di Andrea de Prisco

Seconda puntata

Teoria della computabilità

Algoritmi e Macchine di Turing

Forse qualcuno avrà già sentito parlare di Turing, del suo test e della sua macchina. Qualcun altro saprà che quest'ultima è stata inventata dallo stesso negli anni trenta, quando i calcolatori non solo non esistevano ma, pur desiderandoli ardentemente, non si immaginava nemmeno come realizzarli. In questo articolo ve ne parleremo brevemente, non senza anticiparvi che a tutt'oggi non esiste nessun altro calcolatore che superi le capacità di calcolo della suddetta macchina, tant'è che dire che una funzione è (in generale) calcolabile è sinonimo di «calcolabile con una macchina di Turing» o più semplicemente «Turing-calcolabile».

Definizione di Algoritmo

Per studiare adeguatamente la calcolabilità delle funzioni, è necessario stabilire, prima di muovere qualsiasi passo, cosa si intende per algoritmo e cosa per agente di calcolo. Empiricamente parlando, un algoritmo è un procedimento, in qualche modo preciso, che descrive una serie di operazione da compiere. L'agente si occuperà, una volta fornitogli l'algoritmo di «eseguirlo».

Il formalismo della Macchina di Turing fa capo ad alcuni requisiti di definizione di algoritmo ormai riconosciuti universali nell'ambito della calcolabilità non probabilistica. In tali requisiti, come vedremo, si fa riferimento anche a caratteristiche dell'agente di calcolo adoperato (ripetiamo: le due cose sono sempre strettamente legate).

Innanzitutto l'algoritmo, descritto da un programma, deve essere di lunghezza finita e il calcolo deve avvenire per passi discreti. Ovvero il numero di istruzioni di cui esso è composto può essere quanto grande vogliamo ma sempre un numero naturale (ricordiamo che ai numeri naturali non appartengono oggetti loro stessi infiniti, pur essendo questi in numero infinito). Per quanto riguarda l'agente di calcolo e in particolare il set di istruzioni che esso può eseguire, occorre che queste siano in numero finito e che la loro complessità non sia infinita. In altre parole, iniziata una singola istruzione in un tempo non infinito questa deve essere completata.

L'intrinseca finitezza di un programma e delle sue istruzioni, non implica altrettanto circa il numero di passi (in un certo senso «iterazioni») necessari affinché il programma stesso arrivi a «conclusione». Cioè, teoricamente parlando (quello che stiamo facendo da due mesi), un programma descrivente un determinato algoritmo, il quale calcola una qualsiasi funzione può benissimo richiedere un numero illimitato di passi o addirittura infinito.

Si noti che, sebbene la differenza è molto sottile, illimitato e infinito non sono la stessa cosa: per tornare come esempio ai numeri naturali, se è vero che non esiste un numero infinito appartenente a questi, possiamo affermare che esistono numeri naturali composti da un numero illimitato di cifre ovvero che non esiste un limite finito al numero di cifre di cui può essere composto un numero naturale (la non esistenza di un limite finito non autorizza a pensare a qualcosa di infinito). Nel caso dei nostri algoritmi, in un certo senso si taglia la testa al toro affermando che un algoritmo può anche non terminare mai, continuando a calcolare infinitamente. Per terminazione si intende, qualora non fosse chiaro dal contesto, che il programma in questione «sputa» fuori il suo risultato e si arresta.

Chiunque arrivi a questo punto di teoria si chiederà certamente a cosa servano algoritmi che non terminano mai. O meglio, tutti direbbero: perché occuparci anche di questi se, tangibilmente parlando, non sappiamo cosa farcene?

La risposta non è immediata. Di riflesso però si dimostra che se adoperiamo formalismi in grado di calcolare solo programmi che terminano, non solo perderemmo tutti i programmi che non terminano mai (di questo poco ci importa) ma, sottolineamo è dimostrato e lo dimostreremo, perderemo anche algoritmi che col formalismo precedente (terminazione e non) ci avrebbero fornito prima o poi un risultato.

Come dire che non è possibile ripulire un formalismo dai suoi algoritmi che non terminano mai senza portar via anche algoritmi che sarebbero terminati. Del resto questa scrittura non

è poi tanto «araba»: in qualsiasi Basic (dei miei stivali, ndr) esiste un programma tipo «10 GOTO 10», è possibile e non termina mai. Se facessimo un Basic col quale non è possibile scrivere programmi che «vanno in loop» non avremmo fatto qualcosa di utile ma un vero e proprio disastro: usando tale linguaggio di programmazione prima o poi cominceremo a imprecare perché non riusciamo a fare quella determinata cosa che vorremmo: tutto qui.

Memoria illimitata ed altro

Per non limitare la potenzialità di calcolo di un formalismo, occorre che non siano posti limiti alla dimensione dei dati in ingresso così come per la quantità di memoria necessaria al calcolo. Ovvero un calcolatore calcola la somma di due numeri naturali se ha l'opportunità di ricevere in ingresso due numeri qualsiasi su cui calcolare la somma. Se limito la dimensione dei dati in ingresso certamente non riuscirò a fare neanche un calcolatore che somma due numeri qualsiasi. Figuriamoci qualcosa di più.

Per la memoria illimitata basta pensare a un qualsiasi algoritmo che per raggiungere il risultato ha bisogno di immagazzinare una quantità di risultati parziali proporzionale ai dati in ingresso (es. la moltiplicazione tra due numeri). Se i dati in ingresso non sono in alcun modo limitati per dimensione, segue che anche la memoria usata per il calcolo deve essere illimitata.

E qui comincia a trasparire fortemente l'alto contenuto teorico di quanto stiamo trattando: è ovvio che un calcolatore siffatto non esiste e mai esisterà (nemmeno usando per memoria centrale tutti gli atomi di cui è composto l'universo, limitati); ciò significa solo che a causa di questo fatto non si riuscirà mai a costruire un calcolatore ideale. Ma questo nella teoria poco importa. Il bello è che anche riuscendo a realizzare quanto richiesto dai requisiti di ogni definizione di algoritmo sopra dati, si dimostra che alcune funzioni non sono calcolabili. Di altre non si sa nulla, di altre ancora, apparentemente non calcolabili risultano essere calcolabili da una macchina di Turing.

La Macchina di Turing: descrizione generale

E veniamo a questo benedetto calcolatore ideato da Turing negli anni trenta, che tanto ha fatto parlare il mondo di allora, come quello di oggi. Innanzitutto, la macchina di Turing (per brevità d'ora in poi la indicheremo anche col suo acronimo MDT) non è fisicamente realizzabile per il solito motivo della memoria illimitata. Essa quindi non va intesa come una vera e propria macchina ma come un modello matematico di un oggetto capace di calcolare. Se però dimentichiamo per un attimo la memoria, la MDT diventa un oggetto, non solo tangibile, ma realizzabile con pezzi elettromeccanici di fortuna come dei ricambi di un registratore a bobina e una manciata di componenti, allo stato solido di vario genere. Infatti, una MDT è composta essenzialmente da tre parti: un nastro magnetico, una testina di registrazione/riproduzione e una parte di controllo (figura 1).

Il nastro, suddiviso in celle ed usato come memoria di lettura e scrittura, ha lunghezza illimitata e viene utilizzato sia per leggere i dati in ingresso (qualcuno provvederà a inciderlo, prima di fare partire il tutto), sia per i calcoli intermedi, sia per scrivere i risultati prima di terminare l'elaborazione.

Per definizione di MDT, il nastro prima di una computazione è interamente blank tranne un insieme finito (volendo, anche illimitato, ma non infinito) di celle. In queste, come già detto, sono incisi i dati del programma.

La testina di lettura-scrittura, come è facile prevedere, legge e scrive sulle celle del nastro magnetico che, proprio sullo stile di un registratore, scorre davanti a questa.

Infine, la parte di controllo, serve per elaborare quanto è inciso sul nastro dando ordini sia alla testina che al meccanismo di scorrimento del nastro.

A questo punto dovrebbe essere ben chiara la semplicità di tutto l'apparato: ripetiamo, l'unico problema è il nastro illimitato, se no l'avrebbero realizzata. Anche perché a tutt'oggi non è stato ancora trovato un formalismo capace di calcolare più cose della MDT: equipotenti sì, ma più potenti no.

Giusto per chiarire subito una cosa, vogliamo aggiungere che anche un VIC-20 con memoria infinita sarebbe equipotente alla MDT: di questa se ne parla con tanta ammirazione proprio per la sua semplicità e per il fatto di essere stata ideata e studiata negli anni trenta.

Il funzionamento

Detto questo, vediamo come funziona una MDT. Innanzitutto una cella del nastro può contenere o il carattere blank oppure uno dei caratteri del cosiddetto alfabeto del nastro: in genere una manciata di simboli qualsiasi (in numero finito), di solito quelli che conviene a noi trattare, come le cifre binarie 0 e 1, le cifre decimali, le lettere dell'alfabeto inglese o altro.

La parte di controllo funziona a stati finiti: durante l'elaborazione a seconda dello stato del calcolo, assumerà un proprio stato interno. Operativamente parlando, la parte di controllo legge dal nastro un simbolo, a seconda di questo e del suo stato interno deciderà (univocamente):

a) cosa riscrivere sul nastro nella medesima cella

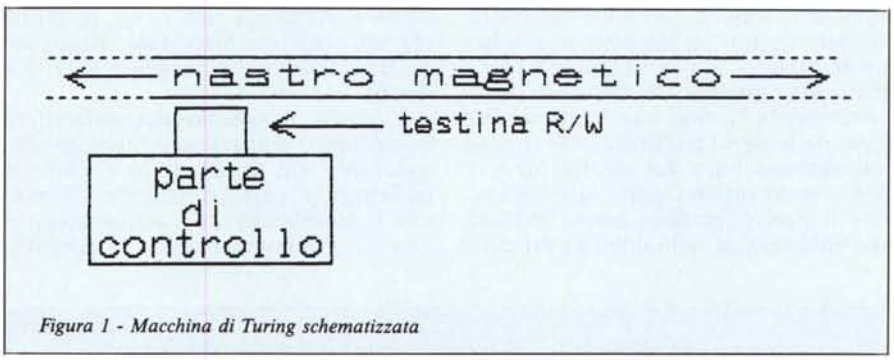

Figura 1 - Macchina di Turing schematizzata

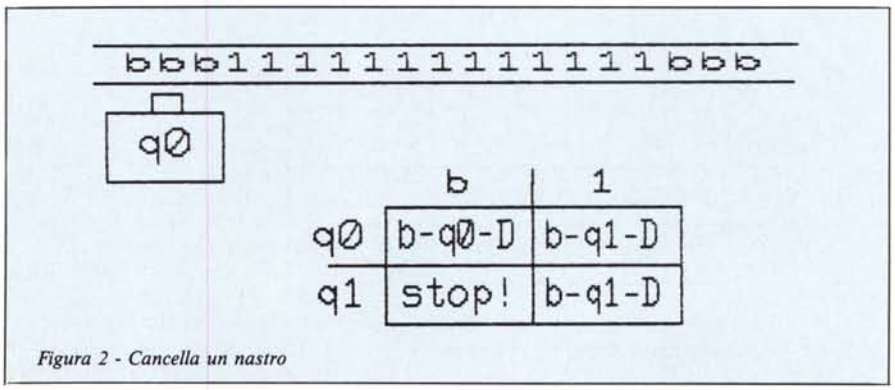

Figura 2 - Cancella un nastro

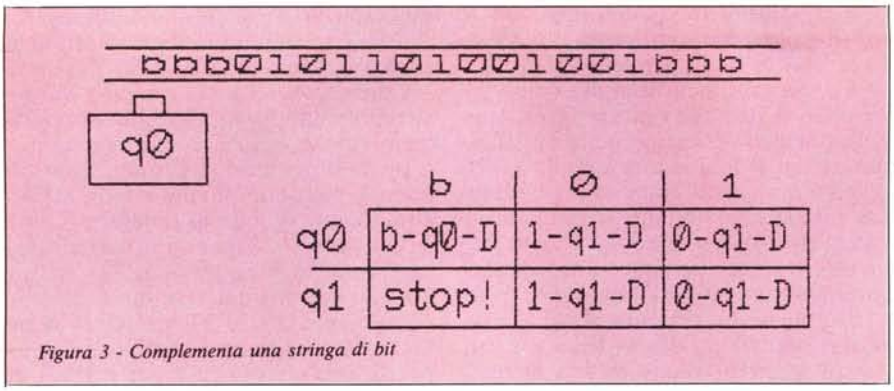

Figura 3 - Complementa una stringa di bit

b) se posizionare la testina sulla cella a destra o a sinistra della cella appena letta
c) in quale dei suoi possibili stati interni traslare.

Univocamente nel senso che se dovesse ritrovarsi in un secondo momento nelle stesse condizioni (stesso simbolo in lettura e stesso stato interno) effettuerà come conseguenza le stesse operazioni di prima.

Riassumendo il tutto, abbiamo che una elaborazione completa corrisponde a preincidere il nastro con i dati del programma, a memorizzare questo nella parte controllo e, a elaborazione ultimata, leggere dal nastro stesso il risultato del calcolo.

Il programma

Il programma della parte controllo altro non è che una bella tabellina che riassume ciò che questa dovrà fare in funzione del suo stato interno e del simbolo in lettura. La suddetta tabella conterrà quindi un insieme di quintuple del tipo (q, s, q', s', {DIS}) che identifica una precisa transizione della macchina di Turing. «q» è lo stato interno della parte controllo, «s» il simbolo appena letto dal nastro, «q'» il nuovo stato interno dopo tale lettura, «s'» il nuovo simbolo inciso sul nastro, nella stessa cella dove è avvenuta la lettura. {DIS} sta a indicare che la quinta posizione della quintupla è occupata da una D o da una S ovvero dove la testina dovrà spostarsi dopo la lettura e scrittura: a sinistra o a destra. Si noti che tanto il nuovo simbolo quanto il nuovo stato non necessariamente sono diversi da quelli precedenti.

Detto ciò se ad un certo istante la macchina di Turing si trova nello stato q_i e legge dal nastro il simbolo s_i non fa altro che andare a cercare tra le sue quintuple quella che inizia per q_i, s_i e comportarsi di conseguenza. Se tale quintupla non c'è, vuol dire che il calcolo è terminato e la MDT può arrestarsi.

Ad esempio: con stato interno q0 e simbolo in lettura «1», se nella nostra tabella abbiamo la quintupla (q0, 1, q1, b, S) significa che dobbiamo scrivere un blank, spostarci a sinistra e passare nello stato interno q1.... con stato interno q2, simbolo in lettura blank e quintupla (q2, b, q2, b, D) rimarremo nello stesso stato, riscrivendo il simbolo blank per poi spostarci a destra.

Essendo il calcolo deterministico, come detto, non possono esistere due quintuple con uguale stato e simbolo in lettura e parte rimanente diversa, che indurrebbero un comportamento non deterministico della macchina: aleatoriamente dovrebbe scegliere tra due comportamenti diversi. Ciò induce una rappresentazione più compatta e leggibile delle quintuple di una macchina di Turing: una tabella bidimensionale che ha per ascissa il simbolo in lettura e per ordinata lo stato interno le cui caselle contengono la tripla rimanente. A questo punto leggere cosa fare col dato simbolo e stato interno non vuol dire altro che (stile battaglia navale) far incrociare le due coordinate e leggere nella casella così trovata.

Sembra evidente a questo punto che senza un po' di esempi si rischia fortemente di diventare scemi.

Qualche esempio

Vedremo ora alcune macchine di Turing, programmate per fare delle semplici operazioni. Vista la non limitatezza della stessa, applicazioni più complesse richiedono solo insiemi di stati più grandi e relativa tabella delle transizioni opportunamente dimensionata: per ragioni di spazio ci occuperemo solo di casi semplici.

Un primo esempio, potrebbe essere una macchina di Turing che preso un nastro su cui sono incisi un certo numero di «1» contigui restituisce un nastro completamente blank. La situazione è mostrata in figura 2: dato che gli «1» possono essere solo in numero finito, è facile immaginare il nastro tutto blank fino a un certo punto, poi una quantità più o meno grande di «1» e dopo questi di nuovo tutti blank. Nella configurazione iniziale, la macchina è posizionata sui blank iniziali (a sinistra, per esempio) e il suo compito possiamo suddividerlo in due fasi: dapprima scorrere il nastro verso destra finché non troviamo un «1» e, trovatolo, cambiare questo e i successivi, con un blank finché non finiscono. Senza accorgercene abbiamo già identificato i due stati interni della macchina: ricerca e sostituzione che indicheremo rispettivamente con q0 e q1. Sempre in figura 2, è mostrata la tabella relativa al programma di cancellamento nastro e, indicato nella parte controllo della MDT, lo stato iniziale col quale viene avviata la macchina.

Uno sguardo alla tabella per rendersi subito conto della semplicità di una di queste macchine. In ogni posizione della tabella, identificata come detto da una ascissa (simbolo in lettura, b sta per blank) e da una ordinata (stato del controllo), leggiamo cosa la macchina, farà in ognuna delle possibili situazioni. Ad esempio, con stato interno q0 e simbolo in lettura blank leggiamo nella corrispondente casella la scritta b-q0-D: significa che riscriviamo un blank, rimaniamo in q0 e ci

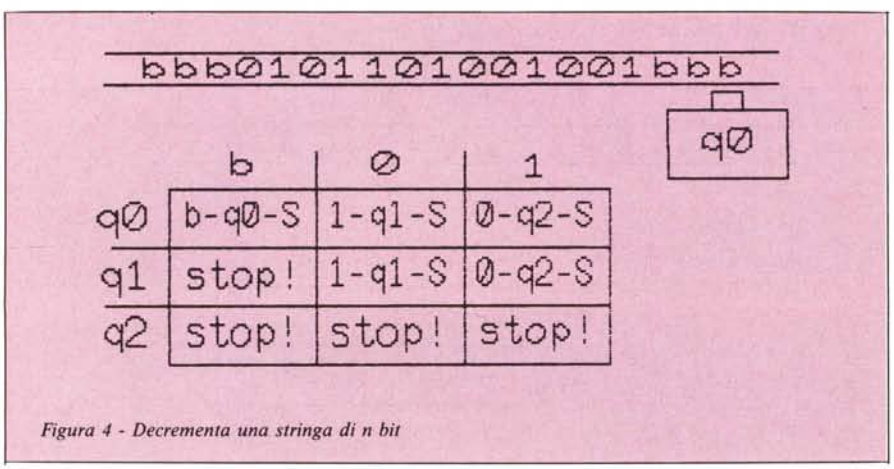

Figura 4 - Decrementa una stringa di n bit

spostiamo a destra. Ed è proprio quello che dovremo fare per scorrere il nastro fino al primo «1». Sempre da tabella, vediamo cosa succede quando incontriamo un «1» dallo stato q0. Leggiamo b-q1-D: vuol dire che scriviamo un blank (questa volta al posto dell'«1») trasliamo nello stato q1 (inizia la seconda fase, di cancellamento) e ci spostiamo a destra. In tale stato, sempre come da tabella, continuiamo a scambiare «1» con blank fino a quando non troviamo in lettura un blank (abbiamo finito gli «1»). Nella tabella, in posizione stato q1, simbolo b, troviamo scritto «stop!», ciò che la macchina in tale condizione, farà.

Secondo esempio: complemento a 1 di una stringa di bit (figura 3). La situazione è analoga a quella precedente: abbiamo una sequenza di «0» e «1» immersa nella miriade di blank di cui il nastro è composto. Complemento ad 1 significa che dovremo sostituire ad ogni «0» un «1» e viceversa. Sempre in figura 3 è mostrata la corrispondente tabella che descrive il programma «complemento». Stato iniziale e prima casella della tabella, come prima: continuiamo a scorrere il nastro finché non troviamo qualcosa diverso da un blank. A questo punto, se troviamo uno «0» scriviamo un «1», trasliamo nello stato q1 e ci spostiamo a destra; se troviamo un «1» scriviamo uno «0» e procediamo in maniera analoga. Nello stato q1 scambiamo «0» con «1» e viceversa (tenete sott'occhio sempre la tabella di figura 3) fino a quando non troviamo un blank: abbiamo finito e la macchina di Turing si può fermare.

Terzo esempio (un tantino più complicato): decremento di un numero binario di n bit modulo 2^n (figura 4). Ovvero preso un numero binario, si restituisce lo stesso numero decrementato di uno, e se il numero di partenza era 0, restituiamo il massimo numero binario rappresentabile con n bit. Esattamente come accade in linguaggio macchina e con un qualsiasi registro del processore. In questo esempio, a differenza di prima, la testina della macchina di Turing è posizionata sui blank a destra del nostro numero binario, quindi la prima cosa che farà sarà di scorrere verso sinistra fino al primo simbolo non blank: vedasi prima casella della tabella di figura 4 in cui con lettura di blank e stato interno q0 (quello iniziale) si riscrive il blank, si rimane in q0 e ci si sposta a sinistra.

Se come primo carattere incontrato troviamo un «1», è sufficiente scrivere al suo posto uno «0» e abbiamo finito; se incontriamo uno «0» bisogna ricorrere al ben noto prestito delle scuole elementari ovvero scrivere un «1» e manipolare le cifre successive tenendo conto che abbiamo un debito. Ciò si traduce nel fatto che continueremo a cambiare tutti gli «0» che incontreremo a sinistra con «1» sino al prossimo «1» sul nastro che complementeremo per poi fermare l'elaborazione. Se non troviamo altre cifre, ma un blank, ci fermeremo ugualmente. Quanto qui descritto a parole è esattamente ciò che è codificato nella tabella in figura 4 lo stato q0 è quello iniziale, lo stato q1 quello in condizione di debito, lo stato q2 di stop.

Infine, in figura 5, una macchina di Turing, che preso un numero naturale, restituisce lo stesso moltiplicato per due. Come nel caso precedente la scansione avviene da destra verso sinistra (a tal proposito nella tabella inserita in fig. 5 per ragioni di spazio è stato omesso lo spostamento della testina, da ritenersi sempre uguale a S, sinistra) e la tentazione di lasciare al lettore l'arduo compito di raccapezzarcisi, è forte. Per aiuto comunque diremo che lo stato q0 è come sempre quello iniziale, lo stato q1 è lo stato in cui va la macchina quando raddoppia una cifra minore di 5 (non c'è stato riporto), lo stato q2 è di contro quello in cui si trova la macchina quando deve riportare una unità (nel senso «elementare» del termine) alla cifra successiva (ovvero 7-q2 della tabella, ad esempio, potrebbe essere letto «scrivo 7 e porto 1»).

La tesi di Church

Dopo tutto questo parlare, è d'obbligo una domanda: siamo proprio sicuri che la Macchina di Turing sia in grado di calcolare qualsiasi funzione calcolabile? O meglio: esiste una dimostrazione del fatto che qualsiasi altro formalismo prendiamo esso non è più potente dell'automa di Turing?

Una tale dimostrazione non esiste: secondo Church e la sua tesi, qualunque algoritmo prendiamo, scritto in qualsiasi formalismo, esso può essere calcolato da una apposita MDT. Lo stesso affermò che tale macchina non solo esiste, ma è possibile costruirla effettivamente partendo dall'algoritmo e dal formalismo in questione. Purtroppo Church morì prima di riuscire a dimostrare il suo asserto e oggi, quello che poteva essere il teorema più importante della teoria della computabilità resta solo una tesi. Ovviamente ci potrà riuscire qualcun altro così come potrebbe essere dimostrato che Church aveva torto.

Resta però da sottolineare il fatto che altri formalismi, completamente diversi da quello di Turing, partoriti in epoche assai diverse e per vie diverse, messi a confronto, risultano essere meno potenti o avere la stessa potenzialità della MDT. Il confronto consiste naturalmente nel fornire un procedimento effettivo (ed eseguibile) per passare da un formalismo ad un altro. Quando riusciamo a passare indifferentemente dal primo al secondo e viceversa, i due formalismi sono equipotenti, se ci si riesce solo in un verso è più potente quello che, ovviamente, riesce a coprire anche gli algoritmi calcolati dall'altro. In tutte le ricerche effettuate e a confronti avvenuti, semplicemente il formalismo della macchina di Turing non è stato mai «battuto». Tutto qui.

MC

bbb43165721438Ø2bbb

q0

	b	Ø	1	2	3	4	5	6	7	8	9
q0	b-q0	Ø-q1	2-q1	4-q1	6-q1	8-q1	Ø-q2	2-q2	4-q2	6-q2	8-q2
q1	stop	Ø-q1	2-q1	4-q1	6-q1	8-q1	Ø-q2	2-q2	4-q2	6-q2	8-q2
q2	1-q1	1-q1	3-q1	5-q1	7-q1	9-q1	1-q2	3-q2	5-q2	7-q2	9-q2

Figura 5 - Raddoppia un numero naturale

BIBLIOGRAFIA

Aiello, Albano, Attardi, Montanari:
Teoria della computabilità logica, teoria dei linguaggi formali
Editrice ETS Pisa, 1976