Note per il corso di

Teoria degli automi e calcolabilità

Laurea in Informatica

a.a. 2024/25

Elena Zucca[1]

25 febbraio 2025

[1]Con contributi di Giorgio Delzanno e Francesco Dagnino.

Indice

1 Preliminari **5**
- 1.1 Alfabeti, stringhe, linguaggi [richiami da LPO] . . . 5
- 1.2 Sistemi di transizione e macchine astratte . . . 6

2 Automi e linguaggi formali **9**
- 2.1 Linguaggi regolari . . . 9
  - 2.1.1 Automi a stati finiti . . . 9
  - 2.1.2 Espressioni regolari [approfondimenti da LPO] . . . 17
  - 2.1.3 Proprietà dei linguaggi regolari . . . 19
- 2.2 Linguaggi context-free . . . 21
  - 2.2.1 Grammatiche context-free [richiami da LPO] . . . 21
  - 2.2.2 Grammatiche come definizioni induttive . . . 22
  - 2.2.3 Automi a pila . . . 24
  - 2.2.4 Proprietà dei linguaggi context-free . . . 29

3 Teoria della calcolabilità **33**
- 3.1 Nozione di algoritmo e funzioni ricorsive primitive . . . 33
- 3.2 Macchine di Turing e funzioni ricorsive . . . 38
  - 3.2.1 Macchine di Turing . . . 38
  - 3.2.2 Funzioni T-calcolabili . . . 43
  - 3.2.3 Tesi di Church-Turing . . . 44
  - 3.2.4 Numerazione algoritmica delle funzioni ricorsive . . . 45
  - 3.2.5 Macchina di Turing universale e calcolatore . . . 46
- 3.3 Risolvibilità di problemi . . . 47
  - 3.3.1 Problema dell'arresto . . . 47
  - 3.3.2 Insiemi ricorsivi e ricorsivamente enumerabili . . . 48
  - 3.3.3 Problemi decidibili e indecidibili . . . 52
  - 3.3.4 Riducibilità . . . 52
  - 3.3.5 Teorema di Rice . . . 54
- 3.4 Altri modelli di calcolo . . . 56
  - 3.4.1 Funzioni μ-ricorsive . . . 56
  - 3.4.2 Varianti delle macchine di Turing . . . 57

4 Appendice **61**
- 4.1 Relazioni e funzioni . . . 61
- 4.2 Cardinalità . . . 62
- 4.3 Famiglie . . . 63

4.4 Definizioni induttive 64
4.5 Principio di induzione 64
4.6 Definizioni induttive multiple 66

Capitolo 1

Preliminari

1.1 Alfabeti, stringhe, linguaggi [richiami da LPO]

Def. 1.1.1 [Alfabeto] Un *alfabeto* è un insieme finito non vuoto di oggetti detti *simboli*.

Def. 1.1.2 [Stringa] Una *stringa* u su un alfabeto Σ è una funzione totale[1] da $[1, n]$ in Σ, per qualche $n \in \mathbb{N}$; n si dice *lunghezza* di u e si indica con $|u|$.

Nel seguito useremo σ per indicare generici simboli e u, v, w per indicare generiche stringhe.
L'unica stringa u di lunghezza 0 si chiama *stringa vuota* e si indica con Λ oppure con ϵ; l'insieme delle stringhe su Σ si indica con $\Sigma^\star$ e l'insieme delle stringhe non vuote su Σ si indica con Σ^+.

Def. 1.1.3 [Linguaggio] Un *linguaggio* su un alfabeto Σ è un insieme di stringhe su Σ, ossia un sottoinsieme di $\Sigma^\star$.

Nel seguito useremo L per indicare generici linguaggi. Notiamo che $\Sigma^\star$ è sempre un insieme infinito[2]. L'insieme vuoto $\emptyset$ e l'insieme costituito solo dalla stringa vuota $\{\epsilon\}$ (attenzione alla differenza!) sono linguaggi su qualunque alfabeto.
In genere, scriviamo le stringhe utilizzando la rappresentazione *per giustapposizione*, cioè semplicemente scrivendo i simboli uno dopo l'altro da sinistra a destra.[3]
Consideriamo nel seguito un alfabeto fissato Σ.

Def. 1.1.4 [Operazioni su stringhe] Se u e v sono stringhe di lunghezza n ed m rispettivamente, allora $u \cdot v$ è la stringa di lunghezza $n + m$, definita da

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

È facile vedere che l'operazione di concatenazione di stringhe è associativa, quindi possiamo utilizzare la notazione $u_1 \cdot \ldots \cdot u_n$, e ha come identità la stringa vuota[4].
Inoltre u^n, per $n \geq 0$, è definita induttivamente da $u^0 = \epsilon$, $u^{n+1} = u \cdot u^n$.

[1] Vedi la Definizione 4.1.1.

[2] Più precisamente è un insieme numerabile, ossia in corrispondenza biunivoca con l'insieme $\mathbb{N}$ dei numeri naturali. Perché?

[3] Tuttavia questa rappresentazione è arbitraria e può risultare ambigua, mentre la Definizione 1.1.2 è rigorosa e indipendente dalla rappresentazione, per cui è conveniente richiamarsi a questa definizione se sorge qualche problema di ambiguità. Nella pratica, si può invece utilizzare in ogni caso concreto una rappresentazione non ambigua.

[4] Si tratta quindi di un *monoide*.

La stringa $u \cdot v$ viene anche scritta semplicemente uv.

Def. 1.1.5 [Operazioni su linguaggi] Se L e L' sono linguaggi, $L \cdot L' = \{u \cdot v \mid u \in L, v \in L'\}$. Scriveremo anche semplicemente LL'. Inoltre L^n, per $n \geq 0$, è definito induttivamente da $L^0 = \{\epsilon\}$, $L^{n+1} = L \cdot L^n$. Infine, la *chiusura di Kleene* $L^\star$ di un linguaggio L è definita da $L^\star = \cup_{n \geq 0} L^n$, e la *chiusura positiva* L^+ da $L^+ = \cup_{n>0} L^n$.

È facile vedere che anche l'operazione di concatenazione di linguaggi è associativa, quindi possiamo utilizzare la notazione $L_1 \cdot \ldots \cdot L_n$, e ha come identità l'insieme $\{\epsilon\}$, mentre l'insieme vuoto costituisce uno zero dell'operazione, ossia $L \cdot \emptyset = \emptyset \cdot L = \emptyset$. Si noti la coerenza con le precedenti definizioni di $\Sigma^\star$ e Σ^+, infatti possiamo vedere $L^\star$ come l'insieme delle stringhe sull'alfabeto L, ossia i cui simboli sono a loro volta stringhe.
Si chiama anche *linguaggio*, con abuso di terminologia, una *famiglia* di insiemi di stringhe, vedi la Definizione 4.3.1. Per esempio, nel caso dei linguaggi di programmazione, vi sono diversi insiemi di oggetti sintattici, come programmi, comandi, dichiarazioni, espressioni ecc., e si usa "linguaggio" in questa accezione quando si sia interessati a tutti questi insiemi e non solamente a un insieme "principale".

1.2 Sistemi di transizione e macchine astratte

Un *transition system* (sistema di transizione) consiste di:

- un insieme *Conf* di *configurazioni*
- una relazione di *riduzione* (o *transizione*) su *Conf*, che indichiamo con $\rightarrow$.

Indichiamo con $\rightarrow^\star$ la chiusura riflessiva e transitiva (Definizione 4.1.4) di $\rightarrow$.
Diciamo che una configurazione c è *ferma*, o che è una *configurazione di arresto* (halting configuration), e scriviamo $c \not\rightarrow$, se non esiste c' tale che $c \rightarrow c'$. Indicheremo con $\mathit{Conf}_{\mathrm{H}}$ il sottoinsieme delle configurazioni di arresto.
Una *computazione finita* è una sequenza di passi $c_0 \rightarrow \ldots \rightarrow c_n$, con $n \geq 0$, tale che $c_n \not\rightarrow$.
Una *computazione infinita* è una sequenza infinita di passi $c_0 \rightarrow \ldots \rightarrow c_i \rightarrow \ldots$.
La relazione di riduzione è *deterministica* se per ogni $c \in \mathit{Conf}$ esiste al più una configurazione c' tale che $c \rightarrow c'$. Ossia, la relazione di riduzione è una funzione parziale (Definizione 4.1.1). Si noti che nel caso non deterministico, a partire da una certa configurazione, si ha un *albero* di computazioni nel quale i cammini (finiti o infiniti) sono le computazioni e le foglie sono le configurazioni di arresto. Nel caso deterministico si ha un'unica computazione.
La relazione di riduzione è *terminante* se non ci sono computazioni infinite.
Nel seguito vedremo diversi formalismi di *macchine astratte* (automi). In particolare, nella prima parte del corso consideremo automi *riconoscitori*. In questo caso, avremo i seguenti ingredienti.

- Macchine: per ogni alfabeto Σ, una classe $\mathbb{M}_\Sigma$ di automi che riconoscono linguaggi su Σ.
- Direttive di esecuzione: un modo di associare, ad una macchina $\mathcal{M}$, un sistema di transizione $\rightarrow$. In questo contesto le configurazioni si chiamano anche *descrizioni istantanee*.
- Direttive di input/output: una funzione totale $f_{\mathrm{IN}} \colon \Sigma^\star \rightarrow \mathit{Conf}$ che per ogni stringa in input restituisce una configurazione (detta *configurazione iniziale*), ed una funzione totale $f_{\mathrm{OUT}} \colon \mathit{Conf} \rightarrow \{\mathrm{T}, \mathrm{F}\}$ che "estrae" da una configurazione una risposta positiva o negativa.

Date le direttive di esecuzione e di i/o (che insieme forniscono un *interprete* per la macchina), possiamo definire per ogni $\mathcal{M}$ il linguaggio accettato dalla macchina:

$$L(\mathcal{M}) = \{u \in \Sigma^\star \mid f_{\text{IN}}(u) \to^\star c, f_{\text{OUT}}(c) = \text{T}, \text{per qualche } c \in \mathit{Conf}\}.$$

In altre parole una stringa u su Σ è accettata se la macchina, partendo dalla configurazione iniziale associata ad u, $f_{\text{IN}}(u)$, raggiunge una configurazione c di accettazione, ossia tale che $f_{\text{OUT}}(c) = \text{T}$. Si noti che nel caso non deterministico basta che venga raggiunta *una* configurazione con risposta positiva.
Nella seconda parte del corso, consideremo automi *calcolatori*, ossia che calcolano funzioni $f\colon \mathit{Input} \to \mathit{Output}$. In questo caso gli ingredienti diventano.

- Macchine: un'unica classe $\mathbb{M}$ (la macchina può codificare diversi tipi di input ed output attraverso le direttive di i/o, vedi sotto).
- Direttive di esecuzione: come sopra, un modo di associare, ad una macchina $\mathcal{M}$, un sistema di transizione $\to$.
- Direttive di input/output: una funzione totale $f_{\text{IN}}\colon \mathit{Input} \to \mathit{Conf}$ che per ogni input restituisce una configurazione iniziale, ed una funzione *parziale* $f_{\text{OUT}}\colon \mathit{Conf}_{\text{H}} \to \mathit{Output}$ che "estrae" da una configurazione di arresto un output.

In altre parole, dato un input i, si ottiene un output o se la macchina, partendo dalla configurazione iniziale associata ad i, $f_{\text{IN}}(i)$, si *ferma* in una configurazione c dalla quale si estrae l'output o, ossia tale che $f_{\text{OUT}}(c) = o$.
La funzione f_{OUT} è parziale perché, come vedremo, in certi casi la computazione può fermarsi in configurazioni "mal formate" (errori a runtime) dalle quali non si può estrarre un output.
Date le direttive di esecuzione e di i/o (che insieme forniscono un *interprete* per la macchina), nel caso deterministico[5] possiamo definire per ogni $\mathcal{M} \in \mathbb{M}$ la *funzione calcolata* dalla macchina $f_{\mathcal{M}}\colon \mathit{Input} \to \mathit{Output}$:

$$\text{per ogni } i \in \mathit{Input},\ f_{\mathcal{M}}(i) = o \text{ se } f_{\text{IN}}(i) \to^\star c \text{ tale che } f_{\text{OUT}}(c) = o, \text{ per qualche } c \in \mathit{Conf}_{\text{H}}.$$

Si noti che la funzione calcolata è in generale una funzione parziale, in quanto su qualche input la computazione potrebbe non terminare, o terminare in una configurazione di arresto mal formata.

[5] Più in generale, basta che per ogni input tutte le computazioni forniscano lo stesso risultato (output o indefinito).

Capitolo 2

Automi e linguaggi formali

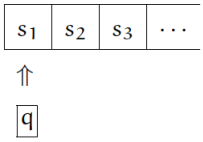
2.1 Linguaggi regolari

2.1.1 Automi a stati finiti

Def. 2.1.1 Un *automa a stati finiti deterministico* (DFA) è una quintupla $\mathcal{M} = \langle Q, \Sigma, \delta, q_0, F \rangle$ dove:

- Q è un insieme finito di *stati*
- Σ è un alfabeto (alfabeto di input)
- $\delta \colon Q \times \Sigma \to Q$ è una funzione totale detta *funzione di transizione*
- $q_0 \in Q$ è lo *stato iniziale*
- $F \subseteq Q$ è l'insieme degli stati *finali*.

Un DFA può essere immaginato come una testina (in un certo stato) che legge, spostandosi sempre nella stessa direzione, un nastro contenente dei simboli, come illustrato in figura.



A seconda dello stato del simbolo letto la testina si porta in un altro stato (o rimane nello stesso) e si sposta a destra. Quando la lettura dei simboli termina, a seconda dello stato della testina, l'automa fornisce un risultato di accettazione o rifiuto.
Con riferimento alle nozioni generali introdotte nella Sezione 1.2, dato $\mathcal{M} = \langle Q, \Sigma, \delta, q_0, F \rangle$, possiamo definire il sistema di transizione associato nel modo seguente.

- Le configurazioni sono della forma $\langle q, u \rangle$ con $q \in Q$, $u \in \Sigma^\star$ (come illustrato dalla figura sopra).
- La relazione di riduzione è definita nel modo seguente:

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

  Si noti che la relazione di riduzione è deterministica, e inoltre terminante (a ogni passo di computazione si consuma un simbolo).

- Le configurazioni di arresto risultano quindi essere quelle della forma $\langle q, \epsilon \rangle$. Infatti, essendo la funzione di transizione totale, un DFA non si blocca mai prima di aver letto tutto l'input.[1]

Le direttive di input/output sono $f_{\text{IN}}(u) = \langle q_0, u \rangle$, e $f_{\text{OUT}}(\langle q, u \rangle) = \begin{cases} \text{T} & \text{se } q \in F, u = \epsilon \\ \text{F} & \text{altrimenti.} \end{cases}$

Il linguaggio $L(\mathcal{M})$ accettato (riconosciuto) da $\mathcal{M}$ risulta quindi essere:

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

Un modo equivalente per definire il linguaggio accettato, utilizzato generalmente nei testi, è il seguente. Definiamo induttivamente la funzione $\hat{\delta} \colon Q \times \Sigma^\star \to Q$ nel modo seguente:

$$\hat{\delta}(q, \epsilon) = q$$
$$\hat{\delta}(q, u\sigma) = \delta(\hat{\delta}(q, u), \sigma)$$

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

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

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

Esercizio 2.1.2 Proviamo che $\emptyset$, $\{\epsilon\}$ e $\Sigma^\star$ sono insieme regolari. Infatti, è immediato vedere che:

- $\emptyset$ è il linguaggio accettato da un qualunque automa con insieme di stati finali vuoto
- $\{\epsilon\}$ è il linguaggio accettato da un automa con stato iniziale e finale q_0, un altro stato q_1 non finale, e $\delta(q_0, \sigma) = q_1$, $\delta(q_1, \sigma) = q_1$ per ogni simbolo dell'alfabeto σ
- $\Sigma^\star$ è il linguaggio accettato da un automa con stato iniziale e finale q_0, e $\delta(q_0, \sigma) = q_0$ per ogni simbolo dell'alfabeto σ.

Un DFA può essere rappresentato come un grafo orientato etichettato detto *grafo di transizione*, oppure dando una *matrice di transizione*, come illustrato dai seguenti esempi. Si noti che un DFA accetta una stringa in input se e solo se esiste un cammino i cui archi sono etichettati con i simboli della stringa nell'ordine dallo stato iniziale a uno stato finale.

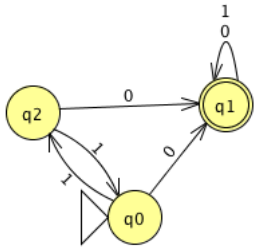
Esempio 2.1.3 Determiniamo il linguaggio accettato dal seguente DFA, rappresentato come matrice di transizione. Lo stato iniziale è indicato da una freccia entrante e gli stati finali da un asterisco.

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

La rappresentazione come grafo di transizione è la seguente. Lo stato iniziale è indicato da una freccia entrante e gli stati finali sono evidenziati con un doppio cerchio.[2]

[1] Si potrebbe dare una definizione alternativa in cui la funzione di transizione δ può essere parziale, interpretando questa come un'abbreviazione per un DFA equivalente con funzione di transizione totale δ' e uno "stato morto" in più q_{dead}, con δ' definita come δ in tutti i casi in cui δ è definita, e inoltre $\delta'(q, \sigma) = q_{dead}$ se $\delta(q, \sigma)$ indefinito e $\delta'(q_{dead}, \sigma) = q_{dead}$ per ogni σ.

[2] La maggior parte dei disegni di automi in queste note sono realizzati con JFLAP (`www.jflap.org`), uno strumento che permette di scrivere facilmente automi ed effettuare test e conversioni automatiche.



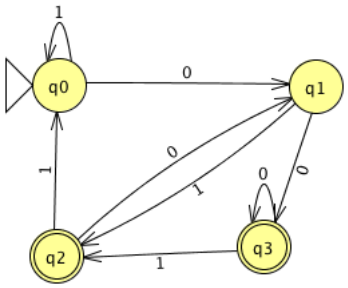
È facile provare che il linguaggio accettato consiste delle stringhe che contengono almeno uno zero. Ossia, ogni stringa della forma $1^n 0u$ con $n \geq 0$ è accettata[3], e ogni stringa della forma 1^n con $n \geq 0$ è rifiutata. Infatti è immediato osservare (e potrebbe essere formalizzato per induzione aritmetica) che a partire da q_0 oppure q_2 leggendo una stringa di 1^n con $n \geq 0$ si arriva in q_0 oppure q_2.

Nel seguito, utilizzeremo talvolta per maggiore leggibilità la notazione $q \xrightarrow{u} q'$ per $\hat{\delta}(q, u) = q'$.

Esempio 2.1.4 Sia $\Sigma = \{0, 1\}$. Proviamo che il linguaggio formato da tutte le stringhe in cui il penultimo simbolo è uno zero è regolare.
Per provare che un linguaggio è regolare è sufficiente dare un DFA che accetti il linguaggio. Possiamo costruire questo DFA ragionando nel modo seguente:

- nello stato iniziale q_0 non ho ancora letto uno 0, quindi questo stato non deve essere finale
- se dallo stato iniziale leggo un 1, resto nello stato q_0 ("non ho ancora letto uno 0 che sia possibile penultimo")
- se dallo stato iniziale leggo uno 0, passo nello stato q_1 ("se leggo ancora solo un simbolo ok")
- se da q_1 leggo un 1, passo nello stato q_2, che dovrà essere finale ("ho letto uno 0 e un 1, se non leggo più simboli ok")
- se da q_1 leggo uno 0, passo nello stato q_3, che dovrà essere finale ("ho letto due zeri, se non leggo più simboli ok")
- se da q_2 leggo uno 0, passo nello stato q_1, in quanto avendo letto un 1 e uno 0 "se leggo ancora solo un simbolo ok"
- se da q_2 leggo un 1, torno nello stato q_0, in quanto avendo letto due 1 "non ho ancora letto uno 0 che sia possibile penultimo"
- se da q_3 leggo uno 0, resto nello stato q_3, in quanto ho letto due zeri
- se da q_3 leggo un 1, passo nello stato q_2, in quanto ho letto uno 0 e un 1.

Otteniamo quindi il seguente automa dove q_2 e q_3 sono finali:



[3]Se una stringa contiene uno 0 contiene anche un *primo* 0.

Dimostriamo ora che l'automa così costruito effettivamente riconosce il linguaggio dato. Anzitutto notiamo che le stringhe di lunghezza minore di due non sono accettate (correttamente), in quanto q_0 non è finale e $q_0 \stackrel{0}{\to} q_1$, $q_0 \stackrel{1}{\to} q_0$. Consideriamo ora le stringhe di lunghezza almeno due, ossia della forma $u0\sigma$ oppure $u1\sigma$. Notiamo che si ha, qualunque sia lo stato q, che $q \stackrel{00}{\to} q_3$, $q \stackrel{01}{\to} q_2$, $q \stackrel{10}{\to} q_1$, $q \stackrel{11}{\to} q_0$. Di conseguenza si ha anche, qualunque sia lo stato q, che $q \stackrel{u00}{\to} q_3$, $q \stackrel{u01}{\to} q_2$, $q \stackrel{u10}{\to} q_1$, $q \stackrel{u11}{\to} q_0$ (si noti che in questo caso non serve ragionare per induzione in quanto $q \stackrel{u00}{\to} q_3$ se $q \stackrel{u}{\to} q'$ e $q' \stackrel{00}{\to} q_3$, e analogamente per gli altri casi).

Def. 2.1.5 Un *automa a stati finiti non deterministico* (NFA) è una quintupla $\mathcal{M} = \langle Q, \Sigma, \delta, q_0, F\rangle$ dove Q, Σ, q_0, F sono come per i DFA, mentre la funzione di transizione ha forma $\delta\colon\ Q \times \Sigma \to \wp(Q)$.

Ricordiamo che $\wp(Q)$ denota l'insieme delle parti di Q, ossia l'insieme i cui elementi sono i sottoinsiemi di Q.
La rappresentazione a tabella deve ora prevedere in ogni casella un insieme di stati, mentre la rappresentazione a grafo rimane immutata.
Il sistema di transizione associato è definito analogamente a quello per un DFA, ma vi sono delle differenze.

- Le configurazioni sono sempre della forma $\langle q, u\rangle$ con $q \in Q$, $u \in \Sigma^\star$.
- La relazione di riduzione è definita nel modo seguente:

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

  La relazione di riduzione non è più deterministica, ma sempre terminante.
- In questo caso le configurazioni di arresto risultano essere, oltre a quelle della forma $\langle q, \epsilon\rangle$, anche quelle della forma $\langle q, \sigma u\rangle$ con $\delta(q, \sigma) = \emptyset$. Infatti è possibile che per un certo stato e simbolo *non* vi sia transizione possibile, quindi l'automa si blocchi prima di aver letto tutto l'input.

Le direttive di input/output sono come prima.
Il linguaggio $L(\mathcal{M})$ accettato (riconosciuto) da $\mathcal{M}$ risulta essere, come prima:

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

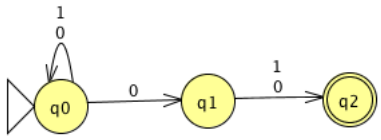
Si noti che una stringa u è accettata se *esiste* una configurazione per cui la macchina con input u restituisce T.
Un modo equivalente per definire il linguaggio accettato, utilizzato generalmente nei testi, è il seguente.
Definiamo $\hat{\delta}\colon\ Q \times \Sigma^\star \to \wp(Q)$ nel modo seguente:

$$\hat{\delta}(q, \epsilon) = \{q\}$$
$$\hat{\delta}(q, u\sigma) = \bigcup_{q' \in \hat{\delta}(q,u)} \delta(q', \sigma)$$

e diciamo che una stringa u è accettata se *esiste* uno stato in $\hat{\delta}(q_0, u)$ che sia finale. Il linguaggio accettato (riconosciuto) da $\mathcal{M}$ è quindi:

$$L(\mathcal{M}) = \{u \mid \hat{\delta}(q_0, u) \cap F \neq \emptyset\}$$

Esempio 2.1.6 Sia $\Sigma = \{0, 1\}$. Diamo un NFA per il linguaggio formato da tutte le stringhe in cui il penultimo simbolo è uno zero.
Possiamo costruire questo NFA ragionando in modo più semplice, e ottenendo un automa più compatto, di quanto visto prima nel caso deterministico: partendo dallo stato iniziale q_0, finché leggo degli 1 resto sicuramente in q_0, mentre quando leggo uno 0 posso non deterministicamente restare in q_0 (lo 0 che ho letto non è il penultimo simbolo), oppure passare in q_1 e poi qualunque sia il simbolo letto nello stato finale q_2, nel quale non è più possibile leggere ulteriori simboli. Otteniamo quindi:



Proviamo ora che le classi dei linguaggi accettati dai DFA e dagli NFA coincidono. Un'inclusione è ovvia in quanto un DFA è un caso particolare di NFA. Per l'inclusione inversa, si può convertire un NFA in un DFA con l'idea seguente:

- gli stati del DFA sono insiemi di stati del NFA
- per ogni stato q_D del DFA, che quindi è un insieme di stati del NFA, per ogni simbolo si ha una transizione nell'insieme degli stati ottenuti con transizioni da stati in q_D nel NFA .

Formalmente si ha il seguente teorema.

Teorema 2.1.7 [Rabin-Scott, 1959] Sia $\mathcal{M} = \langle Q, \Sigma, \delta_N, q_0, F_N \rangle$ un NFA. Allora esiste un DFA $\mathcal{M}_D$ tale che $L(\mathcal{M}_D) = L(\mathcal{M})$.

Prova Costruiamo $\mathcal{M}_D$ come la quintupla $\langle \wp(Q), \Sigma, \delta_D, \{q_0\}, F_D \rangle$ dove:

- $\delta_D(q_D, \sigma) = \bigcup_{q \in q_D} \delta(q, \sigma)$ per $q_D \in \wp(Q)$
  ossia, $q' \in \delta_D(q_D, \sigma)$ se e solo se $q' \in \delta(q, \sigma)$ per qualche $q \in q_D$
- $F_D = \{q_D \subseteq Q \mid q_D \cap F_N \neq \emptyset\}$
  ossia, $q_D \in F_D$ se e solo $q \in F_N$ per qualche $q \in q_D$.

Proviamo per induzione aritmetica sulla lunghezza della stringa che, per ogni $u \in \Sigma^\star$, $\hat{\delta}_D(\{q_0\}, u) = \hat{\delta}_N(q_0, u)$. Si noti che in entrambi i casi il risultato è un elemento di $\wp(Q)$, ossia un insieme di stati.

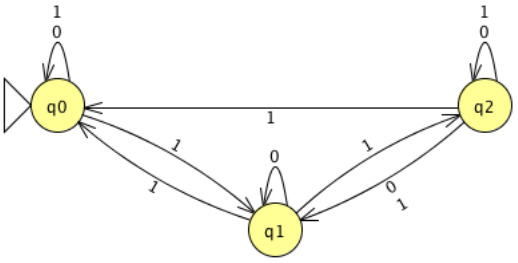
Base Per la stringa vuota si ha $\hat{\delta}_D(\{q_0\}, \epsilon) = \{q_0\}$ per definizione di $\hat{\delta}_D$ (caso deterministico), e $\hat{\delta}_N(q_0, \epsilon) = \{q_0\}$ per definizione di $\hat{\delta}_N$ (caso non deterministico).

Passo induttivo Consideriamo una stringa $u\sigma$. Per ipotesi induttiva, si ha che $\hat{\delta}_D(\{q_0\}, u) = \hat{\delta}_N(q_0, u)$, e vogliamo provare che si ha anche $\hat{\delta}_D(\{q_0\}, u\sigma) = \hat{\delta}_N(q_0, u\sigma)$. Infatti si ha:

$\hat{\delta}_D(\{q_0\}, u\sigma) =$
$\delta_D(\hat{\delta}_D(\{q_0\}, u), \sigma)$ per definizione di $\hat{\delta}_D$ (caso deterministico) =
$\delta_D(\hat{\delta}_N(q_0, u), \sigma)$ per ipotesi induttiva (si noti che $\hat{\delta}_N(q_0, u)$ è un insieme di stati) =
$\bigcup_{q \in \hat{\delta}_N(q_0, u)} \delta_N(q, \sigma)$ per definizione di δ_D =
$\hat{\delta}_N(q_0, u\sigma)$ per definizione di $\hat{\delta}_N$ (caso non deterministico).

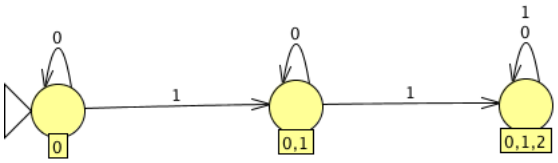
Avendo provato che, per ogni $u \in \Sigma^\star$, $\hat{\delta}_D(\{q_0\}, u) = \hat{\delta}_N(q_0, u)$, è facile vedere che u è accettata da $\mathcal{M}_D$ se e solo se è accettata da $\mathcal{M}_N$. Infatti u è accettata da $\mathcal{M}_D$ se e solo se $\hat{\delta}_D(\{q_0\}, u) \in F_D$, ossia, per definizione, se e solo se $\hat{\delta}_D(\{q_0\}, u) = \hat{\delta}_N(q_0, u) \cap F_N \neq \emptyset$, ossia se e solo se u è accettata da $\mathcal{M}_N$. □

Esempio 2.1.8 Applichiamo la costruzione del precedente teorema al seguente NFA:



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

Per semplicità, consideriamo solo gli insiemi di stati che sono raggiungibili a partire dallo stato iniziale $\{q_0\}$, infatti è chiaro che un automa può sempre essere trasformato in uno equivalente eliminando tutti gli stati non raggiungibili. Otteniamo:



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

Il linguaggio accettato è l'insieme delle stringhe che contengono almeno due 1 (lo si giustifichi per esercizio).

Esercizio 2.1.9 Sia $\Sigma = \{0, 1\}$. Si consideri l'automa non deterministico che accetta il linguaggio formato da tutte le stringhe in cui il penultimo simbolo è uno zero visto sopra. Si applichi a questo NFA la costruzione del teorema precedente e lo si confronti con quello dell'esempio 2.1.4.

Def. 2.1.10 Un *ϵ-NFA*, o NFA *con transizioni silenti* è una quintupla $\mathcal{M} = \langle Q, \Sigma, \delta, q_0, F\rangle$ dove Q, Σ, q_0, F sono come per gli NFA, mentre la funzione di transizione ha forma $\delta\colon Q \times (\Sigma \cup \{\epsilon\}) \to \wp(Q)$.

Intuitivamente, è permesso passare da uno stato a un altro anche senza leggere simboli di input. Questa possibilità permette in molti casi di scrivere automi più semplici e leggibili, si consideri per esempio il linguaggio $\{a^n b^m c^k \mid n, m, k \geq 0\}$.
NB: l'uso del simbolo ϵ per le transizioni silenti è assolutamente arbitrario, potremmo utilizzare qualunque altro simbolo; scegliere proprio lo stesso che utilizziamo per indicare la stringa vuota serve solo a suggerire l'idea che non c'è lettura.
Il sistema di transizione associato è definito nel modo seguente.

- Le configurazioni sono sempre della forma $\langle q, u\rangle$ con $q \in Q$, $u \in \Sigma^\star$.
- La relazione di riduzione è definita nel modo seguente:

  $\langle q, \sigma u\rangle \to \langle q', u\rangle$ se $q' \in \delta(q, \sigma)$
  $\langle q, u\rangle \to \langle q', u\rangle$ se $q' \in \delta(q, \epsilon)$

  La relazione di riduzione è non deterministica (si noti che in questo caso vi è anche un non determinismo che consiste in leggere o non leggere) e non terminante. Tuttavia, le computazioni infinite sono dovute solo alla presenza presenza di cicli di transizioni ϵ da uno stato q in se stesso, che possono essere eliminati.

- Le configurazioni di arresto sono quelle della forma $\langle q, \sigma u\rangle$ con $\delta(q,\sigma) = \emptyset$ e $\delta(q,\epsilon) = \emptyset$, oppure $\langle q, \epsilon\rangle$ con $\delta(q,\epsilon) = \emptyset$.

Le direttive di input/output sono immutate, e la definizione di linguaggio accettato è immutata.
Un modo equivalente per definire il linguaggio accettato, utilizzato generalmente nei testi, è il seguente.
Anzitutto, dato uno stato q, possiamo definire induttivamente l'insieme ϵ-*closure*(q) di tutti gli stati raggiungibili da q con transizioni silenti:

$q \in \epsilon$-*closure*(q)
se $q' \in \delta(q,\epsilon)$, allora $q' \in \epsilon$-*closure*(q)
se $q' \in \epsilon$-*closure*(q) e $q'' \in \epsilon$-*closure*(q') allora $q'' \in \epsilon$-*closure*(q)

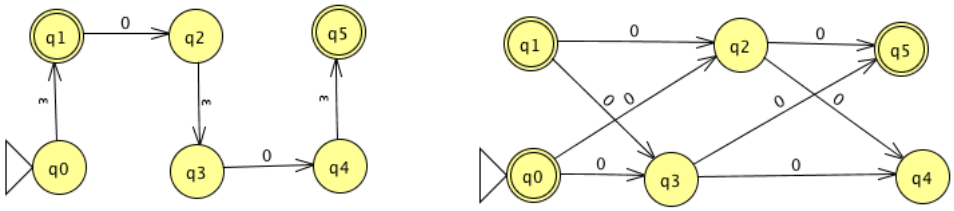
Possiamo ora estendere δ alle stringhe nel modo seguente:

$\hat{\delta}(q,\epsilon) = \epsilon$-*closure*$(q)$
$\hat{\delta}(q,u\sigma) = \bigcup_{q' \in \hat{\delta}(q,u)} \epsilon$-*closure*$(\delta(q',\sigma))$

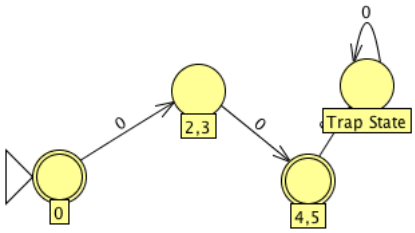
ossia, $\hat{\delta}(q,\sigma_1 \dots \sigma_n)$ è l'insieme degli stati che possiamo ottenere a partire da q con una sequenza di transizioni σ_1, …, σ_n intervallata da eventuali transizioni silenti, e definire il linguaggio accettato come $\{w \mid \hat{\delta}(q_0,w) \cap F \neq \emptyset\}$.
Si osservi che in questo caso $\delta(q,\sigma)$ non coincide necessariamente con $\hat{\delta}(q,\sigma)$, e che in questa classe di automi è chiaramente sempre possibile ridursi ad avere un unico stato finale.
Le classi dei linguaggi accettati dagli NFA e dagli ϵ-NFA coincidono. Un'inclusione è ovvia in quanto un NFA è un caso particolare di ϵ-NFA. Per l'inclusione inversa, dato un ϵ-NFA $\mathcal{M}_\epsilon = \langle Q, \Sigma, \delta_\epsilon, q_0, F_\epsilon\rangle$, costruiamo $\mathcal{M} = \langle Q, \Sigma, \delta, q_0, F\rangle$ nel modo seguente:

- $F = F_\epsilon \cup \{q_0\}$ se ϵ-*closure*$(q_0) \cap F_\epsilon \neq \emptyset$, F_ϵ altrimenti.
- $\delta(q,\sigma) = \hat{\delta}_\epsilon(q,\sigma)$.

Si noti che, per definizione, $\hat{\delta}_\epsilon(q,\sigma) = \bigcup_{q' \in \hat{\delta}(q,\epsilon)} \epsilon$-*closure*$(\delta(q',\sigma)) = \bigcup_{q' \in \epsilon\text{-}closure(q)} \epsilon$-*closure*$(\delta(q',\sigma))$.
In altri termini $\delta(q,\sigma)$ è definita come l'insieme degli stati raggiungibili da q con una sequenza (anche vuota) di transizioni silenti, seguita da una transizione σ, seguita ancora da una sequenza (anche vuota) di transizioni silenti. Non diamo la prova della correttezza della costruzione. Vediamo un esempio:



Applicando poi a sua volta all'automa non deterministico così costruito la trasformazione vista precedentemente si ottiene il seguente automa deterministico:



Il linguaggio accettato è $\{\epsilon, 00\}$. Si noti che ϵ viene accettata in quanto q_0 è finale, e lo stato non finale "morto" corrispondente all'insieme vuoto.
Le due trasformazioni possono anche essere effettuate in un unico passo, in sostanza si applica la trasformazione da non deterministico a deterministico considerando però per ogni transizione anche gli stati raggiungibili con transizioni silenti aggiuntive (si precisi per esercizio, e si applichi la trasformazione in un unico passo al precedente esempio).
Concludiamo la sezione illustrando una tecnica per *minimizzare* i DFA, ossia dato un DFA $\mathcal{M} = \langle Q, \Sigma, \delta, q_0, F\rangle$ trovarne uno equivalente, ossia che accetti lo stesso linguaggio, con il numero minimo di stati.
Anzitutto, come già accennato, è chiaramente possibile eliminare tutti gli stati che non sono raggiungibili a partire da quello iniziale. Inoltre, è possibile far "collassare" più stati in un solo, come illustrato nel seguito.
La nozione di linguaggio accettato può essere estesa a qualunque stato q di un automa. Diciamo che due stati q e q' sono *indistinguibili*, e scriviamo $q{\sim}q'$, se il loro linguaggio accettato è lo stesso, ossia, per qualunque stringa u, $\hat{\delta}(q, u)$ è finale se e solo se $\hat{\delta}(q', u)$ è finale. Chiaramente si tratta di una relazione di *equivalenza*, vedi Definizione 4.1.3. Indichiamo con $[q]_\sim$ la classe di equivalenza di q e con $Q/\sim$ il quoziente di Q rispetto alla relazione di indistinguibilità. Due stati sono quindi *distinguibili*, scriviamo $q \not\sim q'$, se esiste almeno una stringa che viene accettata partendo da uno stato ma non dall'altro. Per esempio, uno stato finale è sempre distinguibile da uno non finale mediante la stringa vuota.
Due stati indistinguibili possono intuitivamente essere trasformati in un unico stato. In altri termini, possiamo costruire l'automa $\mathcal{M}' = \langle Q/\sim, \Sigma, \delta', [q_0]_\sim, F'\rangle$ che ha:

- come stati le classi di equivalenza rispetto a $\sim$
- come stato iniziale la classe di equivalenza dello stato iniziale di $\mathcal{M}$
- come stati finali le classi di equivalenza degli stati finali di $\mathcal{M}$, ossia $[q]_\sim \in F'$ se e solo se $q \in F$
- funzione di transizione definita da $\delta'([q]_\sim, \sigma) = [\delta(q, \sigma)]_\sim$.

È facile vedere che la funzione di transizione è ben definita (infatti se $q{\sim}q'$ non può essere $\delta(q, \sigma) \not\sim \delta(q', \sigma)$) e che il linguaggio accettato da $\mathcal{M}'$ coincide con quello accettato da $\mathcal{M}$ (in quanto $\hat{\delta}'([q_0]_\sim, u) = [\hat{\delta}(q_0, u)]_\sim$).
Si può provare (non lo vediamo) che l'automa definito in questo modo risulta essere *minimale*, nel senso che non è possibile trovare un automa con un numero strettamente inferiore di stati che generi lo stesso linguaggio.
Inoltre, la relazione di indistinguibilità (e quindi l'automa minimale) può essere costruita per approssimazioni successive utilizzando il seguente algoritmo. Inizialmente suddividiamo gli stati in due classi di equivalenza, quelli finali e quelli non finali. Poi, a ogni passo consideriamo le classi di equivalenza ottenute al passo precedente, e per ognuna di esse, sia C, per ogni simbolo dell'alfabeto σ, si controlla se con una transizione σ si arriva, per ognuno degli stati in C, nella stessa classe di equivalenza C'. In caso contrario si scompone ulteriormente C in più classi di equivalenza. L'algoritmo termina quando si arriva a una situazione stabile in cui non si effettuano più modifiche. Una semplice descrizione in pseudocodice dell'algoritmo è la seguente:

```
∼ = {⟨q, q'⟩ | q ∈ F ⇔ q' ∈ F}
repeat
  ∼old = ∼
  ∼ = ∅
  for each ⟨q, q'⟩ in ∼old
    if (δ(q,σ)∼oldδ(q',σ) ∀σ ∈ Σ) ∼ = ∼ ∪ {⟨q, q'⟩}
until ∼ = ∼old
```

Esempio 2.1.11 Consideriamo il seguente DFA:

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

Inizialmente si ha: $\sim = \{\{q_3\}, \{q_0, q_1, q_2\}\}$. Alla prima iterazione, q_0 e q_1 risultano distinguibili da q_2 leggendo b, quindi si ha $\sim = \{\{q_3\}, \{q_0, q_1\}, \{q_2\}\}$. Infine, alla seconda iterazione q_0 risulta distinguibile da q_1 leggendo b, quindi si ha $\sim = \{\{q_3\}, \{q_0\}, \{q_1\}, \{q_2\}\}$ (ossia il DFA era già minimo).

2.1.2 Espressioni regolari [approfondimenti da LPO]

Def. 2.1.12 L'insieme delle *espressioni regolari* (RE) (su Σ) e i linguaggi (su Σ) da esse denotati sono definiti induttivamente nel modo seguente:

- $\emptyset$ è una RE che denota il linguaggio vuoto
- ϵ è una RE che denota $\{\epsilon\}$
- per ogni $\sigma \in \Sigma$, σ è un'espressione regolare che denota $\{\sigma\}$
- se r_1 è un'espressione regolare che denota L_1, ed r_2 è un'espressione regolare che denota L_2, allora $r_1 + r_2$ (alternativamente si usa anche la sintassi $r_1 \mid r_2$) è un'espressione regolare che denota $L_1 \cup L_2$, ed $r_1 r_2$ è un'espressione regolare che denota $L_1 \cdot L_2$
- se r è un'espressione regolare che denota L, $r^\star$ è un'espressione regolare che denota $L^\star$.

Un'espressione regolare è una stringa che descrive schematicamente un insieme di stringhe. Le espressioni regolari sono utilizzate in moltissimi campi dell'informatica, dovendo la loro popolarità principalmente a due fattori:

- permettono di descrivere in modo estremamente sintetico stringhe di varia natura
- a partire da esse è possibile generare automaticamente dei *riconoscitori* (ossia, programmi che stabiliscono se una stringa appartiene al linguaggio oppure no) estremamente efficienti per i linguaggi regolari.

A seconda del campo di applicazione, la sintassi concreta con cui vengono rappresentate le espressioni regolari può cambiare, e vengono in genere aggiunti agli operatori base visti nella definizione precedente molti altri derivati da questi. In questa maniera, si riescono a descrivere insiemi di stringhe in modo molto compatto e flessibile.

Esercizio 2.1.13 Per ognuna delle seguenti identità tra espressioni regolari si dica se è vera, giustificando la risposta.

1. $r + s = s + r$
2. $r + (s + t) = (r + s) + t$
3. $r(st) = (rs)t$

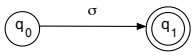
4. $\emptyset^\star = \epsilon$
5. $(r + s)^\star = r^\star + s^\star$
6. $r^\star(r + s)^\star = (r + s)^\star$
7. $(r^\star + s^\star) = (rs)^\star$

Possiamo provare che la classe dei linguaggi denotati da espressioni regolari coincide con quella dei linguaggi riconosciuti dagli automi a stati finiti. A tale scopo, si mostra che, data un'espressione regolare r, è possibile costruire un un NFA con transizioni ϵ che riconosce il linguaggio da essa denotato. Tale automa, con stato iniziale q_0 e un unico stato finale, è definito per induzione, vedi Sezione **??**, sull'espressione regolare r, nel modo seguente.

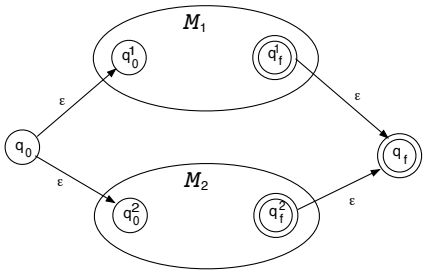
- Il linguaggio $\emptyset$ è riconosciuto dall'automa

- Il linguaggio $\{\epsilon\}$ è riconosciuto dall'automa

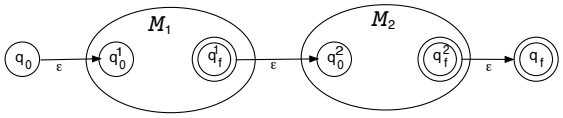
- Il linguaggio $\{\sigma\}$ è riconosciuto dall'automa



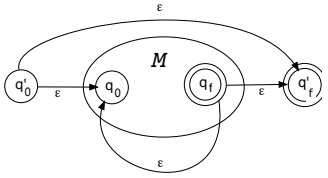
- Se, per $i = 1, 2$, $\mathcal{M}_i$ con stato iniziale q_0^i e finale q_f^i riconosce il linguaggio denotato da r_i, allora il linguaggio denotato da $r_1 + r_2$ è riconosciuto dall'automa



- Se, per $i = 1, 2$, $\mathcal{M}_i$ con stato iniziale q_0^i e finale q_f^i riconosce il linguaggio denotato da r_i, allora il linguaggio denotato da $r_1 r_2$ è riconosciuto dall'automa



- Se $\mathcal{M}$ con stato iniziale q_0 e finale q_f riconosce il linguaggio denotato da r, allora il linguaggio denotato da $r^\star$ è riconosciuto dall'automa



È possibile (non lo vediamo) dare anche la costruzione inversa, che a partire da un automa $\mathcal{M}$ prova l'esistenza di un'espressione regolare che denota il linguaggio riconosciuto da $\mathcal{M}$.
Il fatto che sia possibile costruire in modo algoritmico, a partire da un'espressione regolare, un ϵ-NFA, e da questo un DFA (minimo), costituisce la base teorica per la costruzione di *generatori di analizzatori lessicali*.

2.1.3 Proprietà dei linguaggi regolari

Per dimostrare che un linguaggio non è regolare si usa in genere il *pumping lemma*. Diamo prima un'idea informale.
Intuitivamente, un automa a stati finiti ha solo una memoria finita, data dai suoi diversi stati, e non è quindi in grado di contare o comunque di memorizzare un numero di informazioni non limitato a priori. Dato che è accettato da un automa a stati finiti, un linguaggio regolare è finito oppure contiene sottoinsiemi infiniti non arbitrari, ma formati da stringhe che seguono tutte un certo "pattern" (un'espressione regolare costruita con $\star$, corrispondente a un ciclo nell'automa). Per esempio, possiamo convincerci che il linguaggio $\{0^n 1^n\}$ non è regolare ragionando nel modo seguente.

- Supponiamo (per assurdo) che esista un automa che riconosce questo linguaggio, e sia n il numero di stati dell'automa.
- Supponiamo di fornire in input all'automa una stringa formata di tutti zeri e di lunghezza maggiore di n. L'automa attraverserà una sequenza di stati:
  $q_0 \stackrel{0}{\to} q_1 \stackrel{0}{\to} \ldots \stackrel{0}{\to} q_n \stackrel{0}{\to} \ldots$
- Dato però che l'automa ha solo n stati, necessariamente dovrà passare due volte attraverso lo stesso stato, sia $q_i = q_j = \bar{q}$. Si avrà quindi: $q_0 \stackrel{0^i}{\to} q_i = \bar{q} \stackrel{0\ldots0}{\to} \bar{q}$, con la seconda stringa di 0 non vuota.
- A questo punto, per accettare la stringa $0^i 1^i$, che appartiene al linguaggio, l'automa dovrebbe, a partire dallo stato $\bar{q}$, arrivare a uno stato finale leggendo 1^i. Ma allora accetterebbe anche la stringa $0^i 0 \ldots 0 1^i$ che non appartiene al linguaggio.

Questo ragionamento può essere generalizzato nel modo seguente.

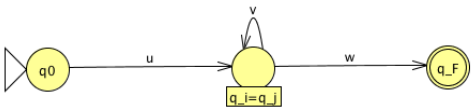
Lemma 2.1.14 [Pumping lemma per i linguaggi regolari (Bar-Hillel, Perles, Shamir, 1961] Sia L un linguaggio regolare. Allora esiste una costante $n \in \mathbb{N}$ tale che, per ogni $z \in L$ con $|z| \geq n$, possiamo decomporre z come uvw in modo che:

1. $|uv| \leq n$,
2. $|v| > 0$,
3. per ogni $i \geq 0$ si ha che $uv^i w \in L$.

Prova Se L è regolare, esiste un DFA che lo riconosce. Scegliamo allora come n il numero degli stati di questo automa. Sia $z = \sigma_1 \ldots \sigma_m$, con $m \geq n$. Utilizziamo, per maggior chiarezza, la notazione $q \stackrel{u}{\to} q'$ per $\hat{\delta}(q, u) = q'$. Dato che la stringa è accettata dall'automa, si ha $q_0 \stackrel{\sigma_1}{\to} q_1 \stackrel{\sigma_2}{\to} \ldots \stackrel{\sigma_m}{\to} q_m = q_F$, con $q_0, \ldots, q_m$ stati dell'automa, q_0 stato iniziale e q_F stato finale. Dato che gli stati dell'automa sono solo n, necessariamente si ha che almeno uno stato, sia $\bar{q}$, viene attraversato due volte nei primi n passi, ossia $\bar{q} = q_i = q_j$ per qualche $0 \leq i < j \leq n$. Allora, ponendo:

- $u = \sigma_1 \dots \sigma_i$
- $v = \sigma_{i+1} \dots \sigma_j$
- $w = \sigma_{j+1} \dots \sigma_m$

si ha che $q_0 \xrightarrow{u} \bar{q} \xrightarrow{v} \bar{q} \xrightarrow{w} q_F$, come schematizzato sotto:



Dato che $\bar{q} \xrightarrow{v} \bar{q}$, è facile vedere che, per ogni $i \geq 0$, si ha $\bar{q} \xrightarrow{v^i} \bar{q}$, e quindi $q_0 \xrightarrow{u} \bar{q} \xrightarrow{v^i} \bar{q} \xrightarrow{w} q_F$. Si noti che $|uv| \leq n$ e $|v| > 0$ per costruzione. □

Il pumping lemma viene tipicamente utilizzato per provare che un linguaggio L *non* è regolare. A tale scopo, occorre mostrare che, fissato un n arbitrario, esiste una stringa z, con $|z| \geq n$, che appartiene al linguaggio, ma tale che, per ogni sua possibile decomposizione come uvw, con $|uv| \leq n$ e $|v| > 0$, $uv^iw \notin L$ per qualche $i \geq 0$. I seguenti esempi illustrano la tecnica.

Esempio 2.1.15 Possiamo dimostrare, utilizzando il pumping lemma, che il linguaggio $\{0^n1^n\}$ non è regolare. Infatti, preso n arbitrario, consideriamo la stringa 0^n1^n. Decomponendo tale stringa in tre parti u, v, w tali che la lunghezza delle prime due sia $\leq n$ e la seconda sia non vuota, si ha chiaramente $u = 0^a$, $v = 0^b$ e $w = 0^c1^n$ con $a+b+c = n, b > 0$. Allora, per esempio per $i = 0$, si ha che $uv^0w = 0^a0^c1^n$ non appartiene al linguaggio in quanto $a + c < n$.

Esempio 2.1.16 Il linguaggio delle stringhe formate da un numero uguale di 0 e di 1 non è regolare. Infatti, preso n arbitrario, consideriamo la stringa 0^n1^n. Decomponendo tale stringa in tre parti u, v, w tali che la lunghezza delle prime due sia $\leq n$ e la seconda sia non vuota, si ha chiaramente $u = 0^a$, $v = 0^b$ e $w = 0^c1^n$ con $a+b+c = n, b > 0$. Allora, per esempio per $i = 0$, si ha che $uv^0w = 0^a0^c1^n$ non ha un numero uguale di 0 e di 1.

Esempio 2.1.17 Il linguaggio delle stringhe 1^p con p numero primo non è regolare. Infatti, preso n arbitrario, scegliamo un numero primo $p \geq n + 2$. Decomponiamo la stringa 1^p in tre parti u, v, w tali che la lunghezza delle prime due sia $\leq n$ e la seconda sia non vuota, e sia m la lunghezza di questa seconda parte. Si ha che $uv^{p-m}w$ non appartiene al linguaggio, ossia la sua lunghezza non è un numero primo. Infatti:

$$|uv^{p-m}w| = |uw| + (p-m)|v| = (p-m) + m(p-m) = (1+m)(p-m)$$

che non è primo, a meno che uno dei fattori non sia 1, ma:

- $1 + m$ è maggiore di 1 poiché $m > 0$
- $p - m$ è maggiore di 1 poiché p supera n (e quindi m) di almeno due.

Proprietà di chiusura: i linguaggi regolari sono chiusi rispetto alle operazioni di unione, concatenazione e chiusura di Kleene (ovvio per definizione), complementazione e intersezione, ossia se L, L' sono linguaggi regolari su Σ, allora lo sono anche:

- $\overline{L} = \Sigma^\star \setminus L$,
- $L \cap L'$.

Infatti, dato $\mathcal{M}$ un DFA che riconosce L, possiamo ottenere da questo un DFA che riconosce $\overline{L}$ semplicemente scambiando l'insieme degli stati finali con quello dei non finali. A questo punto, anche la chiusura rispetto all'intersezione si vede facilmente in quanto $L \cap L' = \overline{\overline{L} \cup \overline{L'}}$.

2.2 Linguaggi context-free

2.2.1 Grammatiche context-free [richiami da LPO]

Def. 2.2.1 [Grammatica CF] Una *grammatica context-free* (*CF*) o *libera da contesto* è una quadrupla $\langle T, N, P, S\rangle$ dove:

- T è l'alfabeto dei *terminali*,
- N è l'alfabeto dei *non terminali*, con $T \cap N = \emptyset$,
- P è un insieme finito di coppie $A ::= \alpha$ dette *produzioni* dove $A \in N$ e $\alpha \in (T \cup N)^\star$,
- $S \in N$ è il *simbolo iniziale* o *assioma* o *start*.

In questa sezione useremo talvolta semplicemente il termine *grammatica*, tuttavia esistono altri tipi di grammatiche, per cui la precisazione "context-free" è rilevante. Riserveremo u, v, w a indicare generiche stringhe di terminali, mentre utilizzeremo α, β, γ per indicare generiche stringhe di terminali e non terminali.
Per esempio, la seguente grammatica, che chiameremo $GExp$:

Exp ::= `(`*Exp* + *Exp*`)` | `(`*Exp* $*$ *Exp*`)` | *Num*
Num ::= `0` | `1` | ... | `9`
Id ::= `a` | `b` | ... | `z`

definisce le espressioni aritmetiche costruite, mediante gli operatori + e $*$, a partire da numeri, formati per semplicità da una sola cifra decimale, e da identificatori formati per semplicità da una sola lettera minuscola. Abbiamo utilizzato, come usuale, l'abbreviazione $A ::= \alpha_1 \mid \ldots \mid \alpha_n$ per indicare le produzioni $A ::= \alpha_1, \ldots, A ::= \alpha_n$.
Per capire in che senso una grammatica definisca un linguaggio introduciamo il concetto di derivazione.

Def. 2.2.2 [Derivazione in un passo] Sia data una grammatica $G = \langle T, N, P, S\rangle$ e siano $\alpha, \beta \in (T \cup N)^\star$. Diremo che β *è derivabile da* α *in un passo* se α è della forma $\alpha_1 A \alpha_2$, β è della forma $\alpha_1\gamma\alpha_2$, ed esiste una produzione $A ::= \gamma$ in P, con $\alpha_1, \alpha_2 \in (T \cup N)^\star$. Diremo anche che $\alpha \to \beta$ è una *derivazione (in un passo)*.

È chiaro che $\to$ è una relazione (Definizione 4.1.1) su $(T \cup N)^\star$. Indicheremo con $\to^\star$ la sua chiusura riflessiva e transitiva (Definizione 4.1.4), e diremo che β *è derivabile da* α se vale $\alpha \to^\star \beta$. Diremo anche che $\alpha \to^\star \beta$ è una *derivazione*.
Per esempio, con riferimento a $GExp$, si hanno le seguenti derivazioni a un passo e derivazioni:

Exp $\to$ `(`*Exp* + *Exp*`)`
Exp $\to^\star$ `(5 + (`*Exp* $*$ *Exp*`))`
Exp $\to^\star$ `(5 + (4 * 2))`
`((` + *Exp*`)` $\to$ `((` + `(`*Exp* $*$ *Exp*`))`

L'ultimo esempio illustra il fatto che esistono derivazioni a partire da una certa stringa a prescindere dal fatto che da questa si possano derivare stringhe appartenenti al linguaggio generato dalla grammatica, vedi sotto.
Il linguaggio generato da una grammatica è l'insieme delle stringhe composte solo di terminali che si possono derivare dal simbolo iniziale. Formalmente:

Def. 2.2.3 [Linguaggio generato da una grammatica] Sia data una grammatica $G = \langle T, N, P, S\rangle$. Il *linguaggio generato* da G è l'insieme $L(G) = \{u \in T^\star \mid S \to^\star u\}$.

Un linguaggio L è *context-free* (CF) o *libero da contesto* se esiste una grammatica libera da contesto G che lo genera, ossia tale che $L(G) = L$.
Talvolta siamo interessati non solo al linguaggio generato a partire dall'assioma, come nella definizione precedente, ma alla famiglia dei linguaggi, vedi la Definizione 4.3.1, generati a partire da ogni non terminale della grammatica, ossia $L_A(G) = \{u \in T^\star \mid A \to^\star u\}$. In questo caso non è quindi significativo scegliere un assioma.
Concludiamo questa sezione con qualche altro semplice esempio con alfabeto dei terminali $\Sigma = \{a, b\}$. Ricordiamo che una stringa u è *palindroma* se rimane uguale letta a rovescio.

$S ::= a\,S \mid bS \mid \epsilon$	genera $\Sigma^\star$
$S ::= S$	genera $\emptyset$
$S ::= a\,S \mid S\,b \mid \epsilon$	genera $\{a^n b^m \mid n, m \geq 0\}$
$S ::= a\,S\,b \mid \epsilon$	genera $\{a^n b^n \mid n \geq 0\}$,
$S ::= a\,S\,b \mid a\,b$	genera $\{a^n b^n \mid n \geq 1\}$,
$S ::= a\,S\,a \mid b\,S\,b \mid \epsilon \mid a \mid b$	genera $\{u \mid u \in \{a, b\}^\star, u$ palindroma$\}$.

Esercizio 2.2.4 Proviamo che il linguaggio $\{a^m b^n a^{m+n} \mid m, n \geq 0\}$ è context-free. Per dimostrarlo occorre dare una grammatica context-free che lo generi. Dato che il linguaggio può essere equivalentemente descritto come $\{a^m b^n a^n a^m \mid m, n \geq 0\}$, è facile convincersi che una grammatica che lo genera è la seguente:

$$\begin{aligned} S &::= a\,S\,a \mid X \\ X &::= b\,X\,a \mid \epsilon \end{aligned}$$

Vedremo nella prossima sezione come provare in modo rigoroso che una grammatica genera esattamente un linguaggio dato.

2.2.2 Grammatiche come definizioni induttive

Notiamo che una grammatica CF può essere vista come una definizione induttiva (multipla) vedi la Sezione **??**. Ossia, ogni produzione

$$A ::= u_0 B_1 u_1 \ldots B_n u_n,$$

con $n \geq 0$, $A, B_1, \ldots, B_n$ non terminali e $u_1, \ldots, u_n$ stringhe di terminali, può essere letta come una condizione di *chiusura* che la famiglia dei linguaggi generati deve soddisfare, nel modo seguente:

> se la stringa v_1 appartiene al linguaggio di tipo B_1, ..., e la stringa v_n appartiene al linguaggio di tipo B_n, allora la stringa $u_0 v_1 u_1 \ldots v_n u_n$ deve appartenere al linguaggio di tipo A.

Quindi, la famiglia dei linguaggi generati dalla grammatica può essere vista equivalentemente come la più piccola famiglia *chiusa* rispetto a tutte le produzioni viste nel modo sopra descritto. Questa definizione alternativa ci fornisce un modo semplice per provare che la famiglia dei linguaggi generati soddisfa certe proprietà, utilizzando cioè il principio di induzione, vedi Proposizione 4.5.1, nel caso particolare in cui la definizione induttiva sia una grammatica CF. Parleremo in questo caso di *induzione strutturale* o *sulla sintassi*.

Prop. 2.2.5 [Principio di induzione strutturale] Sia $G = \langle T, N, P, S\rangle$ una grammatica CF e P una famiglia (indiciata su N) di predicati su $T^\star$.
Se: per ogni produzione $A ::= u_0 B_1 u_1 \dots B_n u_n$, $\quad (P_{B_i}(u) = \text{T per ogni } u \in L_{B_i}(G), \text{ per ogni } i \in [1,n])$ implica $P_$
allora: $P_B(u) = \text{T}$ per ogni $u \in L_B(G)$, per ogni $B \in T$.

Vediamo qualche esempio.

Esempio 2.2.6 Proviamo che il linguaggio generato da $S ::= aSa \mid bSb \mid \epsilon \mid a \mid b$ è l'insieme delle stringhe palindrome. In questo caso la definizione induttiva non è multipla. Dobbiamo provare le due inclusioni, ossia che la grammatica genera:

- *solo* stringhe palindrome
- *tutte* le stringhe palindrome.

La prima inclusione può essere provata per induzione strutturale, ossia facendo vedere che l'insieme delle stringhe palindrome è *chiuso* rispetto alla grammatica vista come definizione induttiva, ossia rispetto a ogni produzione, quindi controllando le seguenti condizioni:

- se u è palindroma allora anche aua è palindroma
- se u è palindroma allora anche bub è palindroma
- ϵ è palindroma
- a è palindroma
- b è palindroma

che valgono tutte ovviamente. Questa tecnica è applicabile canonicamente per provare che il linguaggio generato da una grammatica gode di una certa proprietà, ossia è contenuto in un linguaggio dato.
Per provare la seconda inclusione, occorre invece un ragionamento *ad hoc* che provi che tutte le stringhe del linguaggio dato sono generabili, tipicamente ancora utilizzando una qualche forma del principio di induzione. In questo esempio, possiamo provare per induzione aritmetica che, per ogni $n \geq 0$, le stringhe palindrome di lunghezza $2n$ e $2n+1$ sono generabili. Infatti:

- per $n = 0$, le stringhe palindrome di lunghezza zero o uno sono generate (applicando una delle ultime tre produzioni)
- assumendo vera la tesi per n, le stringhe palindrome di lunghezza $2(n+1) = 2n+2$ oppure $2(n+1)+1 = 2n+3$ sono della forma aua oppure bub con u stringa palindroma di lunghezza $2n$ oppure $2n+1$, quindi si ha la tesi applicando l'ipotesi induttiva e una delle prime due produzioni.

Per esercizio si riformuli la prova utilizzando invece l'induzione forte, vedi Proposizione 4.5.3.

Esempio 2.2.7 Vediamo ora un esempio di grammatica con più non terminali, che corrisponde quindi a una definizione induttiva multipla. Consideriamo la seguente grammatica:

$$\begin{aligned} S &::= \epsilon \mid aB \mid bA \\ A &::= aS \mid bAA \\ B &::= bS \mid aBB \end{aligned}$$

Questa grammatica è una definizione mutuamente induttiva di tre sottoinsiemi delle stringhe su $\{a, b\}$: l'insieme delle stringhe con un numero uguale di a e di b, l'insieme delle stringhe con una a in più, e l'insieme delle stringhe con una b in più. Proviamolo procedendo come nell'esempio precedente ma per induzione multipla. Per provare che la grammatica genera solo stringhe che soddisfano il vincolo dato, basta controllare le seguenti condizioni, che valgono tutte ovviamente:

- ϵ è una stringa con numero di a uguale al numero di b
- se u è una stringa con una b in più, allora au è una stringa con numero di a uguale al numero di b; analogamente se u è una stringa con una a in più, allora bu è una stringa con numero di a uguale al numero di b
- se u è una stringa con numero di a uguale al numero di b, allora au è una stringa con una a in più, e analogamente bu è una stringa con una b in più;
- se u, v sono due stringhe con una a in più, allora buv è una stringa con una a in più, e analogamente se u, v sono due stringhe con una b in più, allora auv è una stringa con una b in più.

Per provare il viceversa, possiamo provare per induzione forte che per ogni $n \geq 0$ le stringhe di lunghezza n con un numero uguale di a e di b, una a in più e una b in più sono generate a partire da S, A o B, rispettivamente. Infatti:

- Per $n = 0$, la stringa vuota è generata a partire da S applicando la prima produzione.
- Per $n > 0$, distinguiamo i tre casi.
  - Se la stringa ha un numero uguale di a e di b, allora: se inizia per a, sarà della forma au con u stringa con una b in più, e quindi è generata a partire da S applicando l'ipotesi induttiva e la seconda produzione; analogamente se inizia per b.
  - Se la stringa ha una a in più, allora: se inizia per a, sarà della forma au con u stringa con numero uguale di a e di b, e quindi è generata a partire da A applicando l'ipotesi induttiva e la quarta produzione; se invece inizia per b, sarà della forma bu con u con due a in più. Ma una stringa con due a in più può chiaramente essere decomposta come $u_1 u_2$ dove in u_1, u_2 vi è una a in più. Quindi, au_1u_2 è generata applicando l'ipotesi induttiva e la quinta produzione. Si noti che è necessaria l'induzione forte in quanto sappiamo solo che le lunghezze di u_1, u_2 sono minori di quella di au_1u_2.
  - Analogamente se la stringa ha una b in più.

2.2.3 Automi a pila

Il fatto che certi linguaggi, per esempio $\{a^n b^n\}$ o $\{ww^R\}$ (w^R denota la stringa ottenuta "rovesciando" w) non possano essere riconosciuti con un automa a stati finiti dipende intuitivamente dall'impossibilità, in questi automi, di contare o memorizzare simboli in un numero non limitato a priori. Questo diventa invece possibile utilizzando una *pila*.

Def. 2.2.8 [Automa a pila] Un *automa a pila (non deterministico)* (*pushdown automaton* o *PDA*) è una tupla

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

dove Q, Σ, q_0 e F sono come per gli automi a stati finiti, Γ è l'alfabeto della pila, Z è il simbolo iniziale nella pila, e $\delta\colon\ Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \wp_F(Q \times \Gamma^\star)$ è la funzione di transizione[4].

[4]Qui $\wp_F(X)$ denota l'insieme dei sottoinsiemi *finiti* di X. Infatti assumiamo che il numero delle mosse possibili sia sempre finito.

In termini informali, un automa a pila che si trova nello stato q, legge il simbolo σ, e toglie dalla pila il simbolo "in cima" (*pop*) Z, può passare in un nuovo stato q' e mettere in cima alla pila (*push*) una sequenza α anche vuota di nuovi simboli (il caso $\alpha = X$ corrisponde a leggere il simbolo in cima senza toglierlo, il caso $\alpha = \epsilon$ corrisponde a leggere solamente). Inoltre, l'automa può anche passare in un nuovo stato e mettere in cima alla pila una sequenza di simboli senza leggere alcun simbolo, ossia con una transizione silente. Si noti che si tratta di automi *non deterministici*, in quanto per lo stesso stato, simbolo corrente e simbolo in cima alla pila si possono avere transizioni diverse (incluse quelle silenti).
Il sistema di transizione associato a un automa a pila è definito nel modo seguente.

- Le configurazioni, o descrizioni istantanee, sono della forma $\langle q, u, \alpha\rangle$ dove $q \in Q$ è lo stato corrente, $u \in \Sigma^\star$ è la stringa ancora da leggere e $\alpha \in \Gamma^\star$ è il contenuto corrente della pila.

- La relazione di riduzione è definita nel modo seguente:

  $\langle q, \sigma u, X\alpha\rangle \to \langle q', u, \gamma\alpha\rangle$ se $\langle q', \gamma\rangle \in \delta\langle q, \sigma, X\rangle$
  $\langle q, u, X\alpha\rangle \to \langle q', u, \gamma\alpha\rangle$ se $\langle q', \gamma\rangle \in \delta\langle q, \epsilon, X\rangle$

  La relazione di riduzione è non deterministica (vi è anche un non determinismo che consiste in leggere o non leggere) e non terminante. Tuttavia, le computazioni infinite sono dovute solo alla presenza presenza di cicli di transizioni ϵ da uno stato q e simbolo in cima alla pila X nella stessa coppia, che possono essere eliminati.

- Le configurazioni di arresto sono quelle della forma $\langle q, u, \epsilon\rangle$ oppure $\langle q, u, X\alpha\rangle$ con $\delta(q, \sigma, X) = \emptyset$ e $\delta(q, \epsilon, X) = \emptyset$ oppure $\langle q, \epsilon, X\alpha\rangle$ con $\delta(q, \epsilon, X) = \emptyset$.

Le direttive di input sono $f_{\text{IN}}(u) = \langle q_0, u, Z\rangle$. Le direttive di output sono diverse a seconda che si consideri il riconoscimento *per pila vuota* o *per stati finali*. Nel primo caso si ha

$$f_{\text{OUT}}(\langle q, u, \alpha\rangle) = \begin{cases} \text{T} & \text{se } u = \epsilon \text{ e } \alpha = \epsilon \\ \text{F} & \text{altrimenti.} \end{cases}$$

Nel secondo caso si ha

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

Il linguaggio $L(\mathcal{M})$ accettato (riconosciuto) da $\mathcal{M}$ risulta quindi essere, nel caso di riconoscimento per pila vuota:

$$L(\mathcal{M}) = \{u \in \Sigma^\star \mid \langle q_0, u, Z\rangle \to^\star \langle \_, \epsilon, \epsilon\rangle\}$$

Nel caso di riconoscimento per stati finali:

$$L(\mathcal{M}) = \{u \in \Sigma^\star \mid \langle q_0, u, Z\rangle \to^\star \langle q, \epsilon, \_\rangle, q \in F\}$$

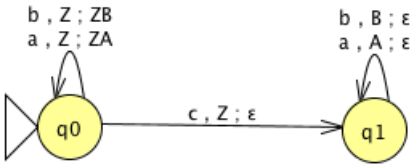
Utilizziamo la wildcard _ per sottolineare il fatto che lo stato non è rilevante nel primo caso, e la pila nel secondo.
Si può provare che le due definizioni sono equivalenti, ossia, se il linguaggio L è riconosciuto da un qualche PDA secondo la prima definizione, allora esiste anche un PDA che lo riconosce secondo l'altra, e viceversa. Quindi nel seguito, senza perdere in generalità, considereremo la prima definizione, e dato che con questa l'insieme degli stati finali non è rilevante possiamo omettere questa componente.

Esempio 2.2.9 Diamo un PDA che riconosce wcw^R. Un modo sintetico per descrivere un PDA è dare una matrice di transizione per ogni stato, come illustrato sotto per l'esempio.

$\to q_0$	ϵ	a	b	c
Z		q_0, ZA	q_0, ZB	q_1, ϵ
A				
B				

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

Le caselle vuote sono un'abbrevazione per l'insieme vuoto (o qualunque altro insieme che conduca sicuramente a una refutazione).
Anche per gli automi a pila è possibile dare una rappresentazione alternativa come grafo di transizione. Nel caso dell'esempio sopra, il grafo è il seguente:



Per provare che un PDA riconosce un certo linguaggio L dobbiamo, come nel caso degli automi a stati finiti, provare che ogni stringa in L viene accettata (ossia, avendo scelto la prima definizione, dopo aver letto la stringa la pila è vuota) e ogni stringa non in L non viene accettata (ossia dopo aver letto la stringa la pila non è vuota). Nell'esempio, possiamo ragionare nel modo seguente:

- Se la stringa è della forma $\sigma_1 \dots \sigma_n \, c \, \sigma_n \dots \sigma_1$, con $\sigma_1, \dots, \sigma_n \in \{a, b\}$, $n \geq 0$, è facile vedere (identificando per comodità a con A e b con B) che si ha

  $\langle q_0, \sigma_1 \dots \sigma_n \, c \, \sigma_n \dots \sigma_1, Z \rangle \to^\star$ [questo passaggio potrebbe essere formalizzato per induzione aritmetica]
  $\langle q_0, c \, \sigma_n \dots \sigma_1, Z\sigma_n \dots \sigma_1 \rangle \to$
  $\langle q_1, \sigma_n \dots \sigma_1, \sigma_n \dots \sigma_1 \rangle \to^\star$ [questo passaggio potrebbe essere formalizzato per induzione aritmetica]
  $\langle q_1, \epsilon, \epsilon \rangle$

- Viceversa, proviamo che ogni stringa accettata è della forma sopra. Anzitutto, se la stringa non contiene c è immediato vedere che non si svuota mai la pila. Quindi la stringa contiene una (prima) c, ossia è della forma $\sigma_1 \dots \sigma_n \, c \, u$, con $\sigma_1, \dots, \sigma_n \in \{a, b\}$, $n \geq 0$, e *l'unica* sequenza di derivazione possibile è quella vista sopra:

  $\langle q_0, \sigma_1 \dots \sigma_n \, c \, u, Z \rangle \to^+$ [questo passaggio potrebbe essere formalizzato per induzione aritmetica]
  $\langle q_1, u, \sigma_n \dots \sigma_1 \rangle$

  A questo punto, è facile vedere (nuovamente questo potrebbe essere formalizzato per induzione aritmetica) che per svuotare la pila u deve necessariamente essere $\sigma_n \dots \sigma_1$, e si ha l'unica sequenza di derivazione vista sopra.

Come evidenziato dalla precedente prova, questo automa è in realtà *deterministico*, nel senso che per ogni configurazione istantanea vi è al più una transizione possibile. Si noti che, per avere un automa deterministico, occorre non solo che per ogni simbolo corrente vi sia al più una transizione possibile, ma anche che non sia contemporaneamente possibile leggere il simbolo corrente o non leggerlo, ossia che le transizioni silenti corrispondano al caso in cui l'input è esaurito. Questo è formalizzato dalla seguente definizione.

Def. 2.2.10 Un *automa a pila deterministico* (*DPDA*) è un automa a pila $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, Z, F\rangle$ tale che, per ogni $\langle q, \sigma, X\rangle \in Q \times \Sigma \times \Gamma$:

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

È facile vedere che la condizione sopra garantisce che la relazione di riduzione sia deterministica.
Contrariamente a quanto visto per gli automi finiti, gli automi a pila non deterministici sono strettamente più potenti di quelli deterministici. Per esempio, l'automa che riconosce wcw^R dato sopra può essere deterministico grazie alla presenza del simbolo c che separa la prima porzione della stringa dalla seconda[5]. In mancanza di tale separatore, intuitivamente un automa a pila "non sa" quando deve iniziare a riconoscere la seconda porzione, e quindi il riconoscimento è possibile solo in modo non deterministico.

Esempio 2.2.11 Diamo un PDA (non deterministico) che riconosce ww^R.

$\rightarrow q_0$	ϵ	a	b
Z	q_1, Z	q_0, AZ	q_0, BZ
A	q_1, A	q_0, AA	q_0, BA
B	q_1, B	q_0, AB	q_0, BB

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

Informalmente, l'automa può restare in q_0 continuando a mettere sulla pila i simboli che via via incontra, lasciando Z in fondo, oppure in qualunque momento può decidere di passare in q_1 lasciando input e stack immutati, assumendo cioè che la parte restante di input sia la stringa rovesciata. Possiamo convincerci della correttezza dell'automa con un ragionamento simile a quello usato per il PDA deterministico: diamo una traccia che può essere dettagliata per esercizio.

- Se la stringa è della forma $\sigma_1 \ldots \sigma_n\sigma_n \ldots \sigma_1$, $n \geq 0$, è facile vedere che viene accettata.
- Viceversa, proviamo che ogni stringa accettata è della forma sopra. Anzitutto, è immediato vedere che per riuscire a vuotare la pila è necessario preliminarmente passare allo stato q_1, ossia effettuare una delle transizioni della prima colonna. Quindi la prima parte della derivazione che accetta la stringa è della forma:

  $$\langle q_0, \sigma_1 \ldots \sigma_n u, Z\rangle \rightarrow^\star \langle q_0, u, \sigma_n \ldots \sigma_1\, Z\rangle \rightarrow \langle q_1, u, \sigma_n \ldots \sigma_1\, Z\rangle$$

  A questo punto, è facile vedere che per svuotare la pila u deve necessariamente essere $\sigma_n \ldots \sigma_1$.

Esercizio 2.2.12 Si dia un PDA per $a^n b^n$.

È possibile provare in modo semplice che un linguaggio non può essere riconosciuto deterministicamente per pila vuota nel caso esistano due stringhe distinte del linguaggio tali che una è un prefisso dell'altra. Per esempio, nel caso del linguaggio ww^R, $abba$ e $abbaabba$ appartengono entrambe al linguaggio. Un PDA deterministico che riconosca il linguaggio per pila vuota deve necessariamente trovarsi in una configurazione in cui la pila è vuota dopo aver letto $abba$, dato che è deterministico questa configurazione è l'unica possibile, quindi non potrà riconoscere $abbaabba$.
È possibile mostrare che i linguaggi generati da grammatiche CF coincidono con quelli riconosciuti da automi a pila (non deterministici). Vediamo una traccia della prova.
Per dimostrare che, dato un automa a pila, è sempre possibile costruire una grammatica CF equivalente, si procede nel modo seguente. Anzitutto, si può provare che (aumentando i simboli nella pila) ci si può sempre ridurre ad automi a pila con un unico stato, ossia che, dato un automa a pila $\mathcal{M}$, esiste un automa $\mathcal{M}'$ con un unico stato tale che $L(\mathcal{M}) = L(\mathcal{M}')$. A questo punto, è sufficiente descrivere la costruzione per il caso di un automa a pila con un unico stato[6], sia $\mathcal{M} = \langle\{q\}, \Sigma, \Gamma, \delta, q, Z\rangle$. La grammatica che genera il

[5] Invece mantenere Z in cima alla pila nella prima fase serve a rendere l'automa, e la prova, più semplici, ma non è necessario. Per esercizio, si dia una variante di automa che rimuove subito Z, e si adatti la prova di conseguenza.

[6] Ricordiamo che, avendo scelto il riconoscimento per pila vuota, si può omettere l'insieme degli stati finali.

linguaggio riconosciuto da $\mathcal{M}$ è definita nel modo seguente:

- l'alfabeto dei terminali è Σ
- l'alfabeto dei non terminali è Γ
- il simbolo iniziale è Z
- vi è una produzione $X ::= \sigma\gamma$ se e solo se $\langle q, \gamma\rangle \in \delta\langle q, \sigma, X\rangle$
- vi è una produzione $X ::= \gamma$ se e solo se $\langle q, \gamma\rangle \in \delta\langle q, \epsilon, X\rangle$

Si può dimostrare per induzione (farlo per esercizio) che per ogni stringa $u \in \Sigma^\star$:

$\langle q, u, Z\rangle \to^\star \langle q, \epsilon, \epsilon\rangle$ se e solo se $Z \to^\star u$

da cui segue la tesi.
La costruzione inversa, da una grammatica CF a un automa a pila equivalente, procede analogamente. Anzitutto, si può provare che[7] ci si può sempre ridurre a grammatiche in *forma normale di Greibach*, ossia dove le produzioni sono tutte della forma $A ::= \sigma B_1 \dots B_n$ con σ terminale, $B_1, \dots, B_n$ non terminali, $n \geq 0$. A questo punto, è sufficiente descrivere la costruzione per il caso di una grammatica di Greibach, sia $G = \langle T, N, P, S\rangle$. L'automa che riconosce il linguaggio generato dalla grammatica è definito nel modo seguente:

- l'insieme degli stati è $\{q\}$
- l'alfabeto è T
- l'alfabeto della pila è N
- il simbolo iniziale nella pila è S
- $\langle q, \gamma\rangle \in \delta\langle q, \sigma, X\rangle$ se e solo se $X ::= \sigma\gamma$ è una produzione

Si può dimostrare per induzione (farlo per esercizio) che per ogni stringa $u \in \Sigma^\star$:

$S \to^\star u$ se e solo se $\langle q, u, S\rangle \to^\star \langle q, \epsilon, \epsilon\rangle$

da cui segue la tesi.

Esercizio 2.2.13 Si consideri la seguente grammatica (in forma normale di Greibach):
$S ::= aB \mid bA$
$A ::= aS \mid bAA \mid a$
$B ::= bS \mid aBB \mid b$
che genera il linguaggio formato dalle stringhe con lo stesso numero (maggiore di zero) di a e di b. Si costruisca il PDA equivalente.

Si noti che il comportamento del PDA riconoscitore corrisponde esattamente a quello di un parser ricorsivo discendente (non deterministico).

[7] Assumendo per semplicità che il linguaggio non contenga la stringa vuota. In caso contrario la costruzione risulta leggermente più complicata.

2.2.4 Proprietà dei linguaggi context-free

Anche per i linguaggi CF vale un pumping lemma che può essere utilizzato per provare che un linguaggio non è CF. L'idea intuitiva è la seguente: in ogni stringa sufficientemente lunga di un linguaggio CF si possono trovare due sottostringhe che è possibile eliminare o ripetere un numero arbitrario di volte, ottenendo sempre stringhe del linguaggio.

Lemma 2.2.14 [Pumping lemma per i linguaggi CF (Bar-Hillel, Perles, Shamir, 1961)] Sia L un linguaggio CF. Allora esiste una costante $n \in \mathbb{N}$ tale che, per ogni $z \in L$ con $|z| \geq n$, possiamo decomporre z come $z = uvwxy$ in modo che:

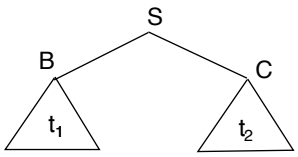
1. $|vwx| \leq n$,
2. $|vx| > 0$,
3. per ogni $i \geq 0$ si ha che $uv^i wx^i y \in L$.

Prova Se L è context-free, esiste una grammatica CF che lo genera. Si può provare (non lo vediamo) che, in particolare, esiste[8] una grammatica che lo genera *in forma normale di Chomsky*, ossia dove tutte le produzioni sono della forma $A ::= BC$, con B e C non terminali, oppure $A ::= \sigma$ con σ terminale.
È facile vedere che, per una grammatica in forma normale di Chomsky, se un albero di derivazione ha altezza m, la sua frontiera (quindi la stringa generata) è lunga al più 2^{m-1}. Infatti:

Base Un albero di derivazione di altezza 1 ha necessariamente forma

quindi genera una stringa di lunghezza 2^{1-1}.

Passo induttivo Un albero di derivazione di altezza $m+1$ ha necessariamente forma

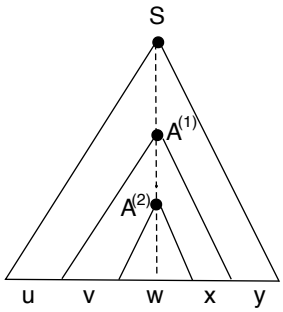


con t_1, t_2 alberi di altezza al più m, le cui frontiere quindi per ipotesi induttiva sono lunghe al più 2^{m-1}. Dato che la frontiera dell'albero completo è la concatenazione delle due, sarà lunga al più $2 \cdot 2^{m-1} = 2^m$.

Sia allora m il numero di non terminali della grammatica in forma normale di Chomsky che genera L, e prendiamo $n = 2^m$. Data una stringa $z \in L$ con $|z| \geq n$, per il risultato provato sopra l'altezza di un albero di derivazione per z deve essere almeno $m+1$. Allora, deve esserci necessariamente un non terminale, sia A, che compare due volte in un cammino dalla radice alle foglie. Indichiamo con $A^{(1)}$ la prima occorrenza e con $A^{(2)}$ la seconda. Allora z può essere decomposta come $uvwxy$ dove w è la stringa ottenuta espandendo l'occorrenza $A^{(2)}$ e vwx è la stringa ottenuta espandendo l'occorrenza $A^{(1)}$. Ma dato che si tratta di due occorrenze dello stesso non terminale A, è chiaro che se sostituiamo il sottoalbero con radice $A^{(1)}$ con

[8] Sempre assumendo per semplicità $\epsilon \notin L$, caso comunque non rilevante qui.

quello con radice $A^{(2)}$ otteniamo ancora un albero di derivazione valido, per la stringa uwy. Viceversa, sostituendo il sottoalbero con radice $A^{(2)}$ con quello con radice $A^{(1)}$, otteniamo un albero di derivazione per la stringa uv^2wx^2y, e iterando la sostituzione otteniamo alberi di derivazione per tutte le stringhe uv^iwx^iy con $i > 2$.
La situazione è schematizzata nella seguente figura:



Dobbiamo ancora controllare che:

- $|vwx| \leq n$; per ottenere questo, basta scegliere le due occorrenze dello stesso non terminale in modo che $A^{(1)}$ sia la prima ripetizione a partire dal basso, ossia che non ci siano altre ripetizioni di non terminali nel sottoalbero di radice $A^{(1)}$. Allora l'altezza di questo albero è al più $m + 1$, quindi la lunghezza di vwx è al più $n = 2^m$.

- $|vx| > 0$; questo si vede facilmente in quanto i figli di $A^{(1)}$ sono necessariamente (etichettati con) due non terminali, quindi v e x non possono essere entrambe vuote. □

Come nel caso dei regolari, il pumping lemma viene utilizzato per provare che un linguaggio L *non* è CF. A tale scopo, occorre mostrare che, fissato un n arbitrario, esiste una stringa z, con $|z| \geq n$, che appartiene al linguaggio, ma tale che, per ogni sua possibile decomposizione come $uvwxy$, con $|vwx| \leq n$ e $|vx| > 0$, $uv^iwx^iy \notin L$ per qualche $i \geq 0$. Il seguente esempio illustra la tecnica.

Esempio 2.2.15 Possiamo dimostrare, utilizzando il pumping lemma, che il linguaggio $\{a^nb^nc^n\}$ non è CF. Infatti, preso n arbitrario, consideriamo la stringa $a^nb^nc^n$. Vi sono molti diversi casi di decomposizione di questa stringa come $uvwxy$. Tuttavia, dato che la lunghezza di vwx deve essere $\leq n$, è facile capire che tale sottostringa *non* può contenere sia a che c, perché in tal caso dovrebbe contenere tutti i b. Quindi, prendendo uv^0wx^0y, si ottiene una stringa in cui il numero di uno o due simboli è diminuito strettamente, quindi la stringa non appartiene al linguaggio.

In modo analogo possiamo provare che non sono CF i linguaggi $\{a^nb^mc^nd^m\}$ (dato n basta considerare $a^nb^nc^nd^n$) e $\{ww\}$ (dato n basta considerare $a^nb^na^nb^n$). Questi due esempi possono essere considerati come la "versione astratta" di situazioni che si verificano usualmente nei linguaggi di programmazione: nel primo caso, due diverse chiamate di funzione devono rispettare ognuna il numero di parametri della corrispondente dichiarazione; nel secondo caso, un identificatore può essere utilizzato solo se è stato precedentemente dichiarato. Questi esempi forniscono quindi una motivazione del fatto che i linguaggi di programmazione non sono CF, ma sono descritti, oltre che attraverso una grammatica CF che genera un soprainsieme del linguaggio, anche specificando dei *vincoli contestuali*.
Per quanto riguarda le proprietà di chiusura, i linguaggi CF risultano chiusi rispetto all'unione, alla concatenazione e alla chiusura di Kleene (questo si può vedere molto semplicemente costruendo le rispettive grammatiche a partire da quelle date). Invece, i linguaggi CF non sono chiusi rispetto all'intersezione: questo può essere visto osservando che i linguaggi $a^nb^nc^m$ e $a^nb^mc^m$ sono CF (è banale dare le grammatiche

che li generano), mentre la loro intersezione $a^n b^n c^n$ non lo è, come osservato sopra. Di conseguenza i linguaggi CF non sono neanche chiusi rispetto alla complementazione.
Per i linguaggi CF il problema dell'appartenenza risulta decidibile. Infatti, dato un linguaggio CF, esiste[9] una grammatica in forma normale di Greibach che lo genera. Allora, data una stringa u, se questa è generata lo è in esattamente $|u|$ passi, quindi per vedere se appartiene al linguaggio basta generare esaustivamente tutte le derivazioni di $|u|$ passi. Questo algoritmo ha complessità esponenziale, esistono algoritmi più efficienti di complessità $O(|u|^3)$ (o anche $O(|u|^{2.8})$). Tali algoritmi hanno comunque un valore puramente teorico, in quanto nella pratica il parsing è effettuato per particolari classi di grammatiche CF per cui è fattibile in modo più efficiente. Si osservi anche che nel caso dei linguaggi regolari il problema è decidibile in tempo lineare.

[9] Assumendo per semplicità $\epsilon \notin L$.

Capitolo 3

Teoria della calcolabilità

Ci proponiamo di rispondere ad alcuni quesiti fondamentali per l'informatica:

- cosa intendiamo per *funzione calcolabile* con un programma?
- definito in qualche modo cosa intendiamo per funzione calcolabile, un particolare linguaggio di programmazione o altro formalismo è atto a calcolare una qualunque funzione calcolabile?
- ci sono funzioni *non* calcolabili con un programma?

Tentando di dare una risposta, si è condotti a proporre diverse definizioni equivalenti di funzione calcolabile (tesi di Church).
Iniziamo spiegando come si arriva in modo naturale ad ammettere tra le funzioni calcolabili anche *funzioni parziali*, ossia non necessariamente definite su tutti gli argomenti, vedi Definizione 4.1.1.

3.1 Nozione di algoritmo e funzioni ricorsive primitive

Alcuni requisiti che sembrano inerenti alla nozione intuitiva di algoritmo sono i seguenti:

- lista di istruzioni
- direttive di esecuzione
- direttive di input/output

Per esempio, troviamo queste caratteristiche negli automi studiati precedentemente per i linguaggi regolari e context-free, che peraltro sappiamo già non essere un formalismo sufficientemente espressivo.
Diamo ora un esempio di formalismo che sembra avere i requisiti richiesti. Consideriamo una classe di funzioni, dette *ricorsive primitive* ($\mathcal{PR}$) da $\mathbb{N}^k$ in $\mathbb{N}$, $k \geq 0$, definite induttivamente nel modo seguente:

- ogni funzione[1] *zero*, che restituisce sempre 0, è ricorsiva primitiva: $Z(x_1, \ldots, x_n) = 0$, con $n \geq 0$
- la funzione *successore* è ricorsiva primitiva: $S(x) = x + 1$
- ogni funzione *proiezione* è ricorsiva primitiva: $\Pi_i^n(x_1, \ldots, x_n) = x_i$, con $1 \leq i \leq n$

[1] In realtà è sufficiente avere la funzione zero senza parametri, le altre possono essere costruite a partire da questa: lo si mostri per esercizio.

- se $h\colon \mathbb{N}^k \to \mathbb{N}$ e $g_i\colon \mathbb{N}^n \to \mathbb{N}$ per ogni $i \in [1,k]$ sono ricorsive primitive, allora lo è la funzione $f\colon \mathbb{N}^n \to \mathbb{N}$ ottenuta per *composizione*, definita da:

$$f(x_1,\ldots,x_n) = h(g_1(x_1,\ldots x_n),\ldots,g_k(x_1,\ldots,x_n))$$

- se $g\colon \mathbb{N}^{n-1} \to \mathbb{N}$ e $h\colon \mathbb{N}^{n+1} \to \mathbb{N}$ sono ricorsive primitive, allora lo è la funzione $f\colon \mathbb{N}^n \to \mathbb{N}$ ottenuta per *ricorsione primitiva*, definita da:

$$f(x_1,\ldots,x_{n-1},0) = g(x_1,\ldots,x_{n-1})$$
$$f(x_1,\ldots,x_{n-1},y+1) = h(x_1,\ldots,x_{n-1},y,f(x_1,\ldots,x_{n-1},y))$$

Ossia, una funzione è ricorsiva primitiva se può essere costruita, a partire dalle funzioni banalmente calcolabili di base (funzioni costanti zero, successore e proiezioni), utilizzando la composizione (generalizzata), e un meccanismo di ricorsione molto semplice nel quale il caso base è zero e la chiamata ricorsiva è sempre effettuata sul predecessore, in modo che la ricorsione sia sempre terminante.
Possiamo introdurre un semplice linguaggio di programmazione per scrivere funzioni ricorsive primitive nel modo seguente. Assumiamo di avere un insieme di *nomi di funzione f*, ognuna con una certa *arità* (numero di parametri), e un insieme di *variabili (nomi di parametro)* x, y, z. Un *programma* è una sequenza di *dichiarazioni di funzione*, che possono avere due formati, assumendo *f* di arità n:

$$f(x_1,\ldots,x_n) = e$$

oppure, se $n > 0$:

$$f(x_1,\ldots,x_{n-1},Z) = e$$
$$f(x_1,\ldots,x_{n-1},S(y)) = e'$$

Nel secondo caso si dice che *f* è definita *per ricorsione primitiva*.
Le espressioni *e* sono definite nel modo seguente:

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

Inoltre, valgono i seguenti vincoli contestuali: un nome di funzione non può essere dichiarato più volte, e una variabile non può apparire più di una volta come parametro di una funzione; ogni dichiarazione di funzione può contenere chiamate solo di funzioni dichiarate precedentemente; nel caso di una funzione *f* definita per ricorsione primitiva, l'espressione e' può contenere anche chiamate ricorsive a *f*, ma solo della forma $f(x_1,\ldots,x_{n-1},y)$. Infine, in ogni chiamata di funzione il numero degli argomenti corrisponde all'arità.
L'ultima funzione dichiarata in un programma è la *funzione principale*.
Possiamo definire il sistema di transizione associato nel modo seguente.

- Le configurazioni sono espressioni senza variabili.

- Le direttive di esecuzione consistono nel procedere per sostituzioni successive fino ad arrivare al risultato finale. Formalmente, la relazione di riduzione è definita dalle seguenti regole, dove $e[e_1/x_1 \ldots e_n/x_n]$ indica l'espressione ottenuta sostituendo tutte le occorrenze di x_i con e_i, per ogni $i \in 1..n$

  - Se c'è una dichiarazione $f(x_1,\ldots,x_n) = e$ allora

    $$f(e_1,\ldots,e_n) \to e[e_1/x_1 \ldots e_n/x_n].$$

  - Se c'è una dichiarazione

$f(x_1, \ldots, x_{n-1}, Z) = e$
$f(x_1, \ldots, x_{n-1}, S(y)) = e'$

allora

$f(e_1, \ldots, e_{n-1}, Z) \to e[e_1/x_1 \ldots e_{n-1}/x_{n-1}]$
$f(e_1, \ldots, e_{n-1}, S(e_n)) \to e'[e_1/x_1 \ldots e_{n-1}/x_{n-1}][e_n/y]$

– Infine, se $e \to e'$, allora $S(e) \to S(e')$, e se $e_i \to e_i'$, allora $f(\ldots, e_i, \ldots) \to f(\ldots, e_i', \ldots)$.

- Le configurazioni di arresto risultano essere quelle della forma $S^x(Z)$, ossia rappresentazioni di numeri naturali, che indicheremo con $\bar{x}$ (quindi $\bar{0} = Z$).

Dato un programma con funzione principale f di arità n, le direttive di input/output sono

$f_{\text{IN}}(x_1, \ldots, x_n) = f(\bar{x}_1, \ldots, \bar{x}_n)$
$f_{\text{OUT}}(\bar{x}) = x.$

Vediamo, per esempio, come definire la somma di due numeri naturali.

$sum(x, Z) = x$
$sum(x, S(y)) = S(sum(x, y))$

Vediamo un esempio di calcolo del risultato di una funzione ricorsiva primitiva.

$$sum(\bar{2}, \bar{2}) \to S(sum(\bar{2}, \bar{1})) \to S(S(sum(\bar{2}, Z))) \to S(S(\bar{2})) \equiv \bar{4}$$

Osserviamo che le direttive di esecuzione sono non deterministiche (per esempio, in $sum(sum(\bar{2}, \bar{2}), sum(\bar{1}, \bar{1}))$ ho due scelte possibili), ma si intuisce, e si può provare, che il risultato sarà sempre lo stesso. Si possono comunque scegliere direttive di esecuzione deterministiche scegliendo in modo "canonico" la chiamata da espandere (per esempio, la più interna più a sinistra). Per formalismi più potenti di $\mathcal{PR}$ invece l'ordine di valutazione può influire sul risultato. In particolare, per $\mathcal{PR}$ l'unicità del risultato deriva dal fatto che, come è facile provare per induzione sulla definizione (utilizzando l'induzione aritmetica per le funzioni definite per ricorsione primitiva), tutte le funzioni in $\mathcal{PR}$ sono *totali*, ossia definite su ogni possibile argomento.
Moltissime funzioni usuali in matematica e informatica sono ricorsive primitive, come illustrato dai seguenti esempi dove, per leggibilità, ammettiamo anche la sintassi infissa (quindi scriviamo $x + y$ invece di $sum(x, y)$), identifichiamo i numeri naturali con la loro rappresentazione, e scriviamo $y + 1$ per $S(y)$: prodotto, potenza, predecessore (assumendo che il predecessore di 0 sia 0), differenza (assumendo che se negativa sia 0), valore assoluto della differenza, minimo tra due elementi, sg che restituisce 0 su 0 e 1 sui positivi, $\overline{sg}$ che restituisce 1 su 0 e 0 sui positivi.

$$x \cdot 0 = 0 \qquad x^0 = S(0)$$
$$x \cdot (y+1) = (x \cdot y) + x \qquad x^{y+1} = x^y \cdot x$$

$$pred(0) = 0 \qquad x \dot{-} 0 = x \qquad |x - y| = (x \dot{-} y) + (y \dot{-} x)$$
$$pred(y+1) = y \qquad x \dot{-} (y+1) = pred(x \dot{-} y)$$

$$\min(x, y) = x \dot{-} (x \dot{-} y) \qquad sg(0) = 0 \qquad \overline{sg}(0) = S(0)$$
$$sg(y+1) = S(0) \qquad \overline{sg}(y+1) = 0$$
$$\text{oppure: } \overline{sg}(x) = 1 \dot{-} sg(x)$$

Le funzioni sg e $\overline{sg}$ sono esempi di *predicati primitivi ricorsivi*, ossia funzioni primitive ricorsive che restituiscono 0 o 1 a seconda che una certa proprietà degli argomenti sia falsa o vera. Infatti $sg(x)$ può essere letto come "x è diverso da 0" e $\overline{sg}(x)$ come "x è 0".
Altri esempi di predicati primitivi ricorsivi sono i seguenti: test di diseguaglianza, test di minore stretto.

$$x \neq y = sg(|x - y|)$$
$$x < y = sg(y \dot{-} x)$$

Possiamo anche codificare meccanismi di composizione tipici dei linguaggi di programmazione all'interno del formalismo. Per esempio, date due funzioni f, g primitive ricorsive e un predicato p primitivo ricorsivo (tutti con un solo argomento per semplicità), la funzione $IF(p, f, g)$ risulta essere primitiva ricorsiva, in quanto può essere definita nel modo seguente:

$$IF(p, f, g)(x) = p(x) \cdot f(x) + \overline{sg}(p(x)) \cdot g(x)$$

Si noti che la ricorsione primitiva è uno schema ricorsivo molto semplice: si procede per induzione aritmetica su *uno* degli argomenti di una funzione. Per esempio, la seguente definizione ricorsiva del test di diseguaglianza:

$$0 \neq 0 = 0$$
$$(x + 1) \neq 0 = 1$$
$$0 \neq (y + 1) = 1$$
$$(x + 1) \neq (y + 1) = x \neq y$$

è corretta, ma non è espressa utilizzando la ricorsione primitiva.
Gli esempi precedenti porterebbero a pensare che la nozione di funzione ricorsiva primitiva catturi il concetto intuitivo di funzione calcolabile mediante un algoritmo. Vedremo nel seguito che questo è falso, in particolare mostrando che la classe delle funzioni calcolabili deve necessariamente contenere funzioni *parziali*.
Si noti anche che le seguenti due affermazioni non coincidono:

- f è ricorsiva primitiva
- conosciamo un programma che calcola f

Consideriamo per esempio le seguenti funzioni:

$$g(x) = \begin{cases} 1 & \text{se ogni numero pari maggiore di due è somma di due primi (congettura di Goldbach)} \\ 0 & \text{altrimenti} \end{cases}$$
$$h(x) = \begin{cases} 1 & \text{se ci sono almeno } x \text{ “5” consecutivi nello sviluppo decimale di } \pi \\ 0 & \text{altrimenti} \end{cases}$$

Le due funzioni sono ricorsive primitive: infatti, la prima è la funzione costante 1 oppure la funzione costante 0, e la seconda è una funzione che vale 1 fino a un certo x (il numero massimo di “5” consecutivi nello sviluppo decimale di π) e poi 0, oppure la funzione costante 1 se tale massimo non esiste. Tuttavia, non sappiamo quale sia un programma che le calcoli, in quanto non sappiamo se la congettura di Goldbach è vera, e non conosciamo il numero massimo di “5” consecutivi nello sviluppo decimale di π.
Si osservi anche la differenza tra h ed f definita da:

$$f(x) = \begin{cases} 1 & \text{se ci sono } \textit{esattamente } x \text{ “5” consecutivi nello sviluppo decimale di } \pi \\ 0 & \text{altrimenti} \end{cases}$$

In questo caso non possiamo concludere che la funzione sia ricorsiva primitiva, nè se sia calcolabile con un algoritmo.

Benché non si conosca un modo di calcolare g, vi è un ovvio procedimento, che siamo tentati di definire algoritmo, che consiste nel controllare successivamente, sui numeri pari maggiori di due, se questi sono o no somma di due primi. Tuttavia si ottiene $g(x)$ soltanto se la congettura è falsa, mentre se la congettura è vera il procedimento non termina, quindi *non* si ottiene $g(x) = 1$. Quindi, la funzione calcolata, nel caso la congettura sia vera, è la funzione totalmente indefinita, mentre è la funzione costante 0 se la congettura è falsa. In ogni caso sembra difficile negare la qualifica di algoritmo al procedimento descritto. Analogamente per h. Questi esempi mostrano quindi come siamo naturalmente condotti a considerare funzioni non sempre definite come funzioni calcolate da un algoritmo.
Diamo ora una motivazione più fondamentale all'introduzione delle funzioni parziali.

Prop. 3.1.1 I programmi che calcolano funzioni ricorsive primitive possono essere numerati *effettivamente* (ossia, mediante un algoritmo).

Prova Un esempio di algoritmo è la numerazione per elencazione. Infatti, assumendo una sintassi per i nomi di funzione e le variabili, un programma è una stringa su un alfabeto (insieme finito di simboli). Possiamo ordinare tali stringhe in qualche modo (per esempio, a seconda della lunghezza, e quelle della stessa lunghezza secondo l'ordinamento lessicografico). A questo punto, possiamo elencare (ossia, generare una dopo l'altra) le stringhe che sono programmi ben formati. Resta così stabilita una corrispondenza biunivoca tra programmi e numeri naturali, algoritmica nei due sensi: dato un programma possiamo calcolare il suo numero d'ordine, e viceversa preso un numero n possiamo costruire tutti i programmi fino all'n-simo.

Teorema 3.1.2 Esistono funzioni calcolabili con un algoritmo che non sono ricorsive primitive.

Prova Come sottocaso della numerazione precedente, consideriamo una numerazione algoritmica delle funzioni ricorsive primitive in una variabile, sia $\{f_x \mid x \geq 0\}$. Sia f definita da $f(x) = f_x(x) + 1$. Ovviamente f è calcolabile con un algoritmo, poiché la numerazione è algoritmica e ogni f_x è a sua volta calcolabile con un algoritmo. Tuttavia, $f \notin \mathcal{PR}$, infatti se fosse $f \in \mathcal{PR}$ allora si avrebbe $f = f_{\bar{x}}$ per almeno un indice $\bar{x}$, quindi si avrebbe $f(\bar{x}) = f_{\bar{x}}(\bar{x}) = f_{\bar{x}}(\bar{x}) + 1$. □
Questo teorema, oltre a dirci che $\mathcal{PR}$ non va bene come classe candidata a essere "la classe di tutte le funzioni calcolabili con un algoritmo", è particolarmente interessante perchè la dimostrazione usata, detta per *diagonalizzazione*[2], si applica a ogni classe di funzioni che goda dei seguenti due requisiti:

1. esiste una numerazione algoritmica degli algoritmi che definiscono le funzioni della classe

2. le funzioni sono totali.

In altre parole, per ogni classe di funzioni calcolabili con un algoritmo che soddisfi 1) e 2), si può costruire, per diagonalizzazione, una funzione calcolabile con un algoritmo che non appartiene alla classe.
Il nome "diagonalizzazione" deriva dal fatto che, pensando di avere una matrice un cui le righe sono i numeri naturali usati come indici e le colonne sono i numeri naturali usati come argomenti, la funzione f è costruita in modo da essere diversa, per ogni argomento x, dal valore $f_x(x)$ che appare sulla diagonale. Quindi, non può corrispondere a nessuna riga.
Il fatto che $\mathcal{PR}$ non contenga tutte le funzioni calcolabili con un algoritmo si può anche provare direttamente fornendo un controesempio, per esempio dimostrando che non è ricorsiva primitiva la seguente funzione (totale) *di Ackermann*:

```
A(m,n) =
  if m=0 then n+1
  else if n = 0 then A(m-1,1) else A(m-1,A(m,n-1))
```

[2]Usata per esempio per distinguere la cardinalità dei numeri naturali e reali.

Questa funzione è ovviamente calcolabile con un algoritmo (in qualunque linguaggio di programmazione possiamo scrivere in qualche modo la definizione ricorsiva data sopra). Si prova che non è ricorsiva primitiva mostrando che cresce più rapidamente di qualunque funzione ricorsiva primitiva.
Il requisito 1) sembra inerente al concetto di "formalismo per definire tutti gli algoritmi", oltre a essere presente in tutte le formalizzazioni tentate. Invece, tralasciando 2), ossia considerando, tra le funzioni calcolabili, anche quelle parziali, si evita lo scoglio della diagonalizzazione. Infatti, dall'equaglianza $f_{\bar{x}}(\bar{x}) = f_{\bar{x}}(\bar{x}) + 1$ si dedurrà semplicemente che $f_{\bar{x}}$ è una funzione della classe che non è definita su $\bar{x}$ e la diagonalizzazione non conduce quindi a priori all'assurdo. Resta aperto il problema di trovare un formalismo adatto a esprimere algoritmi per tutte le funzioni calcolabili (anche parziali).
Introduciamo nel seguito il formalismo delle macchine di Turing che chiarirà un fatto fondamentale: funzione non definita in un punto x significa essenzialmente che l'algoritmo associato alla funzione porta, per l'input x, a un *calcolo che non termina*.
Per brevità, scriveremo a volte $f(x)\downarrow$ per indicare che la funzione f è definita su x, $f(x)\uparrow$ per indicare che non è definita su x (vedi Definizione 4.1.1).

3.2 Macchine di Turing e funzioni ricorsive

3.2.1 Macchine di Turing

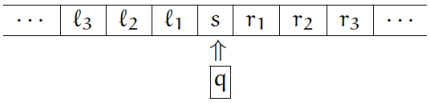
Def. 3.2.1 [Macchina di Turing deterministica] Una *macchina di Turing* (TM) è una tupla $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, B, F\rangle$ dove:

- Q è un insieme finito di stati
- Σ è un alfabeto (alfabeto di input)
- Γ è un alfabeto (alfabeto del nastro), con $\Sigma \subseteq \Gamma$
- $\delta\colon\ Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ è una funzione *parziale* detta *funzione di transizione*
- $q_0 \in Q$ è lo *stato iniziale*
- B è un simbolo speciale in $\Gamma \backslash \Sigma$
- $F \subseteq Q$ è l'insieme degli stati *finali*.

Vi sono molte varianti di questa definizione. In particolare, la scelta di avere *due* alfabeti (di input e del nastro) e un insieme di stati finali è legata alla visione di una macchina di Turing come *riconoscitore*, analogamente agli automi visti precedentemente.
Similmente a quanto visto per gli automi, una macchina di Turing può essere immaginata come:

- un nastro, illimitato nei due sensi e suddiviso in celle
- una testina (di lettura e scrittura) posizionata su una cella del nastro

come illustrato in figura.



Il funzionamento è il seguente. In ogni istante:

- sul nastro vi è un numero finito di simboli in $\Gamma \backslash \{B\}$, ognuno in una cella, mentre B rappresenta la cella vuota (blank)
- la testina si trova in uno stato in Q
- in dipendenza (deterministica) dallo stato e dal simbolo in Γ contenuto nella cella su cui è posizionata la testina, si effettua un'azione multipla, consistente in:
  - cambiamento di stato
  - scrittura di un simbolo di Γ (scrittura di B equivale a cancellazione) sulla cella corrispondente alla testina
  - spostamento della testina a sinistra (L) o a destra (R), talvolta si aggiunge anche nessuno spostamento (N).

  Per qualche stato e simbolo può non essere specificata alcuna azione: si dice in tal caso che la macchina *si ferma*.

Formalmente, la funzione di transizione definisce il programma eseguito dalla macchina, in particolare determina la prossima azione sulla base dello stato e del simbolo corrente. Se $\delta(q, X) = \langle q', Y, D \rangle$, allora quando la testina è nello stato q e si trova su una cella che contiene X:

- q' è il prossimo stato della macchina
- Y è il simbolo in Γ che sovrascrive X nella cella corrente
- D è la direzione, L (left) o R (right), nella quale si muove la testina.

Essendo Q e Γ finiti, analogamente a quanto visto per gli automi la funzione di transizione può essere rappresentata da una matrice di transizione (tabella), o equivalentemente da una lista di quintuple $\langle q, X, q', Y, \boldsymbol{D} \rangle$. Tale tabella corrisponde alla lista di istruzioni richiesta dalla nozione intuitiva di algoritmo, mentre il funzionamento della macchina sopra descritto corrisponde alle direttive di esecuzione.
Formalmente, data $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, B, F \rangle$, possiamo definire il sistema di transizione associato nel modo seguente.

- Le configurazioni (descrizioni istantanee) sono della forma $\langle \alpha, q, \beta \rangle$ dove (come illustrato dalla figura sopra):
  - $\alpha \in \Gamma^\star$ rappresenta i simboli a sinistra della testina
  - $\beta \in \Gamma^\star$ rappresenta i simboli a destra, quindi, se $\beta \neq \epsilon$, il simbolo puntato dalla testina è il primo simbolo di β, altrimenti la testina punta a una cella che contiene B
  - q è lo stato della testina.

  Infatti, le informazioni rilevanti per descrivere la configurazione di una macchina sono lo stato della testina, il contenuto del nastro e la posizione della testina. Più precisamente, è sufficiente rappresentare la porzione di nastro *significativa*, ossia ottenuta escludendo le due sequenze illimitate di celle vuote sempre presenti a sinistra e a destra.[3]

[3] Notiamo come il fatto che la porzione significativa sia sempre finita segue dal fatto che modelliamo gli stati raggiungibili dopo un numero finito di passi elementari.

- La relazione di riduzione è definita nel modo seguente, per $\alpha, \beta \in \Gamma^\star$, X, Y, $Z \in \Gamma$:

$$\begin{array}{ll}
\langle \alpha, q, X\beta\rangle \to \langle \alpha Y, q', \beta\rangle & \text{se } \delta(q, X) = \langle q', Y, \mathrm{R}\rangle \\
\langle \alpha, q, \epsilon\rangle \to \langle \alpha Y, q', \epsilon\rangle & \text{se } \delta(q, B) = \langle q', Y, \mathrm{R}\rangle \\
\langle \alpha Z, q, X\beta\rangle \to \langle \alpha, q', ZY\beta\rangle & \text{se } \delta(q, X) = \langle q', Y, \mathrm{L}\rangle \\
\langle \epsilon, q, X\beta\rangle \to \langle \epsilon, q', BY\beta\rangle & \text{se } \delta(q, X) = \langle q', Y, \mathrm{L}\rangle \\
\langle \alpha Z, q, \epsilon\rangle \to \langle \alpha, q', ZY\rangle & \text{se } \delta(q, B) = \langle q', Y, \mathrm{L}\rangle \\
\langle \epsilon, q, \epsilon\rangle \to \langle \epsilon, q', BY\rangle & \text{se } \delta(q, B) = \langle q', Y, \mathrm{L}\rangle
\end{array}$$

  La relazione di riduzione è deterministica, e in generale non terminante (possono esservi computazioni infinite).

- Le configurazioni di arresto sono quelle della forma $\langle \alpha, q, X\beta\rangle$ tali che $\delta(q, X)\uparrow$, oppure $\langle \alpha, q, \epsilon\rangle$ tali che $\delta(q, B)\uparrow$.

Le direttive di input/output sono $f_{\text{IN}}(u) = \langle \epsilon, q_0, u\rangle$, e $f_{\text{OUT}}(\langle \alpha, q, \beta\rangle) = \begin{cases} \mathrm{T} & \text{se } q \in F, \\ \mathrm{F} & \text{altrimenti.} \end{cases}$

Il linguaggio $L(\mathcal{M})$ accettato da $\mathcal{M}$ risulta quindi essere:

$$L(\mathcal{M}) = \{u \in \Sigma^\star \mid \langle \epsilon, q_0, u\rangle \to^\star \langle \alpha, q, \beta\rangle, q \in F\}$$

Si noti che il contenuto finale del nastro è irrilevante ai fini dell'accettazione. Notiamo inoltre che il linguaggio accettato è un linguaggio sull'alfabeto Σ. Infatti, come l'alfabeto della pila per i PDA, i simboli aggiuntivi sono utilizzati solo ai fini della computazione.
La macchina di Turing è un riconoscitore molto più potente di quelli a stati finiti e a pila visti precedentemente. Infatti nei DFA e PDA (deterministici):

- Il nastro è formato solo dalla sequenza di simboli in input, ed è esaminato una volta sola da sinistra a destra.

- La testina può solo *leggere*.

- La memoria a disposizione è limitata a priori (gli stati della testina nel DFA) oppure non limitata a priori ma accessibile in modo ristretto (lo stack nel PDA).

Nelle TM il nastro è infinito nelle due direzioni (più precisamente, estendibile a piacimento). La testina può scorrere il nastro nelle due direzioni. Inoltre, è una testina di *lettura e scrittura*, ossia può modificare al suo passaggio le celle visitate. La memoria è data dal nastro stesso quindi non è limitata a priori e può essere modificata in modo molto flessibile.
Possiamo assumere, senza perdere in generalità, che non ci siano ulteriori mosse possibili a partire da uno stato finale, quindi quando la stringa in input è accettata. Si noti però che, su stringhe che non sono accettate, la computazione potrebbe non terminare. Potremmo anche dare una definizione alternativa senza stati finali e dove una stringa è accettata se e solo se la computazione corrispondente termina: infatti, è facile vedere che si può sempre trasformare la macchina in modo che una computazione che termina in uno stato non finale diventi non terminante (ma ovviamente non viceversa).

Def. 3.2.2 [Linguaggi ricorsivamente enumerabili] Un linguaggio è *ricorsivamente enumerabile* (*r.e.*) se è accettato da una macchina di Turing.

In particolare, potrebbe accadere che per un linguaggio r.e. qualunque macchina che lo accetta non termini su qualche stringa non nel linguaggio. È conveniente distinguire, all'interno della classe dei linguaggi r.e., la sottoclasse dei linguaggi per cui invece esiste almeno una macchina di Turing accettante che termina sempre (si dice anche in questo caso che la macchina *riconosce* il linguaggio).

Def. 3.2.3 [Linguaggi ricorsivi] Un linguaggio è *ricorsivo* se è accettato da una macchina di Turing che termina su ogni input.

Vediamo alcuni esempi di macchine di Turing usate come riconoscitori, considerando $\Sigma = \{0, 1\}$. Avendo assunto che non ci siamo mai transizioni da uno stato finale, omettiamo la corrispondente riga nella tabella.

Esempio 3.2.4 [TM che accetta $0^\star 1^\star$] Come detto sopra, il set di istruzioni si può estendere aggiungendo la direzione nulla N in modo che $\delta(q, X) = \langle q', Y, \mathrm{N}\rangle$ sovrascriva la cella corrente con Y senza cambiare posizione.
Vediamo come definire la TM che riconosce il linguaggio regolare $L = \{0^n 1^m \mid n, m \geq 0\}$.
Possiamo schematizzare l'algoritmo nel modo seguente.

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

Usiamo una tabella per rappresentare la funzione di transizione di questa TM:

	0	1	B
q_0	$q_0, 0, R$	$q_1, 1, R$	q_2, B, N
q_1		$q_1, 1, R$	q_2, B, N

dove q_0 è lo stato iniziale e q_2 è l'unico stato finale. Le celle vuote corrispondono a casi in cui la funzione di transizione è indefinita, quindi la computazione si blocca, indicando che la stringa in input non appartiene al linguaggio accettato.
Vediamo un esempio di computazione di questa TM sull'input $u = 0011$.

$$\langle \epsilon, q_0, 0011\rangle \to \langle 0, q_0, 011\rangle \to \langle 00, q_0, 11\rangle \to \langle 001, q_1, 1\rangle \to \langle 0011, q_1, \epsilon\rangle \to \langle 0011, q_2, \epsilon\rangle$$

Un altro esempio su 010:

$$\langle \epsilon, q_0, 010\rangle \to \langle 0, q_0, 10\rangle \to \langle 01, q_1, 0\rangle \not\to$$

dove l'ultima configurazione non ha successori.

È possibile utilizzare un interprete, per esempio `http://morphett.info/turing/turing.html`, che usa le seguenti convenzioni sintattiche:

- ogni linea contiene una tupla della forma: `<stato> <simbolo> <nuovo simbolo> <direzione> <nuovo`
- lo stato iniziale è rappresentato con `0`
- il blank è rappresentato con _

- la direzione è `l`, `r` o `*`
- la macchina si ferma in uno stato che inizia con `halt`

Usando la sintassi del simulatore, possiamo riscrivere il programma (ed eseguirlo) nel modo seguente:

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

Esempio 3.2.5 [TM che accetta 0^n1^n] Vediamo ora come definire una TM che riconosce il linguaggio (context-free) $L = \{0^n1^n \mid n \geq 1\}$.
La macchina sovrascrive il primo 0 che trova con X, poi il primo 1 che trova con Y, torna indietro e ricomincia se ci sono ancora 0. Se non ci sono più 0 allora scorre tutto l'input modificato controllando che non ci siano più 1. La tabella è la seguente:

	0	1	X	Y	B
q_0	q_1, X, R				
q_1	$q_1, 0, R$	q_2, Y, L		q_1, Y, R	
q_2	$q_2, 0, L$		q_3, X, R	q_2, Y, L	
q_3	q_1, X, R			q_4, Y, R	
q_4				q_4, Y, R	q_5, B, N

dove q_0 è lo stato iniziale e q_5 è lo stato finale.
Vediamo un esempio di esecuzione di questa TM sull'input $u = 0011$.

$$\langle \epsilon, q_0, 0011\rangle \to \langle X, q_1, 011\rangle \to \langle X0, q_1, 11\rangle \to \langle X, q_2, 0Y1\rangle \to \langle \epsilon, q_2, X0Y1\rangle \to \langle X, q_3, 0Y1\rangle \to \langle XX, q_1, Y1\rangle$$
$$\langle XXY, q_1, 1\rangle \to \langle XX, q_2, YY\rangle \to \langle X, q_2, XYY\rangle \to \langle XX, q_3, YY\rangle \to \langle XXY, q_4, Y\rangle \to \langle XXYY, q_4, \epsilon\rangle \to \langle$$

Con la sintassi del simulatore:

```
0       0 X r goR
goR     0 0 r goR
goR     1 Y l goL
goR     Y Y r goR
goL     0 0 l goL
goL     X X r test
goL     Y Y l goL
test    0 X r goR
test    Y Y r check
check   Y Y r check
check   _ _ * halt-accept
```

dove $q_0 =$ `0`, $q_1 =$ `goR`, $q_2 =$ `goL`, $q_3 =$ `test`, $q_4 =$ `check`, $q_5 =$ `halt-accept`.

3.2.2 Funzioni T-calcolabili

Vediamo ora come utilizzare le TM per calcolare funzioni, quindi come un "calcolatore ideale", secondo quanto introdotto nell'articolo originale di Turing. A tale scopo, è sufficiente considerare una definizione più semplice in cui non distinguiamo l'alfabeto di input da quello del nastro ($\Sigma = \Gamma$), e non consideriamo gli stati finali, quindi $\mathcal{M} = \langle Q, \Sigma, \delta, q_0, B\rangle$, con almeno un altro simbolo in Σ oltre a B, e $\delta\colon Q \times \Sigma \to Q \times \Sigma \times \{L, R\}$.
Consideriamo macchine di Turing su un alfabeto fissato Σ. Per calcolare una funzione $f\colon$ *Input* $\to$ *Output*, occorrerà anzitutto dare una *codifica* che rappresenti gli elementi di dominio e codominio come stringhe su Σ, formalmente due funzioni iniettive $c_{\text{IN}}\colon$ *Input* $\to \Sigma^\star$ e $c_{\text{OUT}}\colon$ *Output* $\to \Sigma^\star$. Inoltre, in accordo con quanto anticipato nella Sezione 1.2, occorrerà fissare delle *direttive di input/output*, per esempio le seguenti.

- Come direttiva di input, si può assumere come fatto in precedenza che la testina sia inizialmente posizionata sul primo simbolo della stringa che rappresenta l'input.
- Come direttiva di output, possiamo assumere che il risultato della funzione sia rappresentato dalla stringa che rimane sul nastro quando l'esecuzione termina, se questa è effettivamente la rappresentazione di un risultato, altrimenti la funzione risulta indefinita.

Formalmente, $f_{\text{IN}}(i) = \langle \epsilon, q_0, c_{\text{IN}}(i)\rangle$, e $f_{\text{OUT}}(\langle \alpha, q, \beta\rangle) = \begin{cases} o & \text{se } c_{\text{OUT}}(\alpha\beta) = o \\ \text{indefinito} & \text{altrimenti.} \end{cases}$
La *funzione calcolata* dalla macchina $f_{\mathcal{M}}\colon$ *Input* $\to$ *Output* risulta quindi essere:

per ogni $i \in$ *Input*, $f_{\mathcal{M}}(i) = o$ se $\langle \epsilon, q_0, c_{\text{IN}}(i)\rangle \to^\star \langle \alpha, q, \beta\rangle$ tale che $\langle \alpha, q, \beta\rangle \not\to$, $c_{\text{OUT}}(\alpha\beta) = o$.

Si noti che la funzione calcolata è in generale una funzione parziale, in quanto su qualche input la computazione potrebbe non terminare, o terminare in una configurazione di arresto mal formata. Si può comunque sempre trasformare la macchina in modo che una computazione che termina con un output mal formato diventi non terminante (ma ovviamente non viceversa), quindi nel seguito assumeremo che se una macchina termina l'output sia sempre la rappresentazione di un risultato.
Diremo che una funzione è *T-calcolabile* se è la funzione calcolata da una qualche macchina di Turing. Si può formalizzare e provare che questa definizione non dipende dall'alfabeto scelto per le macchine di Turing, nè dalle codifiche e direttive input/output purché "ragionevoli".
L'utilizzo di una macchina di Turing come riconoscitore di un linguaggio su Σ si può ottenere come caso particolare considerando funzioni da $\Sigma^\star$ in $\{0, 1\}$. La nozione di funzione T-calcolabile corrisponde a quella di linguaggio r.e. (esiste una macchina che può non terminare su qualche input), mentre quella di funzione T-calcolabile totale corrisponde a quella di linguaggio ricorsivo (esiste una macchina che termina sempre).
Vediamo ora un esempio concreto di codifica. Consideriamo le funzioni $f\colon \mathbb{N}^k \to \mathbb{N}$. La codifica tradizionalmente considerata per i naturali è quella *unaria*, ossia come sequenze di 1 (per esempio, la codifica di 5 è $\overline{5} = 1^5 = 11111$). Una n-upla di argomenti $x_1, \ldots, x_n$ può essere rappresentata utilizzando un separatore. Per esempio, utilizzando B come separatore, come $\overline{x_1}B \ldots B\overline{x_n}$. Un esempio di output mal formato è $1B1$.
Vediamo ora alcuni esempi di programmazione con macchine di Turing.

Esempio 3.2.6 [TM che calcola somma] Vediamo come definire una TM che calcola la somma di due numeri codificati in unario separati da $\#$, quindi $1^n\#1^m$ deve dare come output 1^{n+m}. L'algoritmo sposta semplicemente tutti gli 1 prima di $\#$ alla fine del secondo argomento cancellando alla fine il separatore. La tabella è la seguente:

	1	#	B
q_0	q_1, B, R	q_5, B, R	
q_1	$q_1, 1, R$	$q_1, \#, R$	$q_2, 1, L$
q_2	$q_2, 1, L$	$q_2, \#, L$	q_3, B, R
q_3	q_1, B, R	q_4, B, R	

dove q_0 è lo stato iniziale e q_5 è lo stato finale.

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

3.2.3 Tesi di Church-Turing

Abbiamo visto in Sezione 3.2.2 che le macchine di Turing possono essere usate per calcolare funzioni, in modo analogo a un linguaggio di programmazione. Notiamo che il linguaggio di programmazione delle TM non ha a disposizione istruzioni tipo *goto* per saltare da una cella all'altra. Inoltre, le celle non sono identificate da indici che possiamo usare nei programmi, come accade invece in un linguaggio di programmazione. Un salto dalla cella corrente a una cella che contiene un certo simbolo si può comunque realizzare tramite una serie di spostamenti elementari in una certa direzione.
Questo tipo di macchina può essere vista come una versione ancora grezza dell'architettura di Von Neumann, dove invece la memoria è ad accesso diretto e le istruzioni manipolano direttamente i valori e gli indirizzi di memoria invece del contenuto del nastro. Provando a scrivere programmi ci si rende conto subito della difficoltà addizionale di sviluppare programmi su una TM.
Altri formalismi proposti, come le funzioni μ-ricorsive (che vedremo in Sezione 3.4.1), il lambda-calcolo, le Random Access Machine, le macchine a contatori, sono tutti risultati essere equivalenti alle macchine di Turing (nel senso che definiscono la stessa classe di funzioni calcolabili). Inoltre, è possibile definire interpreti di altri linguaggi attraverso TM che prendono in input programmi e che simulano la loro semantica. In particolare, una TM in grado di simulare altre TM viene chiamata *universale* e rappresenta l'idea alla base dei calcolatori, come vedremo in Sezione 3.2.5.
Sembra quindi possibile identificare la classe delle funzioni T-calcolabili con quella delle funzioni "calcolabili con un algoritmo", come espresso dalla celebre *tesi di Church-Turing*. Evidentemente la tesi di Church-Turing non può essere dimostrata, tuttavia esistono diversi argomenti significativi in suo sostegno.

1. Tutti i formalismi proposti per esprimere algoritmi sono risultati equivalenti, nel senso che si è provato che le classi di funzioni da esse definite coincidono. Questa classe è stata chiamata *classe delle funzioni ricorsive (parziali)*.

2. Ogni funzione che si possa ragionevolmente considerare algoritmica è risultata appartenere a questa classe (cioè, in ogni formalismo è possibile esprimere un algoritmo che la calcola).

3. Tutte le dimostrazioni di equivalenza tra due formalismi di cui al punto 1) hanno la seguente struttura: esiste un algoritmo (che ha come input/output dati simbolici) che, dato un programma nel primo formalismo, produce un programma che calcola la stessa funzione nel secondo.

Osservazione: i primi due punti suffragano la congettura che ogni formalismo riesca a calcolare tutte le funzioni intuitivamente algoritmiche, mentre il terzo punto può essere interpretato nel senso che ogni formalismo fornisce tutti i possibili algoritmi.

Tesi di Church-Turing

> Una funzione è calcolabile con un algoritmo se e solo se è ricorsiva (parziale).

Quindi la tesi consiste nell'identificazione fra la classe, definita intuitivamente, delle funzioni calcolabili con un algoritmo con la classe delle funzioni definite da uno qualunque dei formalismi proposti ed equivalenti, per esempio le funzioni T-calcolabili.
Un formalismo per descrivere algoritmi equivalente alle TM, ossia in grado di definire tutte le funzioni calcolabili, si dice *Turing-completo*.

Uso della tesi di Church-Turing nelle dimostrazioni

> Consiste nel considerare come funzione ricorsiva (e quindi formalizzabile in uno qualunque dei formalismi esistenti) ogni funzione descritta tramite un algoritmo informale (cioè non espresso in un formalismo). Ciò corrisponde all'uso di dimostrazioni "non formalizzate", nella matematica usuale. Precisamente: come ogni dimostrazione matematica (corretta!) può essere formalizzata usando una assiomatizzazione della teoria degli insiemi e una teoria logica formale, così ogni dimostrazione (corretta!) che usa la tesi di Church può essere formalizzata nell'ambito di uno dei formalismi esistenti.

3.2.4 Numerazione algoritmica delle funzioni ricorsive

Analogamente a quanto visto precedentemente per le funzioni primitive ricorsive, è possibile codificare come stringa su un alfabeto fissato Σ una qualsiasi TM.
Anzitutto, possiamo fare in modo che tutte le macchine abbiamo un insieme di stati e un alfabeto contenuti in un insieme infinito prefissato (il funzionamento della macchina è infatti indipendente dai nomi scelti per stati e simboli). In altre parole, ragionando modulo isomorfismo, possiamo lavorare sempre con stati e simboli numerati in $\mathbb{N}$. Poi, possiamo codificare (il numero d'ordine di) stati e simboli come stringa sull'alfabeto Σ. Analogamente possiamo codificare le direzioni L, R. Una quintupla può essere codificata come la sequenza dei numeri d'ordine dei corrispondenti elementi, utilizzando un separatore. Per esempio, assumendo $\Sigma = \{0, 1\}$, possiamo codificare L e R con 0 e 00, q_0 e q_1 con 000 e 0000, B, X e Y con 00000, 000000 e 0000000. Possiamo quindi rappresentare la quintupla $\langle q_0, X, q_1, Y, \mathrm{R}\rangle$ con la stringa $000100000010000010000000100$. La funzione di transizione si può codificare come la concatenazione delle stringhe che rappresentano le quintuple, utilizzando come separatore 11. A questo punto avremo quindi una codifica, che indicheremo con $\langle\mathcal{M}\rangle$, di ogni macchina $\mathcal{M}$.

Prop. 3.2.7 Le macchine di Turing possono essere numerate *effettivamente* (ossia, mediante un algoritmo).

Prova Una volta rappresentate tutte le macchine come stringhe su un certo alfabeto, si può procedere per elencazione analogamente a quanto visto per i programmi che calcolano funzioni ricorsive primitive. □

Indichiamo nel seguito con $\mathcal{M}_x$ la macchina di Turing con indice x nella numerazione.
Consideriamo, per esempio, funzioni da $\mathbb{N}$ in $\mathbb{N}$, e indichiamo con ϕ_x la funzione da $\mathbb{N}$ in $\mathbb{N}$ calcolata dalla x-sima macchina di Turing.

Teorema 3.2.8 Le funzioni ricorsive (da $\mathbb{N}$ in $\mathbb{N}$) possono essere numerate effettivamente.

Prova Immediata dalla proposizione precedente e dalla tesi di Church. □
Osservazioni:

1. Chiaramente, potremmo numerare analogamente le funzioni ricorsive da A in B, per qualunque scelta di A e B, assumendo opportune codifiche.

2. $n \neq m \not\Rightarrow \phi_n \neq \phi_m$ (algoritmi diversi possono calcolare la stessa funzione).

3. Conoscere un algoritmo per calcolare una funzione ricorsiva f è equivalente a conoscere almeno un numero n per cui $f = \phi_n$, in una certa numerazione (infatti conoscendo un algoritmo possiamo tradurlo in una macchina di Turing, e dalla rappresentazione di questa ricavare l'indice n; viceversa dato n possiamo costruire la (rappresentazione della) macchina corrispondente e quindi un algoritmo).

4. Si noti la differenza tra "f è ricorsiva" e "si conosce n tale che $f = \phi_n$" (analoga alla differenza già vista a pagina 36 tra "f è ricorsiva primitiva" e "conosciamo un programma che calcola f").

Corollari:

1. Le funzioni ricorsive hanno la cardinalità del numerabile, come anche le funzioni ricorsive totali (perché tutte le funzioni costanti sono ricorsive totali e per il teorema sopra).[4]

2. Esistono funzioni, parziali e totali, non ricorsive (le funzioni caratteristiche dei sottoinsiemi di $\mathbb{N}$ hanno la cardinalità di $\wp(\mathbb{N})$, che è strettamente maggiore).

3. Ogni funzione ricorsiva ha un'infinità numerabile di indici nella numerazione effettiva (basta aggiungere stati nuovi e righe della tabella inutili).

Riassumendo, possiamo dire che esiste una corrispondenza biunivoca tra tutte le possibili macchine di Turing (algoritmi), tutte le stringhe (su un certo alfabeto Σ) che le rappresentano (rappresentazioni degli algoritmi), e i numeri naturali (indici degli algoritmi). Il modo in cui questa corrispondenza è costruita non è sostanziale nello sviluppo della teoria della calcolabilità. Essenziale è la possibilità di vedere le stringhe (o, ancora più astrattamente, i numeri naturali) al tempo stesso come *dati* (argomenti del calcolo) e *programmi* (macchine che eseguono il calcolo). Questo permette di applicare ai formalismi analizzati finora uno dei principi alla base dell'informatica, ossia quello di poter *passare programmi come argomenti ad altri programmi*, come vedremo nella prossima sezione.

3.2.5 Macchina di Turing universale e calcolatore

È chiaro da quanto precede che una macchina di Turing può essere vista come un programma. Questo programma è fissato, nel senso che non vi è modo di modificare la funzione di transizione.
D'altra parte un calcolatore prende come dati in ingresso dei programmi (con i loro dati). Quindi un calcolatore è a sua volta un programma e deve quindi corrispondere, in base alla tesi di Church, a una macchina di Turing. Questa considerazione intuitiva trova una verifica e una formulazione esatta nel seguente teorema (per funzioni in una variabile).

[4]Tuttavia, le funzioni ricorsive totali *non* possono essere numerate effettivamente, perché, intuitivamente, non possiamo stabilire in modo algoritmico se un algoritmo calcola una funzione totale, come verrà espresso in modo più preciso nel seguito.

Teorema 3.2.9 La funzione $f_U : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ definita nel modo seguente:

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

è ricorsiva, quindi esiste una macchina di Turing che la calcola.

Prova La funzione f_U può essere calcolata mediante il seguente algoritmo:

```
input x,y
genero M_x
return M_x(y)
```

Per la tesi di Church f_U è ricorsiva. □
Questo teorema si generalizza a funzioni in un qualunque numero di variabili (anche se la formulazione precedente ha la stessa generalità, sfruttando un'opportuna codifica dell'input).
La macchina di Turing che calcola f_U definita dal teorema precedente si dice *macchina di Turing universale*.
Nella formulazione data sopra, basata sulla numerazione, la macchina universale agisce in due passi:

- genera la macchina $\mathcal{M}_x$ corrispondente a x
- interpreta tale programma, simulandone il comportamento sull'input y.

Una formulazione alternativa che non passa attraverso la numerazione e si concentra sul secondo passo (interprete) è la seguente: esiste una macchina di Turing (su un certo alfabeto Σ) che, presa in input una stringa (su Σ) $\langle \mathcal{M}, u \rangle$ che codifica una coppia formata da una macchina e un input per essa (ottenuta estendendo opportunamente la codifica delle macchine vista precedentemente), restituisce in output una stringa (su Σ) che codifica l'output di $\mathcal{M}$ su u, se $\mathcal{M}$ termina su u, altrimenti non termina.

3.3 Risolvibilità di problemi

Come menzionato nella parte introduttiva, la Teoria della Calcolabilità ha tra i suoi obiettivi principali quello di identificare la classe dei problemi risolvibili tramite un elaboratore. Il risultato forse più sorprendente della Teoria della Calcolabilità è l'esistenza di problemi *irrisolvibili*. Una volta individuati alcuni problemi irrisolvibili, possiamo utilizzare la nozione di *riduzione* tra problemi per provare che anche altri lo sono.

3.3.1 Problema dell'arresto

Fissiamo un formalismo per esprimere algoritmi (per esempio, un linguaggio di programmazione). Esiste un programma che, per ogni programma P e per ogni input y, avendo come input P e y, stabilisca se il programma P eseguito su y termina o no? Ossia, ci chiediamo se è possibile dare un algoritmo che stabilisca, esaminando unicamente il (codice del) programma P e l'input, se P termina su y. La risposta a questa domanda è negativa. Si noti che simulare l'esecuzione del programma (cosa possibile utilizzando la macchina universale) fornisce una risposta (positiva) solo se il programma termina. Ovviamente, è sempre possibile provare la terminazione di *un* particolare programma.
La risposta negativa si ottiene facilmente formalizzando il problema. Allora un programma è (per esempio) una macchina di Turing con un certo indice nella numerazione, sia $\mathcal{M}_x$, che calcola la funzione ϕ_x, e assegnare il programma equivale ad assegnare l'indice x corrispondente.

Teorema 3.3.1 [Indecidibilità del problema dell'arresto] La funzione $f_{\mathcal{H}} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ definita da:

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

non è ricorsiva.

Prova Il teorema è un'immediata conseguenza del seguente lemma. □

Lemma 3.3.2 La funzione $f_{\mathcal{K}} \colon \mathbb{N} \to \mathbb{N}$ definita da:

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

non è ricorsiva.

Prova Se $f_{\mathcal{K}}$ fosse ricorsiva, ossia calcolabile con un certo algoritmo $\mathcal{M}_{\mathcal{K}}$, potremmo calcolare una funzione g con il seguente algoritmo:

```
input x
if (M_K(x) = 1) non terminazione
else return 1
```

La funzione g sarebbe algoritmica e quindi, per la tesi di Church, esisterebbe $\bar{z}$ tale che $g = \phi_{\bar{z}}$, quindi si avrebbe:

$$\phi_{\bar{z}}(\bar{z})\uparrow \Leftrightarrow g(\bar{z})\uparrow \Leftrightarrow f_{\mathcal{K}}(\bar{z}) = 1 \Leftrightarrow \phi_{\bar{z}}(\bar{z})\downarrow$$
$$\phi_{\bar{z}}(\bar{z}) = 1 \Leftrightarrow g(\bar{z}) = 1 \Leftrightarrow f_{\mathcal{K}}(\bar{z}) = 0 \Leftrightarrow \phi_{\bar{z}}(\bar{z})\uparrow$$

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

3.3.2 Insiemi ricorsivi e ricorsivamente enumerabili

Il teorema dell'arresto afferma che la funzione

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

non è ricorsiva. Alternativamente, possiamo esprimere il teorema dicendo che $\mathcal{H} = \{\langle x,y\rangle \mid x,y \in \mathbb{N}, \phi_x(y)\downarrow\}$ è un sottoinsieme di $\mathbb{N}\times\mathbb{N}$ non ricorsivo, e analogamente $\mathcal{K} = \{x \mid x \in \mathbb{N}, \phi_x(x)\downarrow\}$.
Infatti, le nozioni di insieme ricorsivo e ricorsivamente enumerabile viste precedentemente possono essere anche formulate nel modo seguente.

> Un insieme $A \subseteq \mathbb{N}$ è *ricorsivamente enumerabile (r.e.)* se esiste una funzione ricorsiva $f \colon \mathbb{N} \to \{0,1\}$ tale che
>
> $$f(x) = \begin{cases} 1 & \text{se } x \in A \\ 0 \text{ oppure indefinito} & \text{altrimenti} \end{cases}$$
>
> ossia esiste una macchina di Turing (in generale, un algoritmo) tale che
>
> $$\text{per ogni input } x \in \mathbb{N} \begin{cases} \text{restituisce } 1 & \text{se } x \in A \\ \text{restituisce } 0 \text{ oppure non termina} & \text{altrimenti} \end{cases}$$

Equivalentemente (perché?), possiamo dire che la funzione $\psi_A \colon \mathbb{N} \to \{0,1\}$:

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

(detta *funzione semicaratteristica* di A) è ricorsiva[5] , ossia esiste una macchina di Turing (in generale, un algoritmo) tale che

$$\text{per ogni input } x \in \mathbb{N} \begin{cases} \text{restituisce } 1 & \text{se } x \in A \\ \text{non termina} & \text{altrimenti} \end{cases}$$

Un insieme $A \subseteq \mathbb{N}$ è *ricorsivo* se la funzione (totale) $\chi_A \colon \mathbb{N} \to \{0,1\}$ definita da

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

(detta *funzione caratteristica* di A) è ricorsiva. Ossia, esiste una macchina di Turing (in generale, un algoritmo) tale che

$$\text{per ogni input } x \in \mathbb{N} \begin{cases} \text{restituisce } 1 & \text{se } x \in A \\ \text{restituisce } 0 & \text{altrimenti} \end{cases}$$

Naturalmente le definizioni possono essere date analogamente per sottoinsiemi di un insieme arbitrario, ed è facile vedere che per i linguaggi su un alfabeto Σ (ossia, i sottoinsiemi di $\Sigma^\star$) sono equivalenti a quelle date in precedenza (Definizione 3.2.2 e Definizione 3.2.3).
Indicando con $\mathbb{R}$ la classe degli insiemi ricorsivi e con $\mathbb{RE}$ la classe degli insiemi ricorsivamente enumerabili, si ha ovviamente $\mathbb{R} \subseteq \mathbb{RE}$. Infatti dato $A \in \mathbb{R}$, la funzione caratteristica di A soddisfa anche la condizione richiesta dalla definizione di insieme ricorsivamente enumerabile.
Esempi di insiemi ricorsivi sono tutti gli insiemi finiti e tutti quelli il cui complementare è finito (perché?).
Le due classi non coincidono, per esempio l'insieme $\mathcal{H}$ è ricorsivamente enumerabile ma non ricorsivo. La relazione tra ricorsivo e ricorsivamente enumerabile è messa in luce dal seguente teorema, dove indichiamo con $\overline{A}$ il complementare di A.

Teorema 3.3.3 [Teorema di Post] A ricorsivo $\Leftrightarrow$ $\overline{A}$ ricorsivo $\Leftrightarrow$ A e $\overline{A}$ ricorsivamente enumerabili.

Prova

$\Rightarrow$ Se A è ricorsivo ho una TM (algoritmo) $\mathcal{M}_A$ che termina sempre e restituisce 1 se l'input appartiene ad A, 0 altrimenti. Allora è immediato vedere che anche $\overline{A}$ è ricorsivo, considerando l'algoritmo che su un input x restituisce 0 se $\mathcal{M}_A(x)$ restituisce 1 e viceversa. Quindi sono entrambi ricorsivamente enumerabili.

$\Leftarrow$ Se A e $\overline{A}$ sono entrambi r.e., abbiamo due TM (algoritmi) che terminano con output 1 rispettivamente se l'input appartiene ad A e se non appartiene. Basta allora eseguire le due macchine *in interleaving*, ossa a turno una o più passi della prima TM e uno o più passi della seconda e prima o poi una delle due terminerà. □

Di conseguenza, il teorema dell'arresto asserisce anche che $\overline{K}$ *non* è ricorsivamente enumerabile (perché?).

[5] Altra formulazione equivalente (perché?): A è il dominio di definizione di una funzione ricorsiva.

Esercizio 3.3.4

- Si esprima in modo più preciso la prova di $\Leftarrow$ data sopra utilizzando pseudocodice, e indicando con $\mathcal{M}_A^k$ e $\mathcal{M}_{\overline{A}}^k$ per l'esecuzione di k passi delle due macchine, che restituisce 0 se non si ottiene 1.

- Siano A, B due insiemi ricorsivi. Si provi che anche $\overline{A}$, $A \cup B$, $A \cap B$ lo sono. Inoltre, se A e B sono linguaggi su Σ, si provi che anche $A \cdot B$ e $A^\star$ sono ricorsivi.

- Siano A, B due insiemi ricorsivamente enumerabili. Si provi che anche $A \cup B$ e $A \cap B$ lo sono. Inoltre, se A e B sono linguaggi su Σ, si provi che anche $A \cdot B$ e $A^\star$ sono ricorsivamente enumerabili.

La terminologia "ricorsivamente enumerabile" fa riferimento a una definizione alternativa che si può provare essere equivalente a quella che abbiamo dato: un insieme è ricorsivamente enumerabile se esiste un algoritmo per numerare (con ripetizioni) gli elementi di A, ossia questi si ottengono uno dopo l'altro come risultati corrispondenti agli input di un programma. Gli insiemi r.e. possono quindi essere visti come quelli *generabili con un algoritmo*.

Teorema 3.3.5 [Caratterizzazione degli insiemi r.e.] Le seguenti affermazioni sono equivalenti:

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

Prova

(1)⇒(2) Se A è r.e., sappiamo che esiste una TM (algoritmo) che, per ogni $x \in \mathbb{N}$, termina e restituisce 1 se $x \in A$, mentre non termina se $x \notin A$. Possiamo modificare l'algoritmo in modo tale che, per ogni $x \in \mathbb{N}$, invece di restituire 1 restituisca x. Questo algoritmo definisce una funzione ricorsiva f il cui insieme di valori è A.

(2)⇒(3) Notiamo anzitutto che il punto precedente *non* fornisce un algoritmo per generare tutti gli elementi di A, in quanto il calcolo di $f(x)$ potrebbe non terminare su qualche x. Ossia, considerando una matrice dove le righe sono gli input e le colonne sono "tempi" o "numero di passi di computazione"

		passi				
		0	*1*	...	*y*	...
input	*0*					
	1					
	...					
	x					
	...					

e la casella $\langle x, y\rangle$ rappresenta il risultato del calcolo di $f(x)$ dopo y passi, non possiamo calcolare gli elementi della matrice "per righe" in quanto potremmo finire in una computazione non terminante. D'altra parte, non possiamo neanche calcolare gli elementi della matrice "per colonne" in quanto gli input sono un insieme infinito. Tuttavia, se calcoliamo gli elementi della matrice "a zig zag"[6], siamo sicuri di ottenere prima o poi tutti i casi di terminazione. In particolare, dato che A non è vuoto, esiste una prima coppia che produce un risultato, sia z, ottenuta dopo un certo numero di passi (ogni passo è un elemento della matrice), sia n_0, dell'algoritmo a zig-zag. Definiamo la seguente funzione $g\colon \mathbb{N} \to \mathbb{N}$ dove l'argomento n rappresenta il passo n-simo di esecuzione dell'algoritmo a zig-zag.

$$g(n) = z \text{ per ogni } n \leq n_0$$
$$g(n+1) = \begin{cases} z' & \text{se il passo } n+1 \text{ produce un risultato } z' \\ g(n) & \text{altrimenti} \end{cases}$$

Questa funzione è chiaramente ricorsiva totale (anzi, primitiva ricorsiva), e il suo insieme di valori è A.

(3)⇒(1) Se $A = \emptyset$, è chiaramente ricorsivamente enumerabile. Altrimenti, sia f la funzione totale il cui insieme di valori è A. Definiamo allora la funzione g nel modo seguente:

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

Questa funzione ha chiaramente A come dominio di definizione, ed è calcolabile con un algoritmo: basta eseguire l'algoritmo che genera successivamente tutti gli elementi di A.

Un esempio importante di insieme non ricorsivamente enumerabile è $\mathcal{T} = \{x \mid \phi_x \text{ totale}\}$. Possiamo per prima cosa provare che $\mathcal{T}$ non è ricorsivo. Infatti, se lo fosse, la seguente funzione

$$f(x) = \begin{cases} \phi_x(x) + 1 & \text{se } \phi_x \text{ totale} \\ 0 & \text{altrimenti} \end{cases}$$

risulterebbe ricorsiva e totale. Ma allora avrei $f = \phi_{\bar{x}}$ per un certo indice $\bar{x}$, e quindi $f(\bar{x}) = \phi_{\bar{x}}(\bar{x}) = \phi_{\bar{x}}(\bar{x}) + 1$.
Proviamo ora che $\mathcal{T}$ non è neanche ricorsivamente enumerabile. Infatti, se lo fosse, avremmo una numerazione algoritmica delle funzioni totali, ossia una funzione calcolabile totale f tale che l'immagine di f sia $\mathcal{T}$, ossia un algoritmo per ottenere, a partire da un indice y, l'indice (nella numerazione di tutte le funzioni ricorsive) della y-sima funzione totale. Consideriamo allora la funzione

$$g(y) = \phi_{f(y)}(y) + 1$$

Questa funzione è ricorsiva, quindi $g = \phi_{\bar{x}}$ per un certo indice x. Inoltre, è totale, quindi $\bar{x} \in \mathcal{T}$, quindi esiste un $\bar{y}$ tale che $f(\bar{y}) = \bar{x}$. Quindi si avrebbe per definizione:

$$g(\bar{y}) = \phi_{f(\bar{y})}(\bar{y}) + 1 = \phi_{\bar{x}}(\bar{y}) + 1$$
$$g(\bar{y}) = \phi_{\bar{x}}(\bar{y})$$

che, essendo g totale, è un assurdo.

[6] Un altro nome per questo modo di procedere è *dovetailing* (a coda di rondine).

3.3.3 Problemi decidibili e indecidibili

Un'ulteriore terminologia equivalente molto usata considera la nozione informale di *problema di decisione*, illustrata dai seguenti esempi.

- Decidere se un grafo è aciclico, se una lista è ordinata, se una sequenza compare come sottosequenza di un'altra, ...
- Decidere se una stringa su $\{0,1\}$ ha un numero pari di 1.
- Data una stringa u e un linguaggio su un alfabeto Σ, decidere se $u \in L$.
- Dati un algoritmo e il suo input, decidere se l'algoritmo termina su quell'input (problema dell'arresto), ossia, dati due numeri naturali x, y, decidere se ϕ_x è definita su y.
- Decidere se un algoritmo termina sempre, ossia, dato un numero naturale x, se ϕ_x è totale.
- Dati due linguaggi context-free, decidere se la loro intersezione è vuota.

Un problema di decisione può essere formalizzato come funzione $\mathcal{P}$ da un certo insieme A, l'insieme delle *istanze* del problema, in $\{0,1\}$ (ossia, come *predicato* su A, vedi Definizione 4.1.2), oppure, equivalentemente, come l'insieme $\{a \in A \mid \mathcal{P}(a) = 1\}$.
Allora si dice:

> $\mathcal{P}$ è *decidibile* se $\{a \in A \mid \mathcal{P}(a) = 1\}$ è ricorsivo
> $\mathcal{P}$ è *semidecidibile* se $\{a \in A \mid \mathcal{P}(a) = 1\}$ è ricorsivamente enumerabile

Quindi un problema di decisione $\mathcal{P}$ è decidibile se esiste un algoritmo che, per ogni istanza del problema, fornisce risposta positiva o negativa in un numero finito di passi; è semidecidibile se esiste un algoritmo che, sulle istanze x per cui $\mathcal{P}(x)$ è vero, fornisce una risposta positiva in un numero finito di passi, mentre, sulle istanze x per cui $\mathcal{P}(x)$ è falso, potrebbe anche non terminare.
Il problema dell'arresto è un esempio di problema non decidibile, ma semidecidibile, mentre il problema della totalità è un problema non semidecidibile.

3.3.4 Riducibilità

Questa nozione è assolutamente analoga a quella di *riducibilità polinomiale* vista nel corso di Analisi e Progettazione di Algoritmi, con la sola differenza che si richiede una funzione di riduzione semplicemente *calcolabile* anziché *calcolabile polinomialmente*.
Consideriamo per esempio come problemi i sottoinsiemi di numeri naturali.

Def. 3.3.6 Dati due problemi $\mathcal{P}$ e $\mathcal{Q}$, diciamo che $\mathcal{P}$ *è riducibile a* $\mathcal{Q}$, e scriviamo $\mathcal{P} \leq \mathcal{Q}$, se esiste una funzione totale ricorsiva $f\colon \mathbb{N} \to \mathbb{N}$, detta *funzione di riduzione*, tale che, per ogni $x \in \mathbb{N}$, $x \in \mathcal{P}$ se e solo se $f(x) \in \mathcal{Q}$.

Naturalmente si ha una definizione analoga considerando sottoinsiemi di $\mathbb{N}^k$ o $\Sigma^\star$, e i due problemi potrebbero anche essere sottoinsiemi di due insiemi diversi (vedi ultimo esempio della sezione).
Intuitivamente, $\mathcal{Q}$ è "più difficile" (o meglio, "non più facile") di $\mathcal{P}$. Infatti, è facile vedere che, dato un algoritmo $\mathcal{M}_\mathcal{Q}$ che decide $\mathcal{Q}$, un algoritmo $\mathcal{M}_\mathcal{P}$ che decide $\mathcal{P}$ può essere ottenuto eseguendo, a partire dall'input x, prima l'algoritmo $\mathcal{M}_f$ che calcola $f(x)$, poi l'algoritmo $\mathcal{M}_\mathcal{Q}$ con input $f(x)$. Infatti, è facile vedere che $\mathcal{M}_\mathcal{P}$ è effettivamente un algoritmo che termina sempre (perché?) e restituisce 1 se $x \in \mathcal{P}$ e 0 se $x \notin \mathcal{P}$ (perché?). Una proprietà analoga vale per la semidecidibilità.
La riducibilità può quindi essere utilizzata per provare due cose:

- se $\mathcal{Q}$ è ricorsivo (r.e.) allora anche $\mathcal{P}$ è ricorsivo (r.e.)
- se $\mathcal{P}$ *non* è ricorsivo (r.e.), allora non può esserlo neanche $\mathcal{Q}$ (r.e.) (Perché?)

Esercizio 3.3.7 La relazione di riducibilità è riflessiva e transitiva.

Diamo un primo esempio banale per illustrare l'idea.[7]

Esempio 3.3.8 L'insieme dei numeri pari è riducibile all'insieme dei multipli di 4. Possiamo infatti dare come funzione di riduzione $f(n) = 2n$. Questa funzione è chiaramente calcolabile, totale, e tale che n è pari se e solo se $2n$ è multiplo di 4. A questo punto, è chiaro che dati due programmi (che possiamo pensare come forniti da una libreria) x e y, il primo dei quali calcola f (ossia $\phi_x = f$), il secondo dei quali controlla se un numero è multiplo di 4, possiamo ottenere un programma che controlla se un numero è pari semplicemente applicando all'input n prima l'algoritmo x e poi l'algoritmo y.

L'esempio precedente non è molto significativo perché comunque sappiamo già che entrambi gli insiemi sono ricorsivi. Vediamo ora un esempio in cui trovare una riduzione ci consente di provare che un insieme non è ricorsivo.

Esempio 3.3.9 Consideriamo l'insieme $\mathcal{K} = \{x \mid \phi_x(x)\downarrow\}$ e l'insieme $\mathcal{Z} = \{x \mid \phi_x \text{ è la funzione costante } 0\}$. Mostriamo che $\mathcal{K} \leq \mathcal{Z}$. Questa implicazione mostra che anche l'insieme $\mathcal{Z}$ è non ricorsivo, ossia non è possibile decidere se un algoritmo restituisce 0 su ogni input.
In questo caso, l'input del problema $\mathcal{K}$ è (la descrizione di) un algoritmo x, e dobbiamo trasformarlo in una nuovo algoritmo $g(x)$ in modo tale che ϕ_x sia definita su x se e solo se $\phi_{g(x)}$ è la funzione costante 0.
È chiaro che questo si può ottenere costruendo l'algoritmo $g(x)$ nel modo seguente:

input $y \to$ invoco $\mathcal{M}_x(x) \to$ se termina restituisco 0

In questo modo si ha:

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

Allora g è una funzione di riduzione da $\mathcal{K}$ in $\mathcal{Z}$, in quanto è calcolabile, totale, e si ha:

se $x \in \mathcal{K}$, $\phi_{g(x)}(y) = 0$ per ogni y, quindi $g(x) \in \mathcal{Z}$.
se $x \notin \mathcal{K}$, $\phi_{g(x)}$ è indefinita su ogni y, quindi $g(x) \notin \mathcal{Z}$.

Allora $\mathcal{Z}$ non può essere ricorsivo, infatti: supponiamo di avere una macchina $\mathcal{M}_{\mathcal{Z}}$ che decide $\mathcal{Z}$, ossia che, avendo in input (la descrizione di) un algoritmo, determina se questo restituisce sempre 0. Allora, possiamo ricavare da questa una macchina $\mathcal{M}_{\mathcal{K}}$ che decide $\mathcal{K}$ nel modo seguente: avendo in input (la descrizione di) un algoritmo (x), per prima cosa eseguo l'algoritmo che costruisce $g(x)$. Passiamo ora a $\mathcal{M}_{\mathcal{Z}}$ (la descrizione del)l'algoritmo modificato $g(x)$, se $\mathcal{M}$ fosse in grado di decidere se $\phi_{g(x)}$ è la funzione costane 0 allora $\mathcal{M}_{\mathcal{K}}$ sarebbe in grado di decidere se ϕ_x è definita su x.

[7] Ovviamente altri esempi di riducibilità sono tutti gli esempi di riducibilità polinomiale visti nel corso di Analisi e Progettazione di Algoritmi.

Un punto importante da notare nell'esempio precedente è che l'algoritmo che calcola g effettua una *trasformazione di programmi*, ossia, prende in input un programma x e restituisce un nuovo programma $g(x)$ con il comportamento suddetto. Quindi, l'algoritmo di trasformazione *non esegue* il programma x. Inoltre, quando abbiamo detto che la costruzione dell'algoritmo modificato $g(x)$ può essere effettuata algoritmicamente a partire da x abbiamo usato implicitamente il *teorema del parametro* (*teorema s-m-n*). Vediamo di rendere esplicito il ragionamento effettuato.
Data una funzione $f\colon \mathbb{N}\times\mathbb{N} \to \mathbb{N}$, per esempio $f(x,y) = y^x$, possiamo fissare il valore del primo parametro x. Otteniamo in questo modo una funzione f_x con un solo parametro, per esempio, per $x = 2$ otteniamo $f_2(y) = y^2$. Il teorema del parametro garantisce che, se f è ricorsiva, quindi abbiamo un algoritmo per calcolarla, possiamo ottenere in modo algoritmico, a partire da ogni valore x del primo parametro, un algorimo per calcolare f_x. in modo algoritmico a

Teorema 3.3.10 [Teorema del parametro (forma semplice)] Sia $f\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ ricorsiva. Esiste una funzione ricorsiva totale g tale che per ogni x

$$\phi_{g(x)}(y) = f(x,y)$$

Prova Dato che f è ricorsiva, esiste un indice z tale che $f = \phi_z^2$. A partire da z, possiamo ricavare algoritmicamente la descrizione (nel formalismo Turing-completo considerato) di un algoritmo che prende in input due naturali x e y e calcola $f(x,y)$. In ogni formalismo Turing-completo, è possibile ricavare algoritmicamente da questa descrizione, per ogni x, la descrizione di un algoritmo con x fissato (è facile in un linguaggio di programmazione, più complicato con le TM ma possibile). Data questa descrizione, possiamo ricavare algoritmicamente l'indice $g(x)$. □
Il teorema del parametro corrisponde alla *specializzazione* del software. Il nome "teorema s-m-n" fa riferimento alla versione generalizzata del teorema (nella quale vengono fissati m parametri e ne restano n).
Usando esplicitamente il teorema del parametro possiamo definire g nel modo seguente: consideriamo $\psi\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ definita da:

$$\psi(x,y) = \begin{cases} 0 & \text{se } x \in \mathcal{K} \\ \text{indefinito} & \text{altrimenti} \end{cases}$$

La funzione ψ è ricorsiva. Allora, per il teorema del parametro, esiste una funzione ricorsiva totale g tale che, per ogni x, $\phi_{g(x)}(y) = \psi(x,y)$.
Vediamo un altro esempio di riduzione per provare che un problema non è decidibile.

Esempio 3.3.11 Determinare se due algoritmi calcolano la stessa funzione è un problema non decidibile. Formalmente, l'insieme $\mathcal{EQ} = \{\langle x,y\rangle \mid \phi_x = \phi_y\}$ non è ricorsivo. Possiamo provarlo con una riduzione da $\mathcal{Z}$ in questo insieme. Infatti, se sapessi risolvere il problema $\mathcal{EQ}$, potrei risolvere anche il problema $\mathcal{Z}$, in quanto basterebbe utilizzare l'algoritmo per $\mathcal{EQ}$ passando come input la coppia formata dall'input di $\mathcal{Z}$ e da un (indice di) algoritmo $\bar{x}$ che calcoli la funzione costante 0. In questo caso la funzione di riduzione g manda un indice in una coppia di indici ed è definita nel modo seguente: per ogni $x \in \mathbb{N}$, $g(x) = \langle x, \bar{x}\rangle$.

3.3.5 Teorema di Rice

Il teorema di Rice afferma che ogni proprietà "semantica" dei programmi (algoritmi) non è decidibile.
Possiamo formalizzare una proprietà dei programmi come un sottoinsieme di tutte le possibili TM (algoritmi), equivalentemente quindi come un sottoinsieme di $\mathbb{N}$.

Diciamo che una proprietà Π è *estensionale* se, per ogni $x, y \in \mathbb{N}$, se $x \in \Pi$ e $\phi_x = \phi_y$, allora $y \in \Pi$. In altri termini, si tratta di una proprietà relativa a "ciò che il programma fa" e non al programma stesso.
Come già osservato, esiste un numero infinito di TM equivalenti a una data.

Lemma 3.3.12 Per ogni $x, k \in \mathbb{N}$, esiste y tale che $\phi_x = \phi_y$ e $y > k$.

Esempio 3.3.13 Esempi di proprietà estensionali sono:

- $\emptyset$ e $\mathbb{N}$ (proprietà *banali*)
- dato un insieme di funzioni ricorsive F, $\{x \mid \phi_x \in F\}$
- come caso particolare del procedente, $\{x \mid \phi_x$ è la funzione identità$\}$
- $\{x \mid \phi_x(5) = 3\}$

Esempi di proprietà non estensionali sono:

- $\{x \mid x = 53\}$
- $\{x \mid \mathcal{M}_x$ termina in al più k passi$\}$

Teorema 3.3.14 Una proprietà estensionale Π è ricorsiva se e solo se è banale.

Prova L'implicazione $\Leftarrow$ è ovvia. Per l'implicazione $\Rightarrow$, consideriamo una proprietà Π, supponiamo che Π non sia banale, e mostriamo che non può essere ricorsiva attraverso una riduzione da $\mathcal{K} = \{x \mid \phi_x(x)\downarrow\}$ a Π.
Assumiamo, senza perdere in generalità[8], che $\{x \mid \phi_x$ totalmente indefinita$\} \subseteq \overline{\Pi}$, e scegliamo un elemento z di Π, che esiste sicuramente in quanto Π non vuoto.
L'input del problema $\mathcal{K}$ è (la descrizione di) un algoritmo x, e dobbiamo trasformarlo in una nuovo algoritmo $g(x)$ in modo tale che ϕ_x sia definita su x se e solo se $\phi_{g(x)} \in \Pi$.
È chiaro che questo si può ottenere costruendo l'algoritmo $g(x)$ nel modo seguente:

> input $y \rightarrow$ invoco $\mathcal{M}_x(x) \rightarrow$ se termina restituisco $\phi_z(y)$

Allora g è una funzione di riduzione da $\mathcal{K}$ in Π, in quanto è calcolabile, totale, e si ha:

> se $x \in \mathcal{K}$, $\phi_{g(x)}(y) = \phi_z(y)$ per ogni y, quindi $g(x) \in \Pi$.
> se $x \notin \mathcal{K}$, $\phi_{g(x)}$ indefinita su ogni y, quindi $g(x) \in \overline{\Pi}$

(si noti che per il secondo punto mi serve l'assunzione iniziale che $\{x \mid \phi_x$ totalmente indefinita$\} \subseteq \overline{\Pi}$).
Uso del teorema di Rice: possiamo provare che una certa proprietà dei programmi non è decidibile semplicemente osservando che è estensionale e non banale.

Esempio 3.3.15 Non possiamo decidere se un programma calcola una certa funzione (problema *della correttezza*), ossia, fissata una funzione ricorsiva ψ, l'insieme $\{x \mid \phi_x = \psi\}$ non è ricorsivo. Infatti, questo insieme chiaramente non è banale ed è estensionale.

[8] In caso contrario, basta nel ragionamento scambiare Π e $\overline{\Pi}$. In questo modo si ottiene una riduzione da $\mathcal{K}$ a $\overline{\Pi}$ e si ha la tesi per il teorema di Post.

3.4 Altri modelli di calcolo

3.4.1 Funzioni μ-ricorsive

Vediamo ora come il formalismo delle primitive ricorsive introdotto precedentemente può essere esteso in modo da ottenere un formalismo che si può dimostrare equivalente alle macchine di Turing, suffragando quindi la tesi di Church.

Operatore μ Data $f\colon \mathbb{N}^{n+1} \to \mathbb{N}$, f totale, definiamo la seguente funzione $g\colon \mathbb{N}^n \to \mathbb{N}$:

$$g(x_1, \ldots, x_n) = \mu_y^0(f(x_1, \ldots, x_n, y)) = \begin{cases} \min\{y \mid f(x_1, \ldots, x_n, y) = 0\} & \text{se tale minimo esiste} \\ \text{indefinito} & \text{altrimenti} \end{cases}$$

e diciamo che g è definita per *minimizzazione* o μ*-ricorsione* da f.

Def. 3.4.1 La classe delle funzioni μ-ricorsive è la più piccola classe di funzioni chiusa rispetto alle operazioni di $\mathcal{PR}$ e alla μ-ricorsione (applicata a funzioni totali).

Chiaramente una funzione definita per μ-ricorsione può essere, per definizione, parziale (per esempio, se f è definita come $f(x, y) = 1$). Si noti che l'ipotesi di totalità nella definizione precedente è necessaria, altrimenti non avremmo una procedura effettiva per calcolare la funzione (vedi sotto).
Per esempio, possiamo definire per μ-ricorsione il logaritmo (intero):

$$\lfloor \log_a(x) \rfloor = \mu_y^0(\mathit{leq}(a^{y+1}, x))$$

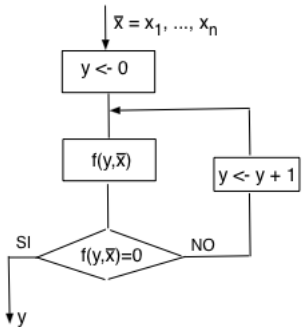
dove $\mathit{leq}(x, y)$ vale 1 se $x \leq y$, 0 altrimenti, ed è primitiva ricorsiva (provarlo per esercizio), come anche a^y. Per esempio, per ottenere $\lfloor \log_2(7) \rfloor$, calcoliamo successivamente:

$\mathit{leq}(2^{0+1}, 7) = 1$
$\mathit{leq}(2^{1+1}, 7) = 1$
$\mathit{leq}(2^{2+1}, 7) = 0 \Rightarrow$ il risultato è 2

Teorema 3.4.2 f è ricorsiva se e solo se f è μ-ricorsiva.

L'implicazione $\Leftarrow$ corrisponde a provare che:

f ricorsiva totale $\Rightarrow$ g definita per μ-ricorsione da f ricorsiva (parziale)
Prova Si usa la tesi di Church osservando che f è calcolata dall'algoritmo sotto schematizzato.



Si noti che se f non è totale il ragionamento sopra non funziona, perché calcolo di $f(y, \bar{x})$ può non terminare e quindi non si arriva al test successivo. In effetti si può provare che la classe delle funzioni ricorsive non è chiusa rispetto all'operatore μ.
Per l'implicazione $\Rightarrow$, si prova più precisamente il seguente risultato.

Teorema 3.4.3 [Forma normale di Kleene] Per ogni $k \in \mathbb{N}$ esistono due funzioni primitive ricorsive $p\colon \mathbb{N} \to \mathbb{N}$ e $t_k\colon \mathbb{N}^{k+2} \to \mathbb{N}$ tali che

$$\text{per ogni } z \in \mathbb{N} \quad \phi_z^k(x_1, \ldots, x_k) = p(\mu_y^0(t_k(z, x_1, \ldots, x_k, y))$$

Questo risultato mette in evidenza che ogni funzione calcolabile può essere ottenuta tramite successive applicazioni di processi iterativi limitati a priori e quindi sempre terminanti (ricorsione primitiva), purché si aggiunga *una sola* operazione di iterazione eventualmente non terminante.

3.4.2 Varianti delle macchine di Turing

In questa sezione studieremo varianti del modello di TM visto precedentemente. In particolare siamo interessati a versioni non deterministiche delle TM e alla loro codifica attraverso macchine deterministiche. Per semplificare la codifica di una macchina non deterministica introduciamo prima una variante con nastro multiplo.

Multitape TM Una *macchina di Turing multinastro* (*multitape TM*) è una TM con k nastri e k testine coordinate da un singolo programma di controllo. Una descrizione istantanea è formata quindi da k configurazioni di TM con un solo nastro. La testina di ogni nastro si muove indipendentemente dalle altre, ma si trovano tutte nello stesso stato. Per esempio, nel caso di due nastri, un esempio di configurazione è $\langle XqY, XqYXY\rangle$. La funzione di transizione è quindi una funzione $\delta\colon\ Q \times \Sigma^k \to\ Q \times (\Sigma \times \{\mathrm{L}, \mathrm{R}, \mathrm{N}\})^k$. Per esempio con due nastri la transizione

$$\delta(q, \langle X, Y\rangle) = \langle q', \langle X', \mathrm{L}\rangle, \langle Y', \mathrm{R}\rangle\rangle$$

se la testina del primo nastro è su X e quella del secondo su Y, la sposta a sinistra sul primo e a destra sul secondo.
È immediato generalizzare al caso delle macchine multinastro le nozioni di computazione, accettazione, riconoscimento e funzione calcolata, scegliendo da quale nastro leggere l'input e da quale recuperare l'output. Lasciamo come esercizio le definizioni formali.
Chiaramente le macchine multitape hanno almeno l'espressività delle TM con un solo nastro. Vale anche il viceversa, ossia possiamo simulare una macchina multitape con una TM con un solo nastro? In base alla tesi di Church-Turing, ci aspettiamo che la risposta sia positiva, in quanto pensiamo che le TM possano codificare qualsiasi altro formalismo per esprimere algoritmi.
Data una macchina multinastro $\mathcal{M}$ con simboli in Σ e stati in Q, vediamo quindi (a grandi linee) come costruire una TM che ne simuli l'esecuzione. L'idea è la seguente: in una configurazione della macchina emulatrice, come, per esempio, la seguente:

$$q_s \begin{bmatrix} \downarrow X \\ X \\ Y \end{bmatrix} \begin{bmatrix} X \\ B \\ Y \end{bmatrix} \begin{bmatrix} X \\ \downarrow X \\ X \end{bmatrix} \ldots \begin{bmatrix} X \\ Z \\ \downarrow Y \end{bmatrix}$$

ogni simbolo sul nastro corrisponde a k (tre nell'esempio) simboli in Σ, come rappresentato dalle colonne racchiuse tra parentesi quadre nella figura. Inoltre, per esempio nel primo elemento della prima colonna in figura, un simbolo in Γ può essere accoppiato a un simbolo speciale $\downarrow$ (sempre lo stesso) a indicare che su quel nastro la testina è posizionata su quel simbolo. In questo modo, rappresentiamo su un solo nastro simultaneamente diversi nastri (righe orizzontali nel disegno, che chiameremo *track*). Le celle dei nastri simulati possono contenere blank (diversi dai blank della macchina che esegue la simulazione).
Abbiamo bisogno di codificare in un singolo alfabeto tutte le possibili configurazioni delle celle. Se i simboli in Γ sono n, ci servono quindi $2 * n$ simboli per rappresentare celle con simboli in Γ puntati o meno dalla testina. Possiamo quindi usare (un alfabeto che rappresenti) le tuple con k di tali simboli, quindi di cardinalità $(2 * n)^k$.
La macchina opera analizzando un nastro alla volta. Prima cerca la posizione della testina nel primo nastro, controlla quale regola applicare e memorizza l'azione da compiere nel suo stato. Poi ripete le stesse operazioni sugli altri nastri. Essendo i nastri k, è possibile memorizzare con un insieme finito di stati quanti nastri sono stati già analizzati e quali regole devono essere applicate su ogni nastro. In una seconda fase si applicano le modifiche e gli spostamenti nastro per nastro.

TM non deterministica Vediamo ora una variante delle TM che ammette il non determinismo. Una TM (con un solo nastro) non deterministica ha le stesse componenti di una TM deterministica. Tuttavia le mosse possibili sono definite tramite una funzione di transizione $\delta\colon\ Q \times \Gamma \to \wp(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\})$ dove ricordiamo che $\wp(A)$ denota l'insieme dei sottoinsiemi di A. In altre parole, come già visto per le versioni non deterministiche di automi a stati finiti e automi a pila, una configurazione può avere diversi possibili successori che danno luogo a diverse possibili computazioni. Si avrà quindi la regola:

$$\langle \alpha, q, X\beta \rangle \to \langle \alpha\, Y, q', \beta \rangle \quad \text{se } \langle q',\, Y, \mathrm{R} \rangle \in \delta(q, X)$$

e analogamente negli altri casi. In generale quindi a partire da una configurazione iniziale avremo un insieme di possibili computazioni, più precisamente un albero di computazioni con ampiezza finita ma rami potenzialmente infiniti. Come per gli automi non deterministici visti precedentemente, l'accettazione di un input u in una macchina non deterministica richiede *l'esistenza* di almeno una computazione accettante (tra tutte quelle possibili) su u.
Il non determinismo aumenta il potere espressivo delle TM? La risposta è no. Infatti, data una TM non deterministica $\mathcal{M}$ che accetta un linguaggio, possiamo costruire una macchina deterministica che accetta lo stesso linguaggio. Intuitivamente, occorre costruire una TM deterministica $\mathcal{M}'$ in grado di esplorare l'albero delle computazioni di $\mathcal{M}$. Abbiamo bisogno quindi di:

- enumerare le possibili computazioni di $\mathcal{M}$ a partire da una stringa in input.
- scelta una computazione, simularla
- controllare se la computazione è accettante e, nel caso, accettare la stringa in input, altrimenti proseguire con la successiva.

Tuttavia, non tutte le visite dell'albero di computazioni di $\mathcal{M}$ risultano adeguate. Infatti, con una visita depth-first potremmo trovare una computazione infinita e quindi la simulazione potrebbe a sua volta non terminare. Usiamo quindi una visita breadth-first dell'albero delle computazioni. Inoltre dobbiamo rendere la visita deterministica, ossia numerare in un certo ordine tutti i cammini livello per livello. Partendo dal primo livello numeriamo i successori di una configurazione in maniera progressiva in base alle possibili scelte della relazione di transizione come in Figura 3.1. Le scelte a partire da uno stato q e da un simbolo X sono sempre finite e quindi si possono codificare con numeri d'ordine. Un cammino dalla radice $(q_0 u)$

Figura 3.1: Enumerazione delle scelte

a una certa configurazione è quindi univocamente determinato da una sequenza di numeri d'ordine. Per facilitare il compito usiamo una TM con tre nastri. Il primo nastro contiene l'input u. Il secondo è un nastro di lavoro. Il terzo contiene le sequenze che rappresentano i cammini (computazioni) che vogliamo simulare. La simulazione ha bisogno quindi di un sottoprogramma che, dato un livello l, enumeri tutte le possibili sequenze di lunghezza l composte da numeri in $[1, max]$ dove max è il numero massimo di scelte possibili in una certa configurazione (determinato da δ).
L'algoritmo procede nel seguente modo. Poniamo inizialmente $l = 0$.

1. Se $l = 0$ oppure abbiamo esplorato tutte le sequenze di lunghezza l, si incrementa l di uno (cioè si aumenta di uno la profondità della visita). Altrimenti si genera la prossima sequenza σ di lunghezza l nel terzo nastro.

2. Si copia l'input nel secondo nastro.

3. Si simula la macchina $\mathcal{M}$ seguendo le scelte specificate in σ controllando che siano possibili, cioè che per ogni scelta ci sia una tranzione eseguibile. Poichè δ ha una rappresentazione finita applicare una per volta le mosse specificate da σ è sicuramente possibile tramite un algoritmo deterministico. Si controlla se la computazione simulata è accettante e in tal caso si accetta e l'algoritmo termina. Altrimenti, si cancella il nastro di lavoro e si torna a (1).

L'algoritmo accetta u se e solo se esiste una computazione di $\mathcal{M}$ che accetta u.

Capitolo 4

Appendice

4.1 Relazioni e funzioni

Indichiamo con $\mathbb{N}$, $\mathbb{Z}$ e $\mathbb{B}$ rispettivamente gli insiemi dei numeri naturali, dei numeri interi e dei valori di verità. Dati due insiemi A e B, indichiamo con $A \times B$ il *prodotto cartesiano* di A e B, cioè l'insieme delle coppie ordinate $\langle a, b\rangle$ dove $a \in A$ e $b \in B$.

Def. 4.1.1 [Relazioni e funzioni] Dati due insiemi A e B, una *relazione* R da A in B è un sottoinsieme di $A \times B$; una *relazione su* A è una relazione da A in A. Se $\langle a, b\rangle \in R$ si scrive anche aRb.
Una *funzione (totale)* da A in B è una relazione f da A in B che gode delle seguenti proprietà:

Univocità per ogni $a \in A$ e $b, b' \in B$, se $\langle a, b\rangle \in f$ e $\langle a, b'\rangle \in f$, allora $b = b'$

Totalità per ogni $a \in A$, esiste $b \in B$ tale che $\langle a, b\rangle \in f$.

Se f soddisfa solo la prima condizione (univocità) allora f si dice *funzione parziale*.

La proprietà di univocità assicura che per una funzione (parziale o totale) f da A in B, per ogni elemento $a \in A$ ci sia al più un elemento (esattamente uno per funzioni totali) $b \in B$ tale che $\langle a, b\rangle \in f$; per questa ragione, tale elemento b viene indicato con $f(a)$.
Diversamente da quanto si fa comunemente in matematica, in queste note, quando non diversamente indicato, assumiamo che le funzioni siano parziali. Scriveremo $f(x)\downarrow$ per indicare che la funzione f è definita su x, $f(x)\uparrow$ per indicare che non è definita su x.

Def. 4.1.2 [Predicato] Dato un insieme A, un *predicato* su A è una funzione totale da A in $\mathbb{B}$.

Def. 4.1.3 [Relazione di equivalenza] Una *relazione di equivalenza* su A è una relazione $\sim$ su A che sia:

- *riflessiva*, ossia $a \sim a$ per ogni $a \in A$,
- *simmetrica*, ossia se $a \sim a'$, allora $a' \sim a$, per ogni $a, a' \in A$,
- *transitiva*, ossia se $a \sim a'$ e $a' \sim a''$, allora $a \sim a''$, per ogni $a, a', a'' \in A$.

Per ogni $a \in A$, la sua *classe di equivalenza* $[a]_\sim$ è l'insieme degli elementi di A in relazione con a. L'insieme *quoziente* $A/\sim$ è l'insieme i cui elementi sono le classi di equivalenza.

Dato un insieme A e una relazione di equivalenza $\sim$ su A, si può definite la *proiezione sul quoziente* come la funzione $\pi\colon A \to A/\sim$ data da $\pi(a) = [a]_\sim$ per ogni $a \in A$. È facile vedere che questa funzione è surgettiva. Una semplice proprietà dell'insieme quoziente è che questo forma una *partizione* dell'insieme A, cioè valgono le seguenti proprietà:

- le classi di equivalenza sono a due a due disgiunte, cioè per ogni $a, b \in A$, se $[a]_\sim \neq [b]_\sim$, allora $[a]_\sim \cap [b]_\sim = \emptyset$
- le classi di equivalenza ricoprono A, cioè $A = \bigcup_{a \in A} [a]_\sim$.

Def. 4.1.4 [Chiusura riflessiva e transitiva] Se R è una relazione su A, la *chiusura transitiva* di R è la relazione R^+ su A definita come segue:

per ogni $a, a' \in A$, aR^+a' sse esistono elementi $a_0, \ldots, a_n \in A$ tali che $a = a_0$, a_0Ra_1, a_1Ra_2, $\ldots$, $a_{n-1}Ra_n = a'$

La *chiusura riflessiva e transitiva* di R è la relazione $R^\star$ su A definita da: $R^\star = R^+ \cup \{\langle a, a\rangle \mid a \in A\}$. Equivalentemente, $R^\star$ può essere definita induttivamente, vedi Sezione **??**, come segue[1]:

- aRa per ogni $a \in A$,
- se aRa', allora $aR^\star a'$,
- se $aR^\star a'$ e $a'R^\star a''$, allora $aR^\star a''$

e analogamente R^+ escludendo la prima metaregola.

4.2 Cardinalità

Def. 4.2.1 [Proprietà delle funzioni] Una funzione f da A in B si dice

iniettiva se, per ogni $x, x' \in A$, se $f(x) = f(x')$, allora $x = x'$

surgettiva se, per ogni $y \in B$, esiste $x \in A$ tale che $y = f(x)$

biunivoca/bigettiva se è sia iniettiva che surgettiva

invertibile se esiste una funzione g da B in A tale che $f \circ g = id_B$ e $g \circ f = id_A$, dove $id_A\colon A \to A$ è l'*identità* su A, definita da $id_A(x) = x$ per ogni $x \in A$.

Prop. 4.2.2 Una funzione f da A a B è biunivoca se e solo se è invertibile.

Def. 4.2.3 [Insiemi equicardinali] Due insiemi A e B sono *equicardinali*, e si scrive $A \simeq B$, se esiste una funzione biunivoca da A in B.

Intuitivamente, si può pensare che due insiemi equicardinali abbiano lo stesso "numero" di elementi. Due insiemi equicardinali si dicono anche *isomorfi*.

Def. 4.2.4 Un insieme A si dice *numerabile* se esiste una funzione iniettiva da A in $\mathbb{N}$.

[1] Ossia, come la più piccola relazione riflessiva e transitiva che contiene R.

Consideriamo un esempio importante di insiemi isomorfi. Indichiamo con $\wp(X)$ l'insieme dei sottoinsiemi di X e con $\mathbb{B}^X$ l'insieme dei predicati su X, e indichiamo i valori di verità con 1 (vero) e 0 (falso)[2]. Per ogni sottoinsieme $A \subseteq X$ possiamo definire un predicato $\chi_A \in \mathbb{B}^X$, detto *funzione caratteristica di* A, come segue:

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

Viceversa, per ogni predicato $p \in \mathbb{B}^X$, possiamo definire un sottoinsieme $A = p^{-1}(1) = \{x \in X \mid p(x) = 1\} \subseteq X$. Queste due costruzioni danno due funzioni $\phi_X \colon \wp(X) \to \mathbb{B}^X$ e $\psi_X \colon \mathbb{B}^X \to \wp(X)$ una inversa dell'altra, per cui vale la seguente proposizione.

Prop. 4.2.5 Per ogni insieme X, $\wp(X)$ e $\mathbb{B}^X$ sono isomorfi.

Concludiamo dimostrando un risultato fondamentale, dovuto a Cantor, sulla cardinalità dell'insieme $\wp(A)$.

Teorema 4.2.6 [Cantor] Per ogni insieme A, non esiste una funzione surgettiva $f \colon A \to \wp(A)$.

Prova Supponiamo esista una funzione surgettiva $f \colon A \to \wp(A)$. Consideriamo il sottoinsieme $B = \{x \in A \mid x \notin f(x)\} \in \wp(A)$. Essendo f surgettiva, esiste un $x_B \in A$ tale che $f(x_B) = B$, distinguiamo quindi due casi:

- se $x_B \in B$, allora, per definizione di B, $x_B \notin f(x_B) = B$ che è assurdo
- se $x_B \notin B$, allora, per definizione di B, $x_B \in f(x_B) = B$, che è assurdo.

Questo prova il teorema.

Corollario 4.2.7 Per ogni insieme A, non esiste una funzione biunivoca $f \colon A \to \wp(A)$.

Prova Se $f \colon A \to \wp(A)$ è biunivoca, allora è surgettiva, ma questo è impossibile per il teorema di Cantor.
Intuitivamente, questo teorema asserisce che in $\wp(A)$ ci sono "più elementi" che in A, perché, comunque si provi ad assegnare a ciascun elemento di A un elemento di $\wp(A)$ (cioè comunque si prenda una funzione $f \colon A \to \wp(A)$), ci sono degli elementi di $\wp(A)$ che rimangono senza un rappresentante (cioè f non è surgettiva).

4.3 Famiglie

Def. 4.3.1 [Famiglia di insiemi] Una *famiglia di insiemi* (o semplicemente *famiglia*) (indiciata su S) è una funzione totale che associa ad ogni $s \in S$ un insieme. Se A è una famiglia di insiemi indiciata su S, per ogni $s \in S$ l'insieme associato ad s si indica A_s.

Si noti la differenza tra una famiglia ed un insieme; ad esempio la famiglia A indiciata su a, b, c definita da $A_a = \mathbb{Z}$, $A_b = \mathbb{B}$, $A_c = \mathbb{Z}$ è diversa dall'insieme $\{A_a, A_b, A_c\} = \{\mathbb{Z}, \mathbb{B}\}$.
Le usuali operazioni sugli insiemi (ad esempio unione, intersezione, differenza) si estendono alle famiglie, componente per componente (è necessario che le famiglie coinvolte siano indiciate tutte sullo *stesso* insiemi di indici).
In modo esattamente analogo ad una famiglia di insiemi si definisce una famiglia indiciata su S di funzioni.

[2]Sono i simboli per i valori di verità generalmente usati nella definizione di funzione caratteristica; altrove in queste note usiamo T, F.

4.4 Definizioni induttive

In matematica e informatica si usano molto spesso definizioni *ricorsive*, cioè definizioni in cui l'espressione che definisce l'oggetto fa riferimento all'oggetto stesso. Un tipo di definizioni ricorsive alle quali è possibile attribuire in modo semplice un significato preciso sono le definizioni cosiddette *induttive*, illustrate nel seguito. L'idea base è quella di definire un insieme come il più piccolo tra tutti gli insiemi X che verificano delle condizioni di "chiusura" del tipo "se certi elementi appartengono a X, allora anche un certo altro elemento deve appartenere a X".
Sia nel seguito U un insieme fissato detto *universo*.

Def. 4.4.1 [Definizione induttiva] Una *definizione induttiva* R è un insieme di *regole* $\frac{Pr}{c}$, dove $Pr \subseteq U$ è l'insieme delle *premesse* e $c \in U$ è la *conseguenza* della regola. Un insieme $X \subseteq U$ è *chiuso rispetto a una regola* $\frac{Pr}{c}$ se $Pr \subseteq X$ implica $c \in X$; è chiuso rispetto a R se è chiuso rispetto a ogni regola di R. L'insieme $I(R)$ *definito induttivamente da* R è il più piccolo[3] insieme chiuso rispetto a R.

Una definizione induttiva viene in genere data in modo più leggibile utilizzando un qualche metalinguaggio (per esempio l'usuale metalinguaggio matematico). Spesso consiste di un insieme di *metaregole*, contenenti *metavariabili* da istanziare con elementi dell'universo, in modo che ogni metaregola descriva un insieme anche infinito di regole. Si dice che le (meta)regole con un insieme vuoto di premesse costituiscono la *base* mentre le altre (meta)regole costituiscono il *passo induttivo* della definizione induttiva.

Esempio 4.4.2 L'insieme dei numeri pari è definito nel modo seguente:

- 0 è un numero pari,
- se n è un numero pari, allora $n + 2$ è un numero pari.

Si può vedere più chiaramente che si tratta di una definizione induttiva riscrivendola nello stile a *regole di inferenza*, in cui le premesse vengono scritte sopra la linea di frazione, la conseguenza sotto e le eventuali condizioni a lato. Si ha:

$$\frac{}{0} \qquad \frac{n}{n+2}$$

In questo caso, l'insieme universo è $\mathbb{N}$, e la definizione induttiva è, formalmente, $\{\frac{\emptyset}{0}\} \cup \{\frac{n}{n+2} \mid n \in \mathbb{N}\}$.

4.5 Principio di induzione

Prop. 4.5.1 [Principio di induzione (forma generale)] Sia R una definizione induttiva, U un insieme tale che $I(R) \subseteq U$, e P un predicato su U, cioè una funzione $P\colon U \to \{\mathrm{T}, \mathrm{F}\}$.
Se: per ogni $\frac{Pr}{c} \in R$ $(P(x) = \mathrm{T}$ per ogni $x \in Pr)$ implica $P(c) = \mathrm{T}$ allora: $P(x) = \mathrm{T}$ per ogni $x \in I(R)$.

[3] Nel senso dell'inclusione insiemistica, ossia l'intersezione di tutti gli insiemi chiusi che si vede facilmente essere a sua volta un insieme chiuso. Si noti anche che vi è sempre almeno un insieme chiuso, l'universo U, quindi la definizione non è vuota. Infine, si noti che è sempre possibile considerare come universo l'insieme dagli elementi delle coppie di R, o anche solo gli elementi conseguenza, quindi in realtà non è necessario fissare un universo a priori.

Prova Poniamo $C = \{x | P(x) = \text{T}\}$ e osserviamo che la condizione in giallo può essere scritta nella forma equivalente:

$$Pr \subseteq C \text{ implica } c \in C.$$

ossia corrisponde a richiedere che C sia chiuso rispetto a R. Quindi, per definizione $I(R) \subseteq C$, ossia $P(x) = \text{T}$ per ogni $x \in I(R)$. □

Si noti che nel caso di una regola con insieme di premesse vuoto la condizione in giallo equivale a $P(c) = \text{T}$. Ciò suggerisce l'usuale formulazione del principio di induzione spezzata in *base* e *passo induttivo*, dove la base corrisponde alle (meta)regole senza premesse.

Un caso particolare del principio di induzione dato qui in forma generale è il principio di induzione aritmetica.

Prop. 4.5.2 [Principio di induzione aritmetica] Sia P un predicato sui numeri naturali tale che

1. $P(0) = \text{T}$;
2. per ogni $n \in \mathbb{N}$, $P(n) = \text{T}$ implica $P(n+1) = \text{T}$.

Allora $P(n) = \text{T}$ per ogni $n \in \mathbb{N}$.

Prova $\mathbb{N}$ può essere visto come il sottoinsieme dei naturali definito induttivamente da

- $0 \in \mathbb{N}$;
- se $n \in \mathbb{N}$ allora $n + 1 \in \mathbb{N}$.

Quindi la proposizione è un caso particolare del principio di induzione. □

Prop. 4.5.3 [Principio di induzione aritmetica completa (o forte)] Sia P un predicato sui numeri naturali tale che

Base vale $P(0)$

Passo induttivo se vale $P(m)$ per ogni $m < n$ allora vale $P(n)$,

allora $P(n)$ vale per ogni $n \in \mathbb{N}$.

Prova $\mathbb{N}$ può essere visto come l'insieme definito induttivamente da

- $0 \in \mathbb{N}$
- se $\{m \mid m < n\} \subseteq \mathbb{N}$ allora $n \in \mathbb{N}$.

Quindi la proposizione è un caso particolare del principio di induzione. □

4.6 Definizioni induttive multiple

Sono spesso utili in informatica le definizioni induttive di famiglie di insiemi (Def.4.3.1), dette anche definizioni *induttive multiple* o *mutuamente induttive*, con l'associato principio di induzione multipla. Si tratta di una facile generalizzazione delle definizioni e dei risultati già dati.
Sia U una famiglia (indiciata su S) fissata detta *universo*.

Def. 4.6.1 [Definizioni induttive multiple] Una *definizione induttiva multipla* è un insieme di *regole* $\frac{Pr}{c:\bar{s}}$, dove $Pr \subseteq U$ (cioè $Pr_s \subseteq U_s$ per ogni $s \in S$) è l'insieme delle *premesse*, e $c : \bar{s}$, dove $\bar{s} \in S$ e $c \in U_{\bar{s}}$, è la *conseguenza* della regola. Una famiglia $X \subseteq U$ è *chiusa rispetto a una regola* $\frac{Pr}{c:\bar{s}}$ se $Pr_s \subseteq X_s$ per ogni $s \in S$ implica $c \in X_{\bar{s}}$; è chiusa rispetto a R se è chiusa rispetto a ogni regola di R. La famiglia $I(R)$ *definita induttivamente da* R è la più piccola[4] famiglia chiusa rispetto a R.

Analogamente a quanto si è visto per le definizioni induttive non multiple, i sistemi induttivi sono in generale presentati dando un insieme di regole di inferenza o metaregole. Il principio di induzione si estende al caso multiplo.

Prop. 4.6.2 [Principio di induzione multipla] Sia R una definizione induttiva multipla, U una famiglia di insiemi (indiciata su S) tale che $I(R) \subseteq U$, e P una famiglia di predicati su U.
Se: $\boxed{\text{per ogni } \frac{Pr}{c:\bar{s}} \in R \qquad (P_s(x) = \text{T per ogni } x \in Pr_s, \text{ per ogni } s \in S) \text{ implica } P_{\bar{s}}(c) = \text{T}}$
allora: $P_s(x) = \text{T}$ per ogni $x \in I_s(R)$, per ogni $s \in S$.

Prova Analoga a quella del caso non multiplo. □
Anche qui l'ipotesi del principio di induzione può essere utilmente spezzata in "base" e "passo induttivo" in corrispondenza dell'analoga suddivisione delle (meta)regole. In conclusione nel caso multiplo si ragiona e procede come nel caso non multiplo, eccetto per il fatto che si ha a che fare con una famiglia di insiemi e che occorre quindi tener conto dell'appartenenza di un elemento a un particolare insieme della famiglia.

[4] Nel senso dell'inclusione di famiglie, ossia l'intersezione di tutte le famiglie chiuse.