \to 0|0A0|\ 00\ |B$$

$$B \to 1|1B1|\ 11\ |A$$

Eliminando $B \to \varepsilon$, 00 e 11 non sarebbero più generabili. e quindi si aggiungono. Possiamo quindi eliminare tutte le ε-produzioni tranne $S \to \varepsilon$ se $\varepsilon \in L$.

5.4.3 Eliminazione delle produzioni unitarie

Una produzione unitaria è una produzione della forma $A \to B$ con $A, B \in V$.

Esempio 5.5

Consideriamo la seguente grammatica:

$$S \to \varepsilon|A|B$$

$$A \to 0|0A0|B|00$$

$$B \to 1|1B1|A|11$$

Guardo S e al posto di A e B inserisco tutte le sue produzioni:

$$S \to \varepsilon|0|0A0|00|1|1B1|11$$

$$A \to 0|0A0|00|1|1B1|11$$

$$B \to 1|1B1|11|0|0A0|00$$

Sostituiamo in tutte le produzioni quello che è producibile dal non terminale a destra della produzione unitaria.

Quando la grammatica è senza simboli inutili, senza ε-produzioni e senza produzioni unitarie possiamo trasformarla nella forma normale di Chomsky.

5.4.4 Riduzione in forma normale di Chomsky

Abbiamo già detto che ogni grammatica CF può essere trasformata in una grammatica equivalente in forma normale di Chomsky dove

$$A \to a \in T \quad \vee \quad A \to BC \quad \text{dove } B, C \in V$$

Per ogni coppia di simboli non terminali creiamo un nuovo simbolo non terminale che li rappresenta. Per esempio:

Esempio 5.6

Consideriamo la seguente grammatica:

$$S \to aAB|aB|b$$

$$A \to b|aS|aa$$

$$B \to a|bbA$$

Creiamo i nuovi simboli:

$$X_a \to a$$

$$X_b \to b$$

Ora riscriviamo la grammatica:

$$S \to X_aAB|X_aB|X_b$$

$$A \to X_b|X_aS|X_aX_a$$

$$B \to X_a|X_bX_bA$$

Ora dobbiamo solo risolvere le produzioni con più di due simboli a destra. Per esempio in $S \to X_aAB$ creiamo un nuovo simbolo C:

$$C \to AB$$

Quindi riscriviamo:

$$S \to X_aC|X_aB|X_b$$

$$C \to AB$$

E così via per tutte le produzioni con più di due simboli a destra.

Esempio di forma normale di Chomsky:

$$\begin{aligned}
S &\to \varepsilon \\
S &\to 0 \mid 1 \\
S &\to V_1V_1 \mid V_3V_1 \mid V_2V_2 \mid V_4V_2 \\
V_1 &\to 0 \\
V_2 &\to 1 \\
V_3 &\to V_1S \\
V_4 &\to V_2S
\end{aligned}$$

5.5 Pumping Lemma per linguaggi CF

Anche questa volta, la definizione di Pumping Lemma è simile a quella dei linguaggi regolare e quindi fornisce una condizione necessaria perché un linguaggio sia CF. Ovvero che se L è CF allora vale Π condizione del pumping lemma.

Teorema 5.5.1 Pumping Lemma per linguaggi CF

Sia $L \subseteq \Sigma^*$ un linguaggio CF. Allora esiste $k \in \mathbb{N}$ t.c. per ogni stringa $z \in L$ con $|z| \geq k$ può essere scritta come:

$$z = uvwxy$$

con:

- $|vwx| \leq k$
- $|vx| > 0$
- $\forall i \in \mathbb{N} \,.\, uv^i wx^i y \in L$

Proof. Sia $L \subseteq T^*$ un linguaggio CF. Esiste una grammatica $G = (V, T, P, S)$ CF t.c. $L = L(G)$. Senza perdita di generalità, possiamo assumere che G sia in forma normale di Chomsky. Quindi le produzioni saranno del tipo $A \to a \in T$ o $A \to BC$ con $B, C \in V$. Sia $n = |V|$ il numero di sinmboli non terminali di G. Questo poiché data la forma normale implica che tutti gli altri alberi di derivazione in G sono binari. Prendiamo $k = 2^n$ dove appunto n è il numero di simboli non terminali. Si fa questo perché un albero binario di altezza h ha al massimo $2^h - 1$ nodi interni. Sia

$$z \in L \text{ con } |z| \geq k$$

Se le foglie sono $\geq 2^n$ allora l'altezza dell'albero di derivazione di z è almeno $n+1$. Il numero di archi (che non è nient'altro l'altezza dell'albero) che collegano simboli non terminali allora sono n se tolgo l'ultimo arco. Quindi il numero di nodi interni (quindi etichettati come non terminali) nel cammino sono $n+1$. Questo vuol dire che almeno una etichetta non termianle è ripetuta nel cammino due volte. Esistono almneo due vertici interni che hanno la stessa etichetta.
Supponiamo che un etichetta A sia il primo nodo (da S) con l'etichetta che si ripete. Adesso andiamo a prendere i sotto alberi radicati in A e dal secondo A che saranno entrambi alberi binari. Osserviamo che la stringa è stata divisa in cinque parti $z = uvwxy$. Preso il vincolo $|vx| > 0$ quindi $A \to AS \to a \not\in$ quindi almeno un simbolo deve essere generato in v o in x (poiché entrambi non possono essere zero).

Supponiamo ora che la ripetizione più vicina alla foglie. Questo vuol dire che dalla prima A alle foglie non ci sono ripetizioni ma questo vuol dire che attraversiamo massimo $n = |V|$ vertici perché se ne attraversassi uno in più allora almeno uno si ripeterebbe. Quindi non generiamo più di 2^n simboli terminali.

5.5.1 Uso del lemma

Quindi L CF vuol dire Π del lemma $\equiv \neq \Pi$ allora L non CF. Dove Π è:

- $\exists k \in \mathbb{N}$
- $\forall z \in L$ con $|z| \geq k$
- $\exists uvwxy = z$ con $|vwx| \leq k$ e $|vx| > 0$
- $\forall i \in \mathbb{N}\ .\ uv^iwx^iy \in L$

$\neg\Pi$ diventa:

- $\forall k \in \mathbb{N}$
- $\exists z \in L$ con $|z| \geq k$
- $\forall uvwxy = z$ con $|vwx| \leq k$ e $|vx| > 0$
- $\exists i \in \mathbb{N}\ .\ uv^iwx^iy \notin L$

Esempio 5.7

Consideriamo il seguente linguaggio:

$$L = \{a^nb^nc^n \mid n \in \mathbb{N}\}$$

Dimostriamoc che L non è context free usando la negazione del pumping lemma. Prendiamo $n = k$ siccome n non ha alcun vincolo e scegliamo la stringa z:

$$z = a^kb^kc^k \in L$$

- Prendiamo $z = a^kb^kc^k$ con $v, x \in a^k$.

  $$z_i = a^{k+(i-1)|vx|}b^kc^k$$

  Prendiamo $i = 2$:

  $$z_2 = a^{k+|vx|}b^kc^k \overset{?}{\in} L$$

Guardando la condizione di appartenenza a L:

$$a^i b^j c^l \in L \text{ sse } i = j = l$$

Quindi

$$z_2 \in L \text{ sse } \begin{cases} k + |vx| = k & \iff |vx| = 0 \\ k = k & ovvio \end{cases}$$

perché per costruzione $|vx| > 0$ allora $z_2 \notin L$.

- $v \in a^k$ e $x \in b^k$.

  $$z_i = a^{k+(i-1)|v|} b^{k+(i-1)|x|} c^k$$

  Prendiamo $i = 2$:

  $$z_2 = a^{k+|v|} b^{k+|x|} c^k \overset{?}{\in} L$$

  Appartiene solo se:

  $$k + |v| = k + |x| = k$$

  Quindi necessariamente $|v| = 0$ e $|x| = 0$ che è una contraddizione perché allora $|vx| = 0$ e quindi $z_2 \notin L$.

5.6 Proprietà di chiusura

Teorema 5.6.1

I linguaggi CF sono chiusi rispetto a:

- Unione
- Concatenazione
- Stella di Kleene

Proof. Consideriamo $G_1 = (V_1, T, P_1, S_1)$ e $G_2 = (V_2, T, P_2, S_2)$ due grammatiche CF tali che $L(G_1)$ e $L(G_2)$ sono due linguaggi CF riconosciuti dalle grammatiche. Allora $L(G_1) \cup L(G_2)$ è CF, ovvero esiste una grammatica che lo genera.

- Questo vuol dire che per l'unione, esiste una grammatica:

  $$G' = (V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \to S_1 | S_2\}, S)$$

- Per la concatenazione vale lo stesso ragionamento:

$$G' = (V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$$

- Per la stella di Kleene dove $L(G_1)^*$:

$$G' = (V_1 \cup \{S\}, T_1, P_1 \cup \{S \to S_1 S \mid \varepsilon\}, S)$$

Teorema 5.6.2

I linguaggi CF **non** sono chiusi per intersezione. Dati L_1, L_2 CF, $L_1 \cap L_2$ potrebbe non essere CF.

Proof. Prendiamo

$$L_1 = \{a^i b^i c^j \mid i, j \in \mathbb{N}\}$$

Questo linguaggio è intuitivamente CF perché non ci sta una dipendenza tra il numero di b e c.

$$S \to RC$$

$$R \to aRb \mid \varepsilon$$

$$C \to cC \mid \varepsilon$$

Prendiamo ora:

$$L_2 = \{a^j b^i c^i \mid i, j \in \mathbb{N}\}$$

Anche questo è CF perché non c'è dipendenza tra a e b.

$$S \to AC$$

$$A \to aA \mid \varepsilon$$

$$C \to bCc \mid \varepsilon$$

Si vuole dimostrare che $L_1 = L(G_1)$ e $L_2 = L(G_2)$. Consideriamo ora l'intersezione:

$$\begin{aligned} L = L_1 \cap L_2 &= \{a^i b^j c^k \mid i = j, j = k\} \\ &= \{a^n b^n c^n \mid n \in \mathbb{N}\} \text{ che non è CF} \end{aligned}$$

L'intersezione non è CF (dimostrato tramite pumping lemma) e quindi abbiamo dimostrato che i linguaggi CF non sono chiusi per intersezione □

Quando si parla di unione e intersezione di linguaggi si intende l'unione e l'intersezione **finita**.

Se L_i sono infiniti linguaggi regolari o CF, allora:

$$\bigcup_i L_i \text{ potrebbe non essere regolare o CF}$$

Teorema 5.6.3

I linguaggi CF **non** sono chiusi per complemento.

Proof. Se dato $L \in CF$ anche $\overline{L}$ fosse CF allora osserviamo che:

$$L_1, L_2 \text{ CF allora } L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}} \text{ (De Morgan)}$$

Se questo fosse vero vorrebbe dire che:

$$\overline{L_1}, \overline{L_2} \text{ sarebbero CF}$$

$$\overline{L_1} \cup \overline{L_2} \text{ sarebbero CF}$$

$$\overline{\overline{L_1} \cup \overline{L_2}} \text{ sarebbe CF}$$

ovvero

$$L_1 \cap L_2 \text{ sarebbe CF}$$

che abbiamo precedentemente dimostrato non essere sempre vero.

5.7 Problemi decidibili per linguaggi CF

Per i linguaggi CF i seguenti problemi sono decidibili: Sia L un linguaggio CF:

1. $L = \varnothing$
2. L è finito
3. L è infinito

È decidibile determninare se $x \in L$.

5.8 Automi a pila

Gli automi a pila ci servono per riconoscere i linguaggi CF. Questo perché gli automi a stati finiti non sono sufficienti per riconoscere i linguaggi CF come ad esempio $L = \{a^n b^n \mid n \in \mathbb{N}\}$ perché dovremmo ricordarci il numero di a letti per poter confrontare con il numero di b letti. Questo comporterebbe ad avere stati infiniti e di conseguenza non sarebbe più sufficiente un automa a stati finiti. Definiamo quindi un automa (a stati finiti) che possiede:

- Un nastro in sola lettura in unica direzione
- Memoria (LIFO)

L'acronimo per questo automa è APND (*Automa a pila non deterministico*). Al NFA aggiungiamo un alfabeto per la pila e un simbolo iniziale della pila.

Definizione 5.2

Un automa a pila non deterministico è una settupla:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

dove:

- Q è un insieme finito di stati
- Σ è un insieme finito di simboli di input (l'alfabeto)
- Γ è un insieme finito di simboli di pila (l'alfabeto della pila)
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ è la funzione di transizione che quindi dato uno stato in cui si trova l'automa, un eventualmente simbolo sul nastro e il simbolo in cima alla pila restituisce un insieme di coppie (stato successivo, stringa da mettere in cima alla pila)
- $q_0 \in Q$ è lo stato iniziale
- $Z_0 \in \Gamma$ è il simbolo iniziale della pila
- $F \subseteq Q$ è l'insieme degli stati finali (accettanti)

La chiusura transitiva di δ fornisce un'esecuzione dell'automa.

5.8.1 Linguaggio riconosciuto da un APND

Supponiamo che $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ sia un APND. Allora abbiamo due possibili linguaggi riconosciuti da M:

- $L_p(M)$: linguaggio riconosciuto per pila vuota
- $L_f(M)$: linguaggio riconosciuto per stato finale

Prendiamo per esempio:

$$L_p(M) = \{\sigma \in \Sigma^* \mid (q_0, \sigma, Z_0) \to^* (q, \varepsilon, \varepsilon), q \in Q\}$$

$$L_f(M) = \{\sigma \in \Sigma^* \mid (q_0, \sigma, Z_0) \to^* (q, \varepsilon, \gamma), \gamma \in R^*, q \in F\}$$

Teorema 5.8.1

Per ogni APND M esiste un APND M' tale che:

$$L_p(M) = L_f(M')$$

e viceversa. Questo vuol dire che la scelta di modalità di riconoscimento **non** influenza la potenza espressiva dell'automa a pila.

Esempio 5.8

Consideriamo di avere il seguente linguaggio:

$$L = \{\sigma C \sigma^R \mid \sigma \in \{a, b\}^*\}$$

dove σ^R è la stringa inversa di σ. Questo è anche l'insieme delle stringhe palindrome.

$$\sigma = abb \rightarrow \sigma^R = bba$$

La grammatica che genera questo linguaggio è:

$$S \rightarrow aSa \mid bSb \mid c$$

Proviamo a fare una produzione del tipo:

$$\begin{aligned} S \rightarrow aSa &\rightarrow abSba \\ &\rightarrow abbSbba \\ &\rightarrow \underbrace{abb}_{\sigma} c \underbrace{bba}_{\sigma^R} \end{aligned}$$

che si rappresenta con la segeunte tabella (una per ogni stato): Definendo questo linguaggio con la settupla abbiamo:

$$(\{q_0, q_1\}, \{a, b, c\}, \{Z, A, B\}, q_0, Z, \varnothing, S)$$

q_0	a	b	c
Z	q_0, zA	q_0, zB	q_1, ε
A			
B			

q_1	a	b	c
Z	q_1, z	q_1, z	
A	q_1, ε	q_1, z	
B	q_1, z	q_1, ε	

Esempio 5.9

Proviamo a fare un esempio analogo ma leggermente più difficile.

$$L = \{\sigma\sigma^R \mid \sigma^R \text{ reverse di } \sigma, \sigma \in a, b^*\}$$

Questa volta non abbiamo il divisore C che ci permette di sapere dove si trova la metà della stringa. Questo ci porta a costruire un automa non deterministico. Dove

$$Q = \{q_0, q_1\} \quad \Sigma = \{a, b\} \quad \Gamma = \{Z, A, B\}$$

q_0	a	b	c
Z		q_0, AZ	q_0, BZ
A		q_0, AA o q_1, ε	q_0, BA
B		q_0, AB	q_0, BB o q_1, ε

q_1	a	b	c
Z	q_1, ε		
A		q_1, ε	
B			q_1, ε

Gli automi a pila determinsitici sono sottoinsiemi degli automi a pila non deterministici.

$$\underbrace{APD}_{\text{Parser}} \subset \underbrace{APND}_{\text{CF}}$$

I linguaggi riconosciuti dagli APD sono i linguaggi di programmazione.

5.8.2 Corrispondenza tra APND e Grammatiche CF

Teorema 5.8.2

$L = L(M)$ con M APND (linguaggio riconosciuto da M a pila non deterministico) Allora esiste M' APND t.c. $L = L(M')$.

Proof. Dato un APND $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ con $|Q| \geq 1$ e con $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ L'idea è quella di prendere un nuovo simbolo per la pila ogni coppia $Q \times \Gamma$.

$$(q_i, z_i) \rightsquigarrow \text{ aggiungiamo } z_i'$$

$$\Gamma' = \Gamma \cup \{z_i'\}$$

Dobbiamo modificare δ:

$$\delta(q_i, a, z_i) = \delta'(q_0, a, z_i')$$

Teorema 5.8.3

Se $L = L(M)$ con M APND con uno stato allora L è CF. Questo vuol dire che $\exists$ G CF t.c. $L(G) = L(M)$.

Proof. Preso $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \varnothing)$ con $Q = \{q_0\}$ e accettiamo L per pila vuota. Definiamo una grammatica $G = (V, T, P, S) \equiv (R, \Sigma, P, Z_0)$. L e produzioni P sono:

$$\boxed{\text{z}} \to \boxed{\text{a}} z_1 z_2 \dots z_n \in P \text{ sse } (q_0, z_1 z_2 \dots z_n) \in \delta(q_0, \boxed{\text{a}}, \boxed{\text{z}})$$

Teorema 5.8.4

Se L è CF allora $\exists$ M APND t.c. $L = L(M)$.

Greibach: ogni linguaggio CF è generatod da una grammatica le cui produzioni sono della forma

$$A \to a\alpha \quad \alpha \in V^* \quad a \in T$$

Per trasformare in forma normale di Greibach: **Unfolding:** $G = (V, T, P, S)$ CF, $A \to \alpha\beta\gamma \in P$ e $B \to B_1 \mid B_2 \mid \dots \mid B_n \in P$ Alora $G' = (V, T, P', S)$ dove eliminaimo $A \to \alpha\beta\gamma$ e aggiungiamo

$$A \to \alpha\beta_1\gamma \mid \alpha\beta_2\gamma \dots \mid \alpha\beta_n\gamma$$

Quindi $L(G') = L(G)$.

Esempio 5.10

Se per esempio avessimo la produzione:

$$S \to aSb \mid ab$$

$$B \to b$$

$$S \to aSB \mid aB$$

Quello che abbiamo fatto è stato sostituire B nella produzione di S. Prendiamo un altro esempio:

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

Teorema 5.8.5 Eliminazione ricorsione sinistra

Sia $G = (V, T, P, S)$ una grammatica CF.

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_n \Leftarrow A \to Aa$$

$$A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \Leftarrow A \to b$$

Se costruiamo $G' = (V \cup \{B\}, T, P', S)$ dove per P' sostituiamo le produzioni di A con:

$$A \to \beta_i \mid \beta_i B$$

Allora $L(G) = L(G')$.

Esempio 5.11

Prendiamo la produzione:

$$A \to Ab \mid b$$

prendiamo:

$$B \to b \mid bB \quad A \to b \mid bB$$

Teorema 5.8.6

Se L è CF allora esiste un APND M tale che $L = (M)$.

Proof. L è CF allora esiste una grammatica $G = (V, T, P, S)$ in forma normale di Greibach tale che $L = L(G)$. Costruiamo M:

$$Q = \{q\} \quad \Sigma = T \quad R = V \quad Z_0 = S \quad F = \varnothing$$

$$(q, \overbrace{a}^{\in R^* = V^*}) \in \delta(q, \overbrace{a}^{T=\Sigma}, \overbrace{A}^{R=V}) \text{ sse } \overbrace{A}^{V} \to \overbrace{a}^{\Sigma}\overbrace{\alpha}^{V^*} \in P$$

Esempio 5.12

$$A \to a\alpha \mid a\beta \rightsquigarrow \begin{matrix} (q, a\alpha) \in \delta(q, a, A) \\ (q, a\beta) \in \delta(q, a, A) \end{matrix}$$

5.9 Gerarchia di Chomsky

La gerarchia di Chomsky classifica i linguaggi formali in quattro tipi principali:

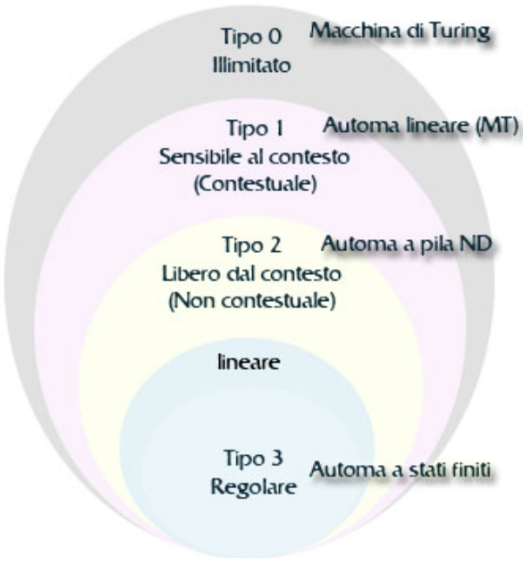


Figure 2: Gerarchia di Chomsky

Proviamo a costruire una grammatica per $a^n b^n c^n$ (Grammatica di tipo 1):

$$S \to aSBC \mid aBC$$
$$CB \to BC$$
$$bB \to bb \quad \text{si vuole sostituire una B solo quando è preceduta da una b}$$
$$\cancel{B \to b} \quad bC \to bc$$
$$cC \to cc$$
$$aB \to ab$$

Esempio 5.13

$$
\begin{aligned}
S \to ASBC &\to aaSBCBC \\
&\to aaSBBCC \\
&\to aaaBCBBCC \\
&\to aaabCBBCC \\
&\to aaabBCBCC \\
&\to a^3bBBCCC \\
&\to a^3bbBCCC \\
&\to a^3bbbCCC \\
&\to a^3bbbcCC \\
&\to a^3bbbccC \\
&\to a^3bbbccc \\
&\to a^3b^3c^3
\end{aligned}
$$

Questa grammatica genera il linguaggio:

$$L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$$

che non è CF.

6 Calcolabilità e funzioni primitive ricorsive

6.1 Potenza espressiva

Esistono diversi modelli di calcolo, ognuno con una diversa potenza espressiva:

- ASD (= Automa a stati finiti deterministico) riconosce i linguaggi regolari
- GCF (= APND)
- APD
- ...
- Linguaggi di programmazione

E poi esistono le specifiche istanze del modello per riconoscere un linguaggio specifico come:

- M con $q_0 \in a, b.$
- $G = (V, T, P, S)$
- $M_{pila} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

- ...
- Grammatica per linguaggio di programmazione

E questo ha una corrispondenza con le funzioni calcolabili. Fino ad ora era superfluo parlare di funzioni poiché non andavamo nel dettaglio dell'implementazione di un linguaggio. Ora però diventa importante per poter ragionare su cosa sta fuori dai modelli visti.

$$L_f = \{a^{f(n)} \mid n \in \mathbb{N}\}$$

in questo modo "trasformiamo" il calcolo di f nel riconoscimento di L_f. Quali funzioni corrispondono a L_f regolari e quali a L_f CF?

$$f(n) = 2n \Rightarrow L_f = \{a^{2n} \mid n \in \mathbb{N}\} \quad \text{Reg}$$

$$f(n) = n^2 \Rightarrow L_f = \{a^{n^2} \mid n \in \mathbb{N}\} \quad \text{non CF}$$

Ora consideriamo un contesto più generale parlando di insiemi (di $\mathbb{N}$) invece che di linguaggi.

$$A_f = \{f(n) \mid n \in \mathbb{N}\}$$

Così ci focalizziamo sul calcolo della funzione. Dividiamo ora in due la funzione calcolata e l'algoritmo come processo di calcolo. Quindi **la funzione** non è altro che l'associazione $f : \mathbb{N} \to \mathbb{N}$ dove $f \subseteq \mathbb{N} \times \mathbb{N}$. Sappiamo che:

$$|f| = |\mathcal{P}(\mathbb{N} \times \mathbb{N})| = 2^{|\mathbb{N}|}$$

Possiamo costruire delle funzioni in maniera ricorsiva come se fosse un **algoritmo** con una sequenza di istruzioni:

$$\begin{cases} f(0) = 1, f(1) = 1 \\ f(x+2) = f(x+1) + f(x) \end{cases}$$

Per calcolabilità, ci domandiamo quali funzioni $f : \mathbb{N} \to \mathbb{N}$ sono calcolabili (mediante un algoritmo possiamo costruire l'output a partire dall'input).

6.2 Funzioni primitive ricorsive

Questo è un modello di calcolo che permette di descrivere un processo di calcolo componendo funzioni più semplici o già definite.

Definizione 6.1: Funzioni ricorsive di base

Le funzioni ricorsive di base sono:

- Funzione zero: $0 : \lambda x.0$
- Funzione successore: $S : \lambda x.x + 1$
- Funzioni proiezione: $P_i^n : \lambda(x_1, x_2, \ldots, x_n).x_i \quad 1 \leq i \leq n$
  O anche la funzione identità dove appunto abbiamo un solo parametro di input: $Id : \lambda x.x$

Queste sono tutte funzioni totali ovvero che sono definite su tutte i numeri naturali (che sono gli input).

6.2.1 Operazioni

- **Composizione:** dati $g : \mathbb{N}^k \to \mathbb{N}$ e $h_1, h_2, \ldots, h_k : \mathbb{N}^n \to \mathbb{N}$ definiamo la funzione $f : \mathbb{N}^n \to \mathbb{N}$ come:

  $$f(x_1, x_2, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), h_2(x_1, \ldots, x_n), \ldots, h_k(x_1, \ldots, x_n))$$

  e in lambda calcolo:

  $$f = \lambda(x_1, x_2, \ldots, x_n).g(h_1(x_1, \ldots, x_n), h_2(x_1, \ldots, x_n), \ldots, h_k(x_1, \ldots, x_n))$$

  La composizione di funzioni totali è totale.

- **Primitiva ricorsione:** supponiamo di avere $g : \mathbb{N}^n \to \mathbb{N}$ e $h : \mathbb{N}^{n+2} \to \mathbb{N}$. Definiamo la funzione $f : \mathbb{N}^{n+1} \to \mathbb{N}$ come:

  $$\begin{cases} f(0, x_1, x_2, \ldots, x_n) = g(x_1, x_2, \ldots, x_n) \\ f(y+1, x_1, x_2, \ldots, x_n) = h(y, f(y, x_1, x_2, \ldots, x_n), x_1, x_2, \ldots, x_n) \end{cases}$$

  La primitiva ricorsione di funzioni totali è totale.

Quando una funzione $f(x)$ è totale la scriveremo come

$$f(x)\downarrow$$

Mentre quando non è definita e quindi non termina si usa la seguente notazione:

$$f(x)\uparrow$$

Esempio 6.1 (Funzione somma : $+ : \mathbb{N}^2 \to \mathbb{N}$)

Definiamo la funzione somma come:

$$\begin{cases} +(0, y) = y \\ +(x + 1, y) = S(+(x, y)) \end{cases}$$

Andiao a definire le funzioni di base:

$$f_1 = \lambda x.x$$

$$f_2 = \lambda x.S(x) = \lambda x.x + 1$$

$$f_3 = \lambda x_1 x_2 x_3.x_2$$

$$f_4 = f_2 \circ f_3$$

$$+ = f_5 : \begin{cases} f_5(0, x) = f_1(x) = x \\ f_5(y + 1, x) = f_4(y, f_5(y, x), x) = f_2 \circ f_3(y, f_5(y, x), x) = S(f_5(y, x)) \end{cases}$$

$$+ : f_1, f_2, f_3, f_4, f_5$$

Esempio 6.2 (Funzione moltiplicazione : $* : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$)

Definiamo la funzione moltiplicazione come:

$$\begin{cases} *(0, y) = 0 \\ *(x + 1, y) = +(*(x, y), y) \end{cases}$$

Andiamo a definire le funzioni di base:

$$f_1 = \lambda x.0$$
$$f_2 = \lambda x_1 x_2 x_3.x_2$$
$$f_3 = \lambda x_1 x_2 x_3.x_3$$
$$f_4 = + = \lambda x_1 x_2.+(x_1, x_2)$$
$$f_5 = f_4 \circ (f_2, f_3)$$
$$f_6 = \begin{cases} f_6(0, x) = f_1(x) = 0 \\ f_6(y + 1, x) = f_5(y, f_6(y, x), x) = +(f_6(y, x), x) \end{cases}$$
$$* : f_1, f_2, f_3, f_4, f_5, f_6$$

Quindi se riesco a definire una funzione tramite composizione e primitiva ricorsione a

partire dalle funzioni di base allora la funzione è sicuramente totale.

Esempio 6.3 (Decremento : $dec : \mathbb{N} \to \mathbb{N}$)

Definiamo la funzione decremento come:

$$\begin{cases} dec(0) = 0 \\ dec(x+1) = x \end{cases}$$

Andiamo a definire le funzioni di base:

$$f_1 = \lambda x.0$$
$$f_2 = \lambda x_1 x_2 x_3 . x_1$$
$$f_3 = \begin{cases} f_3(0, x) = f_1(0) = 0 \\ f_3(y+1, x) = f_2(y, f_3(y, x), x) = y \end{cases}$$
$$dec : f_1, f_2, f_3$$

Esempio 6.4 (Differenza)

Definiamo la funzione differenza come:

$$\begin{cases} -(0, y) = y \\ -(x+1, y) = dec(-(x, y)) \end{cases}$$

Definizione 6.2: Primitiva Ricorsiva

$f \in$ PR (dove PR è la classe delle funzioni primitive ricorsive ovvero la minima classe di funzioni chiuse per composizione e primitiva ricorsione che contiene le funzioni ricorsive di base) se esiste un sequenza P_f di funzioni in PR tali che l'ultima funzione della sequenza è esattamente f.

$$P_f = f_1, f_2, \ldots, f_n \mid \forall i f_i \in PR \text{ e } f_n = f$$

Proof. Mostriamo che questo modello fallisce nel catturare il concetto di calcolabile.

$$f : \mathbb{N} \to \mathbb{N}, f \in PR \ \exists P_f \text{ sequenza di } f_i \in PR \mid f_n = f$$

Defininiamo $h(n) = f_n(n) + 1$:

$$h : \mathbb{N} \to \mathbb{N}$$

Quello che fa è prendere n, generiamo la lista di funzioni fino ad n e gli sommo 1. Di conseguenza $h(n)$ è calcolabile e le parziali ricorsive catturano tutto ciò che è calcolabile. Allora $h \in PR$. Questo vuol dire che $\exists i$ indice t.c. $f_i = h$. Ma allora possiamo calcolare $h(i)$:

$$f_i(i) = h(i) = f_i(i) + 1$$

ma questo è assurdo perchè è impossibile che un numero naturale sia uguale al suo successore. Questo dimostra che PR non può catturare tutte le funzioni calcolabili. Quindi funzioni non è vero che tutte le funzioni totali sono anche calcolabili.

Esempio 6.5 (Funzione di Ackermann)

La funzione di Ackermann è una funzione totale non primitiva ricorsiva.

$$Ack(x,y) = \begin{cases} Ack(0,y) = y+1 \\ Ack(x+1,0) = Ack(x,1) \\ Ack(x+1,y+1) = Ack(x, Ack(x+1,y)) \end{cases}$$

Questa funzione cresce molto velocemente. Infatti se abbiamo $x = 4$ già per $y = 2$ otteniamo un numero con più di 19000 cifre. Notiamo che la nostra h in questo caso non è una funzione primitiva già precedentemente calcolata ma la funzione stessa. Questo non rispetta la definizione di primitiva ricorsione ed è per questo che la funzione di Ackermann non è primitiva ricorsiva.

6.3 Macchine di Turing

Una macchina di Turing è un modello di calcolo basato su DFA più potente degli automi a stati finiti e degli automi a pila. Queste permettono di descrivere come avviene il processo di calcolo come sequenza di operazioni di scrittura e lettura su una memoria. Sarà costituito da un **programma finito** ovvero la funzione di transizione (una tabella finita). Può essere immaginata come una testina di lettura/scrittura che si muove su un nastro infinito (la memoria) leggendo e scrivendo simboli.

Definizione 6.3: Macchina di Turing

Una macchina di Turing è una settupla:

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

dove:

- Q è un insieme finito di stati
- Σ è un insieme finito di simboli di input (l'alfabeto), e conterrà almeno due simboli che rappresentano la cella sul nastro vuoto e almeno un simbolo significativo.
- Γ è un insieme finito di simboli del nastro (l'alfabeto del nastro) e $\Sigma \subseteq \Gamma$
- P è un insieme finito di istruzioni (corrisponde al δ nel DFA) $P = \{I_1, \ldots, I_k\}$ dove ogni i è una quintupla che possono essere di due tipi:
  - q: Stato centrale
  - s: Simbolo letto sul nastro
  - q': Stato successivo
  - s': Simbolo che sostituisce s
  - R: spostamento della testina (Right, Left, Stay)

  Quindi un'istruzione sarà del tipo:

  $$(q, s, q', s', R) \in P$$

  $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ è la funzione di transizione.
- $q_0 \in Q$ è lo stato iniziale
- $B \in \Gamma - \Sigma$ è il simbolo vuoto del nastro
- $F \subseteq Q$ è l'insieme degli stati finali (accettanti)

6.3.1 Descrizione istantanea di una MdT

Una descrizione istantanea (ID) di una MdT rappresenta lo stato corrente della macchina in un dato istante.

$$\ldots \$\$ \underbrace{s_1 s_2 \ldots s_{i-1}}_{w} \underbrace{s_i}_{q} \underbrace{s_{i+1} \ldots s_n}_{v} \$\$$$

La composizione di una MdT è una sequenza di descrizioni istantanee.

$$\delta(q, r) = (q', s', R)$$

Una computazione di una MdT è una sequenza di descrizioni istantanee.

Definizione 6.4: Computazione terminante

Una computazione è terminante se esiste una descrizione istantanea q, v, r, w tale che nessuna istruzione inizia con con q, r ovvero $\delta(q, r)$ non è definita.

Esiste un equivalenza tra MdT e funzioni ricorsivi primitive. Possiamo infatti tradurre le funzioni come $\lambda x.0$ o $\lambda x.x + 1$ in una tabella finita:

Stato	Simbolo letto	Simbolo scritto	Spostamento
q_0	\$	0	R
q_1	\$	\$	S

Table 1: MdT per la funzione zero

Teorema 6.3.1 Funzione Turing calcolabile (DA COMPLETARE)

Una funzione $f : \mathbb{N} \to \mathbb{N}$ è **Turing calcolabile** se esiste una MdT M tale che partendo dalla configurazione iniziale:

Le macchine di Turing riconoscono anche macchine che divergono. Infatti una MdT che non termina:

	\$
q_0	$q_0\$R$

Table 2: $f(x) = \lambda x. \uparrow$

Definizione 6.5: Funzioni parziali ricorsive

Le funzioni parziali ricorsive possono essere viste come parziali ricorsive + minimizzazione. Consideriamo una funzione totale $f : \mathbb{N}^{n+1} \to \mathbb{N}$ (n+1-aria) definiamo:

$$\varphi(x_1, \ldots, x_n) = \underbrace{\mu z.(f(x_1, \ldots, x_n) = 0)}_{\text{Cerca il più piccolo valore z tale che } f(x_1,\ldots,x_n)=0 \text{ Se z non esiste } \varphi(x_1,\ldots,x_n)\uparrow}$$

Questa operazione si chiama **minimizzazione**.

Scriviamo l'algoritmo per calcolare φ:

```
input(x_1, ..., x_n)
z := 0
while f(x_1, ..., x_n, z) != 0 do
```

```
    z := z + 1
output(z)
```

Esempio 6.6

Prendiamo la funzione $\log_a x$ intero inferiore del $\log_a x$.

$$f(x) = \mu z. \quad \underbrace{le(x, a^{z+1})}_{le(x,y)=0 \iff x \leq y} = 0$$

Esempio 6.7

Prendiamo la funzione $\sqrt[n]{x}$ dove $x = a^n$ e $\sqrt[n]{x} = a$.

$$f(x) = \mu z.(le(x, (z+1)^n) = 0)$$

Teorema 6.3.2

Se $f : \mathbb{N} \to \mathbb{N}$ è **parziale ricorsiva** se è solo se f è **Turing calcolabile**.

Proof. $\Rightarrow$
Casi base sono stati già visti!

- **Composizione:** Dato f e g parziali ricorsive.

  $$f \circ g \text{ è parziale ricorsiva} \implies \exists M_{f \circ g}$$

  Allora esistono due MdT M_f e M_g che calcolano f e g per ipotesi induttiva. Costruiamo $M_{f \circ g}$ che simula M_g e poi M_f.

  $$M_{f \circ g} = \begin{cases} input(x) \\ temp := M_g(x) \\ output(temp) \end{cases}$$

- **Primitiva ricorsione:** Sia g e r primitive ricorsive allora esiste M_g ed M_h per ipotesi induttiva.

  $$f(x, n) = \begin{cases} g(x) & n = 0 \\ h(x, n-1, f(x, n-1)) & \text{altrimenti} \end{cases}$$

Costruiamo allora M_g:

$$M_g = \begin{cases} input(x) \\ input(n) \\ temp = M_g(x) \\ \text{for } i = 1 \text{ to } n \text{ do } \{\text{temp} = M_h(x, i, temp)\} \end{cases}$$

- **Minimizzazione:** Sia f parziale ricorsiva allora esiste M_f MdT per ipotesi induttiva.

$$\varphi(x_1, \ldots, x_n) = \mu z.(f(x_1, \ldots, x_n) = 0)$$

Costruiamo M_φ:

$$M_\varphi = \begin{cases} input(x_1, \ldots, x_n) \\ i := 0 \\ temp = M_f(x_1, \ldots, x_k, i) \\ \text{while } (temp \neq 0) \text{ do } \{ \\ \quad i := i + 1 \\ \quad temp = M_f(x_1, \ldots, x_k, i) \\ \} \\ output(z) \end{cases}$$

Proof. $\Leftarrow$
f Turing calcolabile $\Rightarrow$ parziale ricorsiva: Siano:

$$(q, n, s, m) \to (q', n', s', m') \text{ 1 passo da computare}$$

I simboli della MdT sono $\{\$, 0, 1\}$ e n, m sequenza di simboli sul nastro sono numeri binari.

$$L \to 0 \quad R \to 1$$

Scorporare le funzioni in $\delta : Q \times E \to Q \times \Sigma \times \{R, L\}$

$$Q : Q \times \Sigma \to Q \text{ transizione di stato}$$

$$\mathbb{S} : Q \times \Sigma \to \Sigma \text{ transizione di simbolo}$$

$$\mathbb{X} : Q \times \Sigma \to \{R, L\} \text{ transizione di direzione}$$

Distinguiamo 2 casi:

- Se $\mathbb{X}(q,s) = L$:

$$q' = \mathbb{Q}(q,s) \quad s' = n \bmod 2 \quad m' = 2m + \mathbb{S}(q,s) \quad n' = n \operatorname{div} 2$$

  Essendo mod e div PR, sono tutte PR.

- Se $\mathbb{X}(q,s) = R$:

$$q' = \mathbb{Q}(q,s) \quad s' = m \bmod 2 \quad n' = 2n + \mathbb{S}(q,s) \quad m' = m \operatorname{div} 2$$

  Combinaimo questi due casi:

$$q' = \mathbb{Q}(q,s) \quad s' = (n \bmod 2)(1 - \mathbb{X}(q,s)) + (m \bmod 2)\mathbb{X}(q,s)$$

$$m' = (2m + \mathbb{S}(q,s))(1 - \mathbb{X}(q,s)) + (m \operatorname{div} 2)\mathbb{X}(q,s)$$

$$n' = (n \operatorname{div} 2)(1 - \mathbb{X}(q,s)) + (2n + \mathbb{S}(q,s))\mathbb{X}(q,s)$$

$$f : (q,n,s,m) \to (q',n',s',m') \text{ è PR}$$

Per primitiva ricorsione calcoliamo lo stato raggiunto dopo t passi:

$$P_q(t,q,n,s,m)$$

dove senza fare alcun passo:

$$\begin{cases} P_q(0,q,n,s,m) = q \\ P_q(t+1,q,n,s,m) = P_q(t,q',n',s',m') \end{cases}$$

dalla descrizione istantanea (q,n,s,m) dopo t passi e otteniamo lo stato raggiunto. Definiamo per primitiva ricorsione il risultato (di computazione parziale) dopo t passi:

$$\begin{cases} P_o(0,q,n,s,m) = n \\ P_o(t+1,q,n,s,m) = P_o(t,q',n',s',m') \end{cases}$$

e restituiscce la sequenza a sinistra della testina da t passi di calcolo. In Q supponiamo di avere uno stato di terminazione q_f: Quindi possiamo definire il calcolo di M con questa funzione:

$$\varphi_M(x) = P_0(\mu t.(q_f - P_q(t,q_0,x,x \bmod 2,0) = 0), q_0, x \operatorname{div} 2, x \bmod 2, 0)$$

$P_q(t, q_0, x, x \bmod 2, 0) = 0$ cerca il più piccolo t tale che partendo da q_0 con input x a sinistra della testina in t passi arriviamo allo stato finale q_f. Calcolo l'output dopo t passi i quali passano allo stato finale. Quindi:

$$\text{MdT} = \text{ Parziali Ricorsive}$$

φ è una funzione parziale ricorsiva se $\exists M$ MdT che calcola φ se e solo se $\exists$ un algoritmo che calcola φ. Se abbiamo M MdT allora la funzione calcolata è parziale ricorsiva. $\square$

6.3.2 λ-calcolo

Il λ-calcolo è un linguaggio di programmazione funzionale e anche questo si è dimostrato essere uguale alle macchine di Turing. La **Tesi di Church** (ovvero che non è un teorema perché non è stata dimostrata ma è stata accettata) afferma che:

La classe delle funzioni "intuitivamente calcolabili" è equivalente alla classe delle funzioni di Turing calcolabili.

6.4 Aritmetizzazione delle macchine di Turing

Codifichiamo (univocamente) ogni MdT in $\mathbb{N}$: Se consideriamo:

$$\sigma = \{0, \$\} \quad Q = \{q_0\}$$

Tutte le macchine di Turing possibili sono:

	0	\$
q_0	•	•

dove • sono tutte le possibili istruzioni:

$$\begin{cases} (\$, \$, \$) \\ (q_0, \$, R) \\ (q_0, \$, L) \\ (q_0, 0, R) \\ (q_0, 0, L) \end{cases}$$

Il numero di macchine di Turing possibili da 1 stato e 2 simboli è:

$$5 \times 5 = 25$$

Possiamo mettere in ordine gli insiemi degli stati e dei simboli perché sono finiti:

$$Q = \{q_0, q_1, q_2, \ldots, q_{m-1}\} \quad \Sigma = \{s_0, s_1, s_2, \ldots, s_{n-1}\}$$

Anche le triple possono essere messe in ordine lessicografico:

$$(q_i, s_j) < (q_p, s_q) \text{ se } i < p \text{ o } (i = p \text{ e } j < q)$$

6.4.1 Codifica di Gödel

La fattorizzazione di Gödel ci permette di codificare ogni tripla come un numero naturale: L'idea è di prendere un numero $x \in \mathbb{N}$ dove

$$fact(x) = p_1^{x_1} p_2^{x_2} p_3^{x_3} \ldots p_k^{x_k} \quad \text{con } p_i \text{ primi}$$

Associamo ad ogni simbolo y della MdT un numero dispari $> 1 \rightsquigarrow a(y)$, poi calcoliamo il $gn(e)$ dove $e = (q, s, q', s', \{R, L\})$ è una istruzione della MdT.

$$gn(e) = \prod_{i=1}^{5} p_i^{a(s_i)}$$

Esempio 6.8

Immaginiamo un istruzione del tipo:

$$E = (q_0, \$, q_0, 0, R)$$

Allora

$$gn(E) = \underbrace{2^{a(q_0)} \times 3^{a(\$)} \times 5^{a(q_0)} \times 7^{a(0)} \times 11^{a(R)}}_{m}$$

dove m è il numero naturale che codifica l'istruzione E. Nello stesso modo calcoliamo $gn(n)$ dove n è l'insieme di tutte le istruzioni della MdT:

$$n = E_1 E_2 \ldots E_k (\text{istruzioni})$$

$$gn(n) = \prod_{i=1}^{k} p_i^{gn(E_i)}$$

Per decodificare invece una MdT a partire da un numero naturale m, fattorizziamo $n_1 \dots n_k$ dove n_i è una codifica di tuple.

$$\underbrace{\overbrace{m_1^1 \dots m_5^5}^{n_1} \quad \overbrace{m_1^2 \dots m_5^2}^{n_2} \quad \dots \quad \overbrace{m_1^k \dots m_5^k}^{n_k}}_{\text{tabella di } m_x}$$

Dove $n_1 \dots n_k$ poi costituiscono l'intera tabella delle istruzioni della MdT.

D'ora in poi scriveremo che m_x è la MdT il cui programma è codificato in $x \in \mathbb{N}$. m_x calcola la funzione φ_x dove

Definizione 6.6

φ_x è la notazione per elencare la funzione parziale ricorsiva calcolabile dal programma codificato in $x \in \mathbb{N}$.

$$\varphi_x(y) = m_x(y) \text{ calcolo la MdT } m_x \text{ su input } y$$

φ_x è la **semantica** (funzione calcolata) dal programma codificato da x.

Bisogna prestare attenzione perché x_1 programma che calcola $y+y$ e x_2 che calcola $2y$ calcolano con istruzioni diverse ma hanno la stessa semantica e quindi $\varphi_{x_1} = \varphi_{x_2}$.

6.4.2 Macchina universale

La macchina universale (o anche interprete) non è nient'altro che il modello di una macchina programmabile. È un programma che prende input due valori, interpreta il prima come programma e il secondo come codifica.

$$\begin{cases} x_1 \text{ codifica di una MdT } m_{x_1} \\ x_2 \text{ input per la MdT } m_{x_1} \end{cases}$$

$$\varphi_m(\underbrace{x_1}_{\text{programma}}, x_2) = \underbrace{\varphi_{x_1}}_{\text{costruiamo } m_{x_1} \text{ che calcola } \varphi_{x_1}}$$

che sottoforma di pseudocodice diventa:

$$\begin{cases} input(x_1) \\ input(x_2) \\ \text{decodifica } m_{x_1} \text{ da } x_1 \\ \text{se } m_{x_1} \text{ e' valida allora calcola } m_{x_1}(x_2) \\ \text{altrimenti entra in un loop infinito} \end{cases}$$

7 Problemi insolubili

7.1 Problema della terminazione

Non esiste una funzione totale ricorsiva g tale che:

$$\forall x.g(x) = \begin{cases} 1 & \text{se } \varphi_x(x) \downarrow \\ 0 & \text{se } \varphi_x(x) \uparrow \end{cases}$$

dove g decide sempre sì (1) o no (0) se la MdT codificata in x termina o meno.

Proof. Dimostriamo per assurdo. Supponiamo che g esista ed è totale ricorsiva $\implies$ vuol dire che esiste una MdT di codice $i_0 \in \mathbb{N}$ che la calcola. Questo vuol dire che $g = \underbrace{\varphi_{i_0}}_{\text{semantica}}$.

Possiamo definire

$$h(x) = \begin{cases} 0 & \text{se } g(x) = 0 = \varphi_{i_0}(x) \\ \uparrow & \text{se } g(x) = 1 = \varphi_{i_0}(x) \end{cases}$$

Essendo g è calcolabile allora anche h è calcolabile.

$$\exists i_1.h = \varphi_{i_1}$$

$$\begin{cases} input(x) \\ \text{if } m_{i_0}(x) = 1 \text{ then output}(0) \\ \text{else loop infinito} \end{cases}$$

L'ipotesi è che $m_{i_0}(x)$ esiste e termina sempre. Se $g = \varphi_{i_0}$ e $h = \varphi_{i_1}$ allora calcoliamo $h(i_1)$:

$$\varphi_{i_1}(i_1) \uparrow \iff h_{i_1} \uparrow \iff g(i_1) = 1$$

Si dovrebbero aspettare infiniti passiper decidere se $\varphi_x(x)$ termina o diverge quindi g non è calcolabile. Quindi non esiste ψ tale che decida sempre la terminazione di un altro programma su un suo input.

$$\psi(xy) = \begin{cases} 0 & \text{se } \varphi_x(y) \uparrow \\ 1 & \text{se } \varphi_x(y) \downarrow \end{cases}$$

Se esistesse, allora potrei decidere

$$\psi(xx) = g(x) \text{ dimostrato sopra che non è possibile}$$

7.1.1 Altri problemi insolubili

Non esiste f totale ricorsiva tale che:

$$\forall x.f(x) = \begin{cases} 1 & \text{se } \varphi_x \text{ è totale} \\ 0 & \text{se } \varphi_x \text{ non è totale} \end{cases}$$

Totale vuol dire che

$$\forall y \in \mathbb{N} \,.\, \varphi_x(y) \downarrow \text{ mentre non totale vuol dire } \exists y \,.\, \varphi_x(y) \uparrow$$

Altri problemi insolubili sono:

- Decidere se ϕ_x è costante: $\exists n.\forall y.\phi_x(y) = n$
- Decidere se ϕ_x non restituisce mai un valore fissato: $\forall y.\phi_x(y) \neq n$
- Decidere se $\phi_x = \phi_y$.

7.2 Teorema SMN (specializzazione)

Consideriamo di avere una funzione che ha due input:

$$\lambda x \lambda y.x + y \quad \text{programma } f(x, y)$$

specializzare equivale a costruire un nuovo programma in cui una parte dell'input è integrata nel codice. Usiamo f per ottenere f' che somma ad un valore di input il valore fisso 5.

$$f' = \lambda y.5 + y$$

Instanziamo x a 5 dentro f, ovvero **specializziamo** f a 5 per x. Lo specializzatore ritorna un nuovo programma.

Teorema 7.2.1

$\forall n, m \geq 1$ esiste una funzione totale ricorsiva S tale che $\forall x, y_1, y_2, \ldots, y_m \in \mathbb{N}$:

$$\lambda z_1 \ldots z_n \,.\, \underbrace{\varphi_x}_{\text{codice}} (\overbrace{y_1, y_2, \ldots, y_m}^{\text{input che specializziamo}} , \underbrace{z_1, z_2, \ldots, z_n}_{\text{input che rimane incognito}}) = \varphi_{S(x,y_1,y_2,\ldots,y_m)}(z_1, z_2, \ldots, z_n)$$

Quindi abbiamo una trasformazione totale S del codice x e considerando gli input specializzati $y_1, \ldots, y_m$.

Proof. $m = n = 1$ (caso particolare ma $m, n > 1$ è analogo)

$$\lambda z.\varphi_x(y, z)$$

Fissiamo x_0 (programma) e y_0 (input da specializzare).

$$\lambda z.\varphi_{x_0}(y_0, z)$$

dove φ_{x_0} denota la funzione calcolata della MdT il cui codice è aritmetizzato in x. Questa funzione è intuitivamente calcolabile. Di conseguenza esiste una MdT che la calcola per la tesi di Church il cui codice dipende ed è costruibile ricorsivamente a partire da x_0 e y_0. Quindi esiste S totale ricorisivan tale che il codice è $S(x_0, y_0)$.

$$\varphi_{S(x_0,y_0)}(z) = \lambda z.\varphi_{x_0}(y_0, z)$$

7.2.1 Prima proiezione di Futamura

La proiezione di Futamura è una tecnica per ottenere un compilatore a partire da un interprete e uno specializzatore. Dato un programma sorgente $source \in L$ esiste una trasformazione di $source$ in un programma (semanticamente) equivalente $target \in L'$.

Proof. Supponiamo Int sia la macchina universale.

$$Int(p, d) = P(d)$$

Supponaniamo $spec$ sia uno specializzatore, dove Q è un programma a due input

$spec(Q, d) = Q_d$ integra nel codice l'input

$$target = \underbrace{spec(Int, source)}_{\text{compilatore}}$$

7.2.2 Linguaggi Turing completi

Un linguaggio di programmazione è Turing completo se permette di calcolare tutte le funzioni intuitivamente calcolabili.

$$\exists p.\varphi_p = \lambda x.0$$
$$\exists p.\varphi_p = \lambda x.x + 1$$
$$\exists p.\varphi_p = \lambda x.x$$
$$\exists p.\varphi_p = \varphi_h \circ \varphi_q \text{ composizione di due programmi}$$
$$\exists p.\varphi_p = \varphi_{\text{while B}\{Q\}} \text{ minimizzazione}$$
$$\exists p.\varphi_p(x,y) = \phi_x(y)$$
$$\exists spec.\varphi_{spec(xy)(z)} = \lambda z.\varphi_x(y,z) \text{ specializzazione}$$

Se consideriamo la primitiva ricorsione abbiamo solo cicli for. Un linguaggio che non permette cicli non determinati **non** è Turing completo.

$$\exists\ MdT\ m\ .\ L = L(M) = \{y \mid \varphi_m(y)\downarrow\}$$

Quindi in linguaggi di Turing sono linguaggi accettati da una macchina di Turing. y sono le stringhe su cui la MdT termina. Se noi abbiamo φ_x parziale ricorsiva ovvero calcolata da una MdT dal codice x allora definiamo

$$Dom(\varphi_x) = \{y \mid \varphi_x(y)\downarrow\}$$

8 Teoria della ricorsione

8.1 Insiemi ricorsivamente enumerabili

La teoria della ricorsione parla degli insiemi (linguaggi) caratterizzati dalle MdT. **Ricorsivamente enumerabile** è un insieme di elementi (numeri) costruibili in modo algoritmico.

Definizione 8.1: Insieme ricorsivamente enumerabile (RE)

Un insieme $I \subseteq \mathbb{N}$ è recorsivamente enumerabile se è il dominio di una funzione parziale ricorsiva,

$$\exists \varphi_x \text{ parziale ricorsiva } .\ dom(\varphi_x) = I$$

Da qui in poi useremo la notazione dove $W_x \stackrel{def}{=} dom(\varphi_x)$. Se abbiamo φ_x parziale ricorsiva allora W_x è ricorsivamente enumerabile. Se abbiamo $I \subseteq \mathbb{N}$ e riusciamo a costruire la sua

funzione semicaratteristica

$$\psi_I(x) = \begin{cases} 1 & x \in I \\ \uparrow & \text{altrimenti} \end{cases} \implies dom(\psi_I) = I$$

Per dimostrare che un insieme è ricorsivamente enumerabile dobbiamo fornire l'algoritmo della semicaratteristica che deve terminare se l'input appartiene all'insieme.

Esempio 8.1 (Problema della terminazione (variante))

$$\{x \mid \exists y \,.\, \varphi_x(y) \downarrow\}$$

Questo programma cerca di determinare i programmi che terminan su almeno un input. Si può riscrivere con il seguente insieme:

$$\{x \mid \varphi_x(y) \downarrow\}$$

$$\begin{cases} input(x) \\ stop = 0; \\ build\ \varphi_x \text{ esiste algoritmo ricorsivo} \\ \text{while } \neg\ stop \text{ do} \\ \quad \text{Execute next step } \varphi_x(0) \\ \quad \text{if } \varphi_x(0) \downarrow \text{ then stop} = 1 \\ \text{return 1} \end{cases}$$

La funzione è semicaratteristica perché se $\varphi_x(0)$ restituisce 1 altrimenti diverge.

$$I = \{x \mid \exists y \,.\, \varphi_x(y) \downarrow\}$$

$$\begin{cases} input(x) \\ y = 0 \\ stop = 0 \\ \text{while } \neg\ stop \text{ do} \\ \quad \text{Execute next step } \varphi_x(y) \\ \quad \text{if } \varphi_x(0) \downarrow \text{ then stop} = 1 \\ \text{return 1} \end{cases}$$

Se diverge su 0 non potrò mai testare $y > 0$ ma $\varphi_x(1)$ potrebbe terminare. Questo algoritmo diverge per $x \in I$ quindi questo algoritmo NON è la semicaratteristica.

- Passo 0: Esegue un passo di $\varphi_x(0)$ se $\downarrow$ allora fine, se non termino eseguiamo il passo successivo
- Passo $n+1$: Esegue un passo di $\varphi_x(0), \varphi_x(1), \ldots, \varphi_x(n)$ (se mai sono terminate prima) esegui il primo passo per $\varphi_x(n+1)$

Questo algoritmo si chiama **Dovetail**. La semicaratteristica per I quindi:

$input(x)$
costruiamo φ_x (esiste una procedura ricorsiva per costruire φ_x da x)
$y = 0$ (serve per identicare gli input su cui abbiamo iniziato la computazione)
$stop = 0$
while $\neg$ *stop* do
  for $i = 0$ to y do
    Execute next step $\varphi_x(i)$
    if $\varphi_x(i) \downarrow$ then stop $= 1$
  $y = y + 1$
return 1

8.2 Insiemi ricorsivi

Esiste un algoritmo che decide se un elemento (numero) appartiene o meno ad un insieme. Nel caso degli insiemi ricorsivi, al contrario di quelli ricorsivamente enumerabili, è che riesco a dire sia se un elemento appartiene che se non appartiene all'insieme.

Definizione 8.2: Insieme ricorsivo (R)

Un insieme $I \subseteq \mathbb{N}$ è ricorsivo se la sua **funzione caratteristica**

$$\chi_I(x) = \begin{cases} 1 & x \in I \\ 0 & x \notin I \end{cases}$$

è totale ricorsiva.

Per dimostrare che un insieme è ricorsivo, gli dobbiamo fornire l'algoritmo che calcola la sua funzione caratteristica.

Esempio 8.2

Per esempio:

$$\{x \mid x \text{ è pari}\}$$

$$\begin{cases} input(x) \\ \text{if } x \bmod 2 = 0 \text{ then output}(1) \\ \text{else output}(0) \end{cases}$$

Quindi f è la funzione caratteristica dell'insieme dei numeri pari:

$$f(x) = \begin{cases} 1 & x \text{ è pari} \\ 0 & x \text{ è dispari} \end{cases}$$

Esempio 8.3

Prendiamo l'insieme:

$$I = \{x \mid \exists n \,.\, x = n^2\}$$

dove

$$\begin{cases} input(x) \\ n = 0 \\ \text{while } n^2 \neq x \text{ do } \{n = n + 1\} \\ \text{? return 1} \end{cases}$$

Questo programma **NON** è la caratteristica perché non può essere totale.

$$\begin{cases} input(x) \\ n = 0 \\ \text{while } n^2 \leq x \text{ do} \\ \quad \text{if } n^2 = x \text{ then output}(1) \\ \quad n{+}{+} \\ output(0) \end{cases}$$

Questa funzione invece è totale e

$$f(x) = \begin{cases} 1 & \text{se } \exists n \,.\, x = n^2 \\ 0 & \text{altrimenti} \end{cases}$$

Teorema 8.2.1

A ricorsiva $\implies$ A ricorsivamente enumerabile e

$$\exists f_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

ma allora possiamo definire:

$$\psi_A(x) = \begin{cases} 1 & f_A(x) = 1 \\ \uparrow & \text{se } f_A(x) = 0 \end{cases}$$

Supponiamo che C_a è l'algoritmo che calcola f_A

$$\begin{cases} input(x) \\ y = C_a(x) \\ \text{if } y = 1 \text{ then output(y)} \\ \text{else loop infinito} \end{cases}$$

Quindi questa è la semicaratteristica di A.

Teorema 8.2.2 Teorema di Post

Un insieme A è ricorsivo sse A e $\bar{A}$ sono ricorsivamente enumerabili.

Proof. Se A è ricorsiva allora A è RE. Analogamente costruiamo la semicaratteristica per $\bar{A}$:

$$f_{\bar{A}}(x) = \begin{cases} 1 & \text{se } f_A(x) = 0 \\ \uparrow & \text{altrimenti} \end{cases}$$

Siano A e $\bar{A}$ RE.

$$\psi_A(x) = \begin{cases} 1 & x \in A \\ \uparrow & \text{altrimenti} \end{cases} \qquad \psi_{\bar{A}}(x) = \begin{cases} 1 & x \in \bar{A} \\ \uparrow & \text{altrimenti} \end{cases}$$

$$\begin{cases} input(x) \\ stop = 0 \\ \text{while } \neg\, stop \text{ do } \{ \\ \quad \text{Execute next step } \psi_A(x) \\ \quad \text{if } \psi_A(x) \downarrow \text{ then } \{output(1); stop = 1\} \\ \quad \text{Execute next step } \psi_{\bar{A}}(x) \\ \quad \text{if } \psi_{\bar{A}}(x) \downarrow \text{ then } \{output(0); stop = 1\} \\ \} \end{cases}$$

Esempio 8.4 (Esercizio)

$$A_m = \{x \mid \varphi_x(x) \downarrow \text{ in meno di } m \text{ passi}\}$$

È intuitivamente ricorsivo perché è dato da un input finito ovvero m.
Per dimostrare che è ricorsivo basta fornire l'algoritmo della funzione caratteristica:

$$f_{A_m}(x) = \begin{cases} 1 & x \in A_m \\ 0 & x \notin A_m \end{cases}$$

$$\begin{cases} input(x) \\ \text{costruiamo } \varphi_x \\ \text{for } i = 1 \text{ to } m \text{ do} \\ \quad \text{Execute next step } \varphi_x(x) \\ \quad \text{if } \varphi_x(x) \downarrow \text{ then output}(1) \\ output(0) \end{cases}$$