

ALGORITMI, MACCHINE, GRAMMATICHE

MARCELLO FRIXIONE

*Dip. di Scienze della Comunicazione,
Università di Salerno*

frix@dist.unige.it

2008

1. Il concetto di algoritmo

1.1 Che cos'è un algoritmo

Gli *algoritmi* sono metodi per la soluzione di problemi. Possiamo caratterizzare un problema mediante i dati di cui si dispone all'inizio e dei risultati che si vogliono ottenere: risolvere un problema significa ottenere in uscita i risultati desiderati a partire da un certo insieme di dati presi in ingresso. I dati in ingresso vengono anche detti (valori in) *input* e i risultati in uscita (valori in) *output*. Possiamo assumere che ciascun problema consista di un insieme di casi particolari, o istanze. Ogni istanza di un problema è caratterizzata da un insieme specifico di dati in ingresso e da un determinato risultato. Supponiamo ad esempio che il problema generale consista nel calcolare la lunghezza dell'ipotenusa di un triangolo rettangolo date le lunghezze dei due cateti. In questo caso le istanze del problema corrispondono agli specifici triangoli di cui calcolare l'ipotenusa. Le informazioni in ingresso (i dati in *input*) sono le lunghezze dei cateti; il risultato che ci si attende in uscita (l'*output*) è la lunghezza dell'ipotenusa.

Un algoritmo per risolvere un problema è un metodo che consente di calcolare il risultato desiderato a partire dai dati di partenza. Cioè, a partire dai dati in *input*, consente di calcolare l'*output* corrispondente. Il comportamento di un algoritmo può essere schematizzato come nella fig. 1-1.

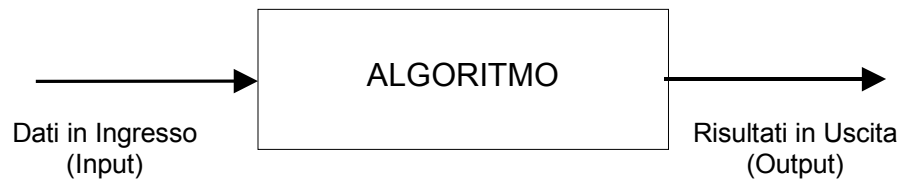


Figura 1-1

Affinché un metodo per la soluzione di un problema costituisca un algoritmo deve essere totalmente esplicito: vanno specificati in maniera precisa e particolareggiata tutti i passi del procedimento da eseguire per ottenere i risultati in uscita a partire dai dati in ingresso. Nel caso del triangolo rettangolo un possibile algoritmo è costituito dalla seguente lista di istruzioni:

- si prenda in input la lunghezza A del primo cateto
- si prenda in input la lunghezza B del secondo cateto
- si calcoli il quadrato di A
- si calcoli il quadrato di B
- si sommino i due quadrati
- si estraiga la radice quadrata del valore così ottenuto
- si produca in output quest'ultimo valore

In generale, un algoritmo è un procedimento di calcolo costituito da un insieme di istruzioni. Tali istruzioni fanno uso di un insieme finito di operazioni elementari, le quali si possono assumere come note e primitive (nell'esempio precedente sono state assunte come note le operazioni di elevamento al quadrato, addizione ed estrazione di radice quadrata). Le istruzioni devono essere tali che, per poterle applicare, basti saper

eseguire le operazioni elementari. Inoltre, affinché un procedimento sia un algoritmo, deve godere delle seguenti proprietà:

- L'insieme delle istruzioni di cui è composto deve essere finito.
- Se la soluzione esiste, deve poter essere ottenuta mediante un numero finito di applicazioni delle istruzioni.
- All'inizio del calcolo, e ogni qual volta sia stata eseguita un'istruzione, si deve sempre sapere in maniera precisa quale istruzione va eseguita al passo successivo, e quindi non devono esserci due istruzioni diverse che possono essere applicate nello stesso momento. In altri termini, in ogni fase del calcolo non deve mai accadere che, per sapere quale istruzione si deve eseguire, ci si debba basare sull'intuizione, o si debba tirare a indovinare. Un procedimento che goda di questa proprietà è detto *deterministico*.
- Infine, deve essere sempre chiaro se si è giunti o meno al termine del procedimento, e se sono stati ottenuti i risultati desiderati.

Quindi gli algoritmi (che sono detti anche *metodi effettivi*) sono procedimenti deterministici che consentono di risolvere determinati problemi senza far ricorso ad alcuna forma di creatività o di inventiva. Per eseguire un algoritmo è sufficiente applicare le istruzioni passo dopo passo, badando solo a non commettere sviste.

Dal fatto che un algoritmo è un processo deterministico consegue che, una volta fissati i dati, il risultato ottenuto è sempre lo stesso. Non può succedere che, eseguendo più volte lo stesso algoritmo con lo stesso input, vengano prodotti output diversi.

Si noti che, pur essendo un algoritmo caratterizzato da un insieme finito di istruzioni, le possibili istanze del problema che esso risolve sono, di norma, infinite. Ad esempio, il precedente algoritmo calcola la lunghezza dell'ipotenusa per ogni coppia A, B di lunghezze dei cateti, ed esiste un numero infinito di tali coppie.

Esempi di algoritmi possono essere tratti dalle matematiche elementari. Sono algoritmi, ad esempio, i procedimenti che consentono di eseguire le quattro operazioni aritmetiche. In questo caso l'input è costituito dai due numeri su cui operare e l'output dal risultato dell'operazione. Come pure è un algoritmo il procedimento euclideo per la ricerca del massimo comun divisore di due numeri naturali non nulli. In logica, il metodo delle tavole di verità è un algoritmo che permette di stabilire se una formula del linguaggio proposizionale è o meno una tautologia. In questo caso l'input è costituito da una formula proposizionale F , l'output è una risposta del tipo *sì/no*: *sì* se F è una tautologia, *no* in caso contrario.

Vi sono algoritmi che operano su dati di tipo numerico e algoritmi che elaborano altri tipi di dati. Vediamo un esempio di questo secondo tipo. Si dice che una parola è *palindroma* (oppure che essa è un *palindromo*) se può essere letta indifferentemente da sinistra a destra o viceversa. Ad esempio, *ossesso* è un palindromo. Il problema che vogliamo risolvere consiste nello stabilire se, data una certa parola, essa è palindroma o meno. L'input del problema è costituito dalla parola di cui vogliamo sapere se è palindroma. L'output è una risposta di tipo *sì* o *no*: *sì* se la parola presa in input è palindroma, *no* se non lo è.

Un possibile algoritmo per questo compito si comporta nel modo seguente. Si inizia confrontando la prima e l'ultima lettera della parola:

ossesso

Se la prima lettera è diversa dall'ultima, il procedimento termina e la risposta è negativa: non si tratta di un palindromo. Altrimenti si cancellano la prima e l'ultima lettera, e si ripete il procedimento dall'inizio con le lettere rimaste:

ssess

Se, dopo un certo numero di iterazioni del procedimento, non è rimasta alcuna lettera, allora il procedimento termina e la risposta è positiva: la parola di partenza era effettivamente un palindromo (se la parola presa in input è palindroma, ed è composta da un numero dispari di lettere – come è appunto il caso di *ossesso* – allora nell'ultima iterazione la prima e l'ultima lettera coincidono e vengono cancellate).

Un altro esempio di algoritmo che elabora dati di tipo non necessariamente numerico riguarda la ricerca di un dato elemento in un elenco ordinato. Si supponga di voler controllare se un certo nome N figura o meno in un elenco di nomi L ordinato alfabeticamente. In questo caso l'input del problema è costituito dal nome N e dall'elenco L , mentre l'output consiste anche qui in una risposta del tipo *sì* o *no*: *sì* se il nome cercato è presente in L , *no* in caso contrario. Un algoritmo ovvio per risolvere questo problema consiste nello scorrere tutto l'elenco partendo dall'inizio fino a che non si trova N , oppure si arriva alla fine dell'elenco senza averlo trovato. Ovviamente questo metodo, che viene detto *ricerca di tipo sequenziale*, funziona, ma è estremamente inefficiente: se, ad esempio, N compare all'ultimo posto nell'elenco, per trovarlo è necessario controllare tutti i nomi presenti.

Dato però che, per ipotesi, l'elenco L è ordinato, si può progettare un algoritmo più efficiente basandosi su una tecnica che gli informatici chiamano *ricerca binaria*. Il principio è il seguente. Si confronta il nome N con il nome che si trova a metà dell'elenco. (Ovviamente, se l'elenco è composto da un numero pari di nomi, bisogna precisare cosa si deve intendere per nome che si trova a metà dell'elenco. Assumiamo che, se L ha n elementi, l'elemento a metà di L sia quello in posizione $quoz(n, 2) + 1$, dove *quoz* è il quoziente della divisione intera tra numeri naturali.) Così, se L ha 10 elementi, il nome a metà di L è quello nella sesta posizione. A questo punto si possono dare tre possibilità:

- (a) il nome a metà dell'elenco coincide con N
- (b) il nome a metà dell'elenco segue N in ordine alfabetico
- (c) il nome a metà dell'elenco precede N in ordine alfabetico

Se si è verificato il caso (a) la ricerca ha avuto successo, e il procedimento può terminare.

Se invece si è verificato il caso (b), allora se N figura nell'elenco, deve trovarsi nella sua metà iniziale. Ripetiamo quindi il procedimento prendendo ora in considerazione la prima metà dell'elenco. Vale a dire, da questo punto in poi chiamiamo L la metà iniziale dell'elenco, e procediamo come prima: confrontiamo il nome cercato con il nome che si trova a metà di L , a quel punto si danno di nuovo tre possibilità, e così via.

Analogamente, se si è verificato il caso (c), il procedimento va eseguito sulla seconda metà dell'elenco.

È evidente che, se il nome N è nell'elenco, ripetendo questo procedimento un numero finito di volte, esso prima o poi verrà trovato. Altrimenti, se N non è nell'elenco, il procedimento avrà comunque termine: a un certo punto, dopo avere

ripetuto il procedimento un numero finito di volte, il pezzo di elenco L che dovremmo prendere in considerazione sarà vuoto, e il calcolo avrà termine.

È anche evidente che questo secondo algoritmo è molto più “intelligente” di quello basato sulla ricerca sequenziale. Esso infatti, in media, per ottenere il risultato richiede un numero molto minore di passi di calcolo¹.

Gli algoritmi di questi due esempi forniscono entrambi una risposta di tipo *sì* o *no* (gli algoritmi con questa caratteristica vengono detti *algoritmi di decisione*). Non è necessario però che le cose stiano in questo modo. È facile ad esempio immaginare una variante del secondo problema, che consiste nel cercare il numero di telefono di una data persona nella guida telefonica. In questo caso l'input è costituito dalla guida (cioè, da un elenco di nomi ordinati alfabeticamente, a ciascuno dei quali è associato il rispettivo numero telefonico) e dal nome della persona di cui si vuole il numero; l'output è dato dal numero di telefono corrispondente (o da una risposta di fallimento se il nome non è sulla guida).

Sin dall'antichità sono stati sviluppati algoritmi per risolvere svariati tipi di problemi. Tuttavia, soltanto nel corso del ventesimo secolo la nozione stessa di algoritmo è diventata uno specifico oggetto di ricerca. Ciò è avvenuto con la nascita di una nuova disciplina, detta *teoria della computabilità*, o *teoria della computabilità effettiva*, o anche *teoria della ricorsività*. La nozione di algoritmo presentata sopra ha un carattere intuitivo, non è basata su una definizione rigorosa di tipo matematico. La teoria della computabilità è stata sviluppata a partire dagli anni intorno al 1930, ed è stata motivata dell'esigenza di fornire un equivalente rigoroso del concetto intuitivo di algoritmo, al fine di indagare le possibilità ed i limiti dei metodi effettivi.

Per le caratteristiche di determinismo e finitezza che abbiamo enunciato, ogni algoritmo si presta, almeno in linea di principio, ad essere automatizzato, ad essere cioè eseguito da una macchina opportunamente progettata. Con lo sviluppo dei calcolatori digitali la teoria della computabilità effettiva ha dunque assunto lo statuto di teoria dei fondamenti per l'informatica, e svolge un ruolo importante nelle riflessioni teoriche relative a tutte le discipline che a qualche titolo sono collegate all'informatica.

Nella parte restante di questa sezione continueremo ad occuparci della nozione intuitiva di algoritmo, ed introdurremo alcune definizioni che ci saranno utili nel seguito.

1.2 Diagrammi di flusso

Non appena si abbia a che fare con algoritmi un po' più complicati di quelli visti nel paragrafo precedente, una descrizione a parole come quelle impiegate fino ad ora diventa scomoda e di difficile comprensione. Un modo più perspicuo e sintetico per rappresentare algoritmi è costituito dai cosiddetti *diagrammi di flusso* (in inglese *flow chart*), talvolta chiamati anche *diagrammi a blocchi*. I diagrammi di flusso utilizzano una notazione di tipo grafico. La fig. 1-2 mostra un possibile diagramma di flusso per l'algoritmo descritto nel paragrafo precedente, che calcola la lunghezza dell'ipotenusa di un triangolo rettangolo a partire da quelle dei due cateti.

¹ Per la precisione, con il primo algoritmo, se L ha n elementi, nel caso peggiore saranno necessarie n operazioni di confronto. Con il secondo algoritmo, nel caso peggiore il numero delle operazioni di confronto necessarie risulterà dell'ordine di $\log_2 n$.

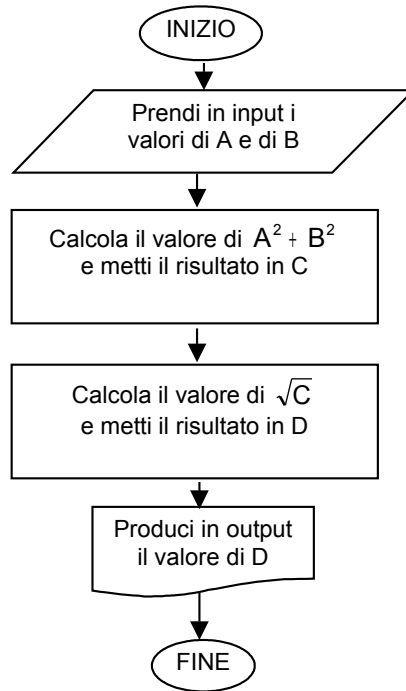


Figura 1-2

La lettura di un diagramma come questo è intuitiva. In generale, i diagrammi di flusso sono dei grafi orientati i cui nodi rappresentano le istruzioni da eseguire. La forma di ciascun nodo indica il tipo di istruzione corrispondente. Gli archi che li collegano rappresentano l'ordine in cui tali istruzioni devono essere effettuate. Essi rappresentano appunto il *flusso* delle informazioni durante l'esecuzione dell'algoritmo.

Vediamo nei particolari i tipi di nodi che compongono il diagramma. In ogni diagramma di flusso vi è un nodo *inizio* e un nodo *fine*, raffigurati entrambi mediante delle ellissi (fig. 1-3). Essi indicano rispettivamente da dove si deve partire per iniziare il calcolo e quando si è giunti al termine dell'esecuzione dell'algoritmo².



Figura 1-3

I nodi la cui forma è raffigurata in fig. 1-4 a) e b) rappresentano rispettivamente operazioni di input e di output.

² In un diagramma di flusso può esserci più di un nodo *fine* (anche se, in questi casi, è sempre possibile scrivere un altro diagramma di flusso equivalente in cui il nodo *fine* compare una sola volta). Non può comparire invece più di un nodo *inizio*, poiché altrimenti non sarebbe più stabilito univocamente da dove si deve iniziare il calcolo, e si perderebbe così la caratteristica del determinismo.

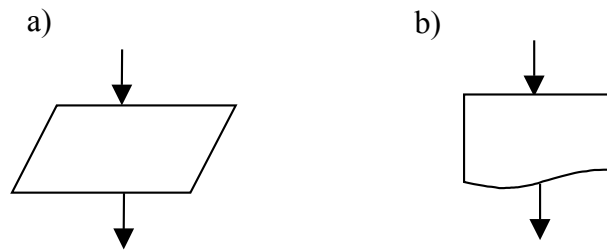


Figura 1-4

I valori presi in input e i risultati delle elaborazioni compiute durante il calcolo vengono immagazzinati in *variabili*. Ad esempio, l'algoritmo della fig. 1-2 utilizza le variabili A, B, C e D. Le variabili impiegate nei diagrammi di flusso vanno intese come locazioni di memoria, come celle in cui sono depositati dei dati, e il cui contenuto può essere modificato nel corso del calcolo. Le istruzioni che portano a modificare il valore di una variabile vengono rappresentate mediante nodi di forma rettangolare, come quello della fig. 1-5.

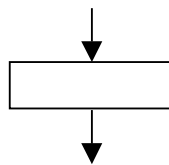


Figura 1-5

Un ulteriore tipo di nodi impiegato nei diagrammi di flusso, che non compare nell'algoritmo della fig. 1-2, è costituito dai *test*. Graficamente i test sono rappresentati per mezzo di rombi, come nella fig. 1-6. All'interno del rombo è scritta un'espressione che viene detta la *condizione* del test. Affinché un'espressione possa fungere da condizione deve poter assumere un valore di verità, vero o falso. Se la condizione di un test è vera, allora nell'esecuzione dell'algoritmo si segue la freccia contrassegnata con 'SI'. Se la condizione è falsa, allora si segue la freccia contrassegnata con 'NO'.

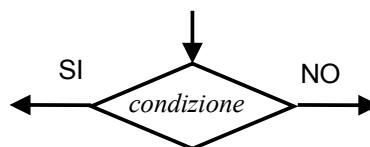


Figura 1-6

Come esempio di impiego del test, presentiamo un semplice algoritmo che, preso in input un numero, produce in output il suo valore assoluto (fig. 1-7). L'algoritmo memorizza nella variabile A il numero preso in input; dopo di che controlla se il valore di A è maggiore o uguale a 0. In caso affermativo assegna il valore di A alla variabile B. Altrimenti assegna a B il valore di A cambiato di segno. Dopo di che produce in output il valore di B, e il calcolo termina.

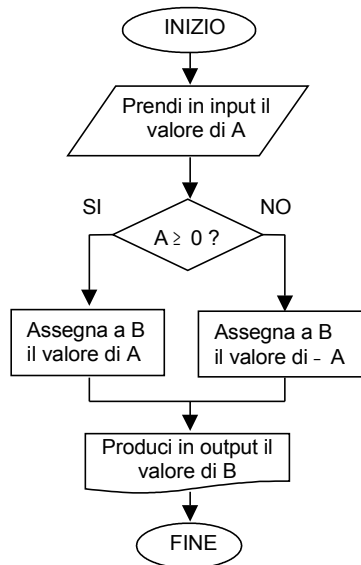


Figura 1-7

Questi sono gli elementi di base che impiegheremo per la costruzione dei diagrammi di flusso. Per ora non ci interessa stabilire con precisione quali operazioni si possano utilizzare all'interno dei blocchi. Basta che si tratti di operazioni che, intuitivamente, siano effettuabili in modo algoritmico (come è il caso delle usuali operazioni aritmetiche). In tal caso è evidente che l'intero procedimento descritto da un diagramma di flusso è a sua volta un algoritmo.

Nei diagrammi di flusso i test possono essere impiegati per la definizione di *cicli*, mediante i quali una stessa istruzione, o un gruppo di istruzioni, possono essere ripetuti più volte. La fig. 1-8 mostra due tipici esempi di strutture cicliche. Nel ciclo di fig. 1-8 a) per prima cosa viene controllato il valore della condizione; se essa è vera, allora vengono eseguite le istruzioni $istruzione_1, \dots, istruzione_n$. Dopo di che, si torna a controllare la condizione, e fino a che essa resta vera le istruzioni vengono ripetute. Il ciclo termina la prima volta che la condizione diventa falsa.

Nel ciclo di fig. 1-8 b) per prima cosa vengono eseguite le istruzioni $istruzione_1, \dots, istruzione_n$; dopo di che viene controllata la condizione del test; se essa risulta falsa, allora $istruzione_1, \dots, istruzione_n$ vengono eseguite di nuovo, e ciò si ripete fino a quando la condizione diventa vera. Il ciclo termina la prima volta che la condizione diventa vera.

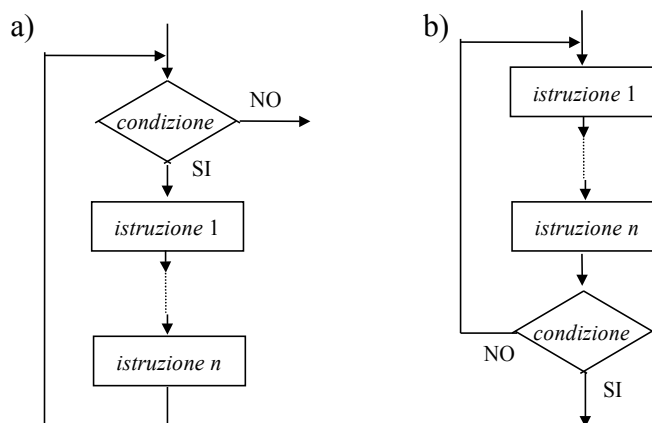


Figura 1-8

Vediamo ora un esempio di diagramma di flusso che rappresenta un algoritmo basato su di un ciclo come quello della fig. 1-8 a). Assumendo come nota l'operazione di addizione, l'algoritmo della fig. 1-9 prende in input due numeri naturali e ne calcola il prodotto.

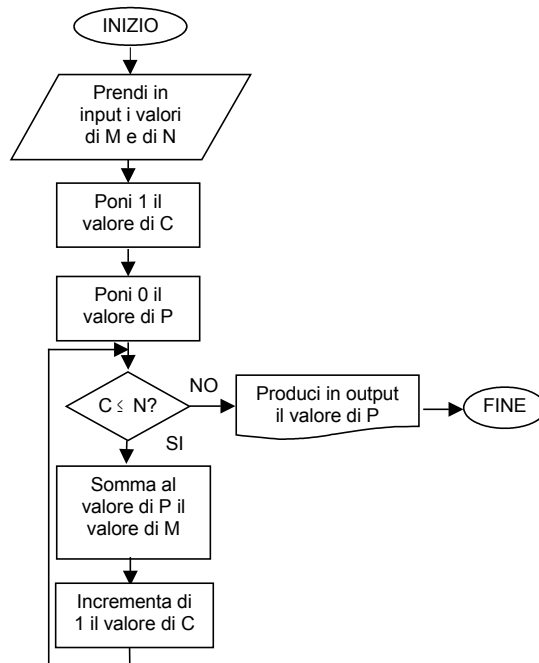


Figura 1-9

Questo algoritmo prende in input i due fattori da moltiplicare, e li memorizza nelle variabili M e N. Dopo di che calcola il prodotto di M per N sommando M a se stesso per N volte. Ciò si ottiene mediante un ciclo e, per fare sì che esso venga ripetuto N volte, viene impiegata un'altra variabile, che abbiamo chiamato C. Prima del ciclo il valore di C è posto uguale a 1. A ogni iterazione del ciclo C viene incrementata di 1; quando il valore di C supera quello di N il ciclo viene fatto terminare, ed è prodotto il valore in output. In informatica una variabile usata come C viene detta *contatore*. Per calcolare il risultato si è usata la variabile P. All'inizio il valore di P viene posto uguale a 0. Ad ogni iterazione, al valore di P è sommato il valore di M, di modo che alla fine in P si ottiene il valore di M sommato a se stesso N volte.

L'algoritmo che verifica se una parola è palindroma, illustrato nel paragrafo precedente, può essere espresso mediante un diagramma di flusso basato su un ciclo (fig. 1-10).

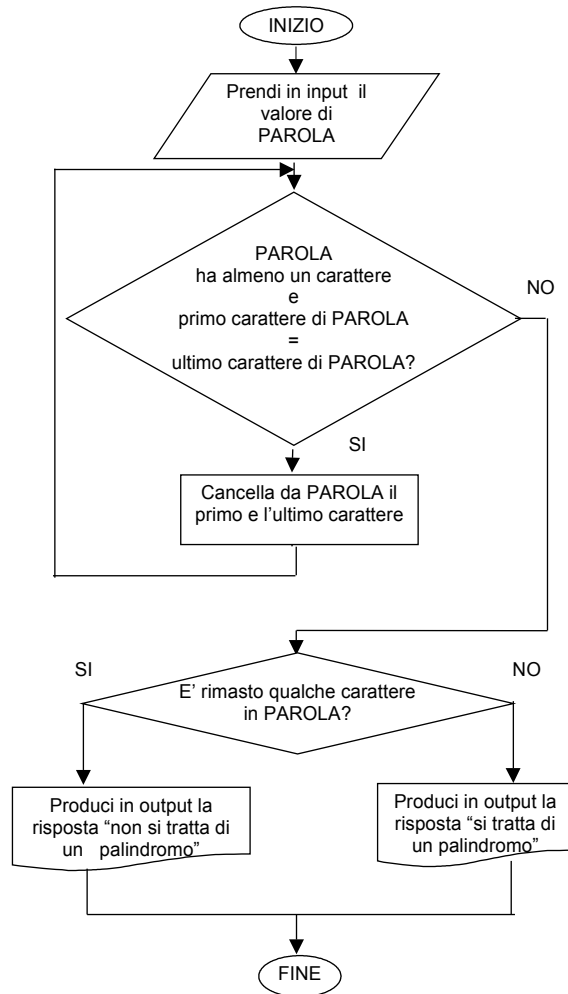


Figura 1-10

Per specificare in tutti i particolari gli algoritmi come questo che operano su dati non numerici (al fine, ad esempio, di implementarli mediante un linguaggio di programmazione), bisognerebbe precisare come devono essere rappresentati i dati da elaborare (cosa che invece può essere data per scontata nel caso di algoritmi che elaborano dati numerici). In questo caso, ad esempio, andrebbe specificato come va rappresentata la parola di cui si vuole controllare se è palindroma; nel caso dell'algoritmo per la ricerca di un nome in un elenco (si veda il paragrafo precedente) andrebbe specificato come vanno rappresentati i nomi e l'elenco ordinato. Si dovrebbe cioè (secondo la terminologia informatica) precisare su quali *strutture dati* l'algoritmo opera. Questi aspetti, che sono centrali dal punto di vista informatico, non sono rilevanti per i nostri scopi, per cui nel seguito li tralascieremo.

1.3 Algoritmi che non sempre producono risultati

Ci sono algoritmi che, per alcuni dei possibili input, non producono in output alcun risultato. Un semplice esempio è costituito dall'algoritmo di fig. 1-11, il quale esegue la sottrazione tra due numeri naturali. Essa è definita soltanto se il minuendo è maggiore o uguale al sottraendo. Pertanto l'algoritmo di fig. 1-11 si comporta come segue: prende in input due numeri naturali A e B; dopo di che, se A è maggiore o uguale a B, produce come output la differenza tra A e B; altrimenti termina senza produrre risultato.

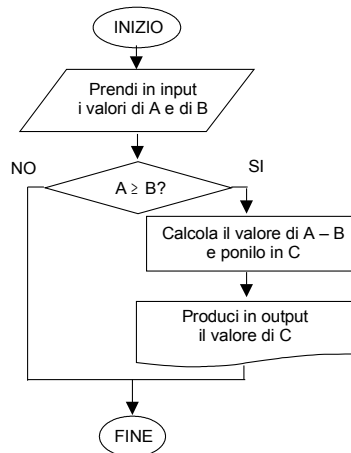


Figura 1-11

L'algoritmo di fig. 1-11 in alcuni casi non produce risultati. Tuttavia, per ogni coppia di numeri presi in input, dà sempre origine a un calcolo che termina. Vi sono algoritmi che, per alcuni input, non producono alcun risultato in quanto danno origine a un calcolo che non termina. Si considerino ad esempio i diagrammi della fig. 1-12.

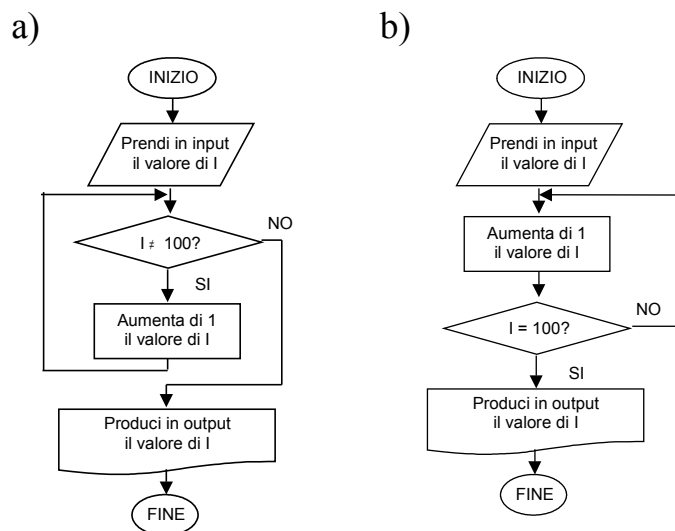


Figura 1-12

Nel caso dell'algoritmo di fig. 1-12 a), supponiamo che venga dato in input alla variabile I un qualsiasi numero maggiore di 100. La condizione risulterà vera, quindi si inizierà ad eseguire il ciclo. Verrà incrementato di 1 il valore di I; si tornerà quindi a verificare la condizione, che risulterà ancora vera. Poiché ad ogni iterazione il valore di

I è destinato a crescere, la condizione del ciclo non diventerà mai falsa, e il ciclo in linea di principio è destinato a continuare all'infinito. Considerazioni analoghe valgono nel caso dell'algoritmo di fig. 1-12 b): se il valore preso in input è maggiore di 100, la condizione resterà sempre falsa, e il ciclo non terminerà mai.

Quindi, per alcuni valori in input, cicli come questi possono dare luogo a un calcolo che non termina. In gergo informatico, si dice che in questi casi un algoritmo va in *loop* (in inglese "loop" significa "cappio", "occhiello").

Un altro esempio di algoritmo che in alcuni casi va in loop è dato dal diagramma di flusso di fig. 1-13. Esso prende in input un numero naturale x e, se x è un quadrato perfetto, ne calcola la radice quadrata³. Dopo aver preso in input il valore di x , l'algoritmo pone a zero il valore di una variabile y , e controlla se x è uguale a y^2 . Nel caso il risultato di questo test sia positivo, il calcolo è terminato: y è la radice quadrata di x , e il suo valore viene prodotto in output. Altrimenti il valore di y viene incrementato di uno, e si torna a controllare se x è uguale a y^2 . È facile constatare che, se x è un quadrato perfetto, allora prima o poi il calcolo termina, e viene prodotta in output la radice quadrata di x . Altrimenti, se x non è un quadrato perfetto, il test $y^2 = x$ sarà sempre falso, e il calcolo andrà avanti all'infinito senza produrre alcun risultato.

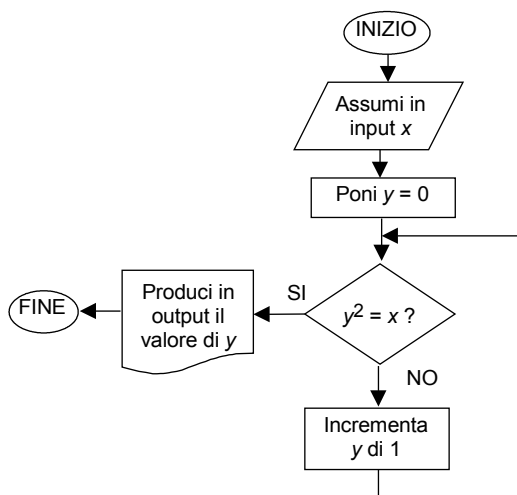


Figura 1-13

1.4 Numeri naturali e codifiche dei dati

Spesso quando si studia la nozione di algoritmo ci si concentra su algoritmi che elaborano numeri naturali. Ciò potrebbe sembrare riduttivo: abbiamo visto che esistono algoritmi definiti su enti assai diversi dai numeri naturali. Vi sono algoritmi che stabiliscono se un certo oggetto appartiene o meno a un dato insieme, o se gode o non gode di una certa proprietà. Vi sono algoritmi che eseguono manipolazioni di vario genere sulle espressioni di un linguaggio o di un sistema formale. In informatica poi vengono impiegati algoritmi per elaborare dati della natura più diversa, dai testi alle immagini, dai suoni ai filmati.

Tuttavia il fatto di concentrarsi su algoritmi che elaborano numeri naturali non comporta una perdita di generalità. Infatti, esistono varie tecniche mediante le quali dati

³ Il calcolo viene effettuato in modo molto inefficiente, ma ciò, in questa sede, non è rilevante.

di tipo diverso possono essere codificati mediante numeri naturali, per cui algoritmi come quelli sopra citati vengono ricondotti ad algoritmi che elaborano dati di tipo numerico. Vediamo alcuni esempi ispirati all'informatica.

È noto che nella memoria di un calcolatore tutti i dati sono rappresentati sotto forma di sequenze di *bit*, ossia di cifre 0 e 1. Tali sequenze possono essere interpretate come la rappresentazione di numeri naturali espressi in notazione binaria (*bit* è infatti la contrazione di *binary digit*, ossia, appunto, *cifra binaria*). In questo modo tutte le manipolazioni che un calcolatore esegue sui dati possono essere lette in termini aritmetici, come calcoli di tipo algoritmico che operano su (insiemi di) numeri naturali. Esistono molteplici tecniche per codificare i dati sotto forma di sequenze di bit. Vediamo sinteticamente un paio di esempi tra i più semplici.

Consideriamo un tipo di dati molto diversi da quelli visti sino ad ora, ossia le *immagini*. In informatica sono state sviluppate svariate tecniche per codificare immagini in modo che possano essere elaborate con un calcolatore. Vediamo a grandi linee come funziona il metodo più semplice per ottenere la codifica binaria di un'immagine. Si supponga di avere a che fare con un disegno in bianco e nero. Si immagini di sovrapporre al disegno una griglia abbastanza fitta da riuscire a rappresentare la figura con il grado di dettaglio desiderato. A questo punto si assegna ad ogni cella della griglia il valore nero se la porzione corrispondente del disegno è prevalentemente nera, il valore bianco se la porzione corrispondente del disegno è prevalentemente bianca. Si otterrà così un'immagine come quella di fig. 1-14.



Figura 1-14

Se ne ingrandiamo un dettaglio, ad esempio la punta di uno dei baffi di Miomao, otterremo un particolare come quello di fig. 1-15, in cui sono evidenti le celle bianche e nere che compongono la griglia (che in questo caso è di dimensioni 266×398).

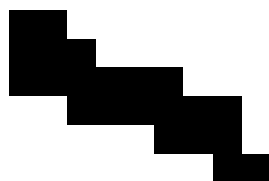


Figura 1-15

Ovviamente, quanto più fitta è la griglia, tanto migliore sarà la qualità dell'immagine. A titolo di esempio, riportiamo nella fig. 1-16 cinque versioni della stessa immagine, ottenute rispettivamente con griglie di dimensioni 20×30 , 40×60 , 50×75 , 80×120 e 120×180 .



Figura 1-16

A questo punto è facile passare a una codifica mediante cifre binarie. Basta rappresentare, ad esempio, ogni cella bianca con 0, e ogni cella nera con 1. In questo modo il disegno di partenza viene codificato con una sequenza di bit. Con buona approssimazione, questo è il tipo di procedimento che uno *scanner* esegue quando acquisisce un'immagine.

In informatica un'immagine codificata mediante questa tecnica viene chiamata *bitmap* (ossia, "mappa di bit"), e ognuna delle celle che la compongono è detta *pixel* (*pixel* sta per *picture element*). Anche immagini più ricche, ad esempio a colori o con vari toni di grigio, possono essere codificate sotto forma di bitmap. In questi casi ogni cella dovrà ammettere più di due valori (in particolare, sarà necessario un valore diverso per ogni possibile colore o tono di grigio). Sarà quindi necessaria più di una cifra binaria per rappresentare lo stato di ciascuna cella, ma, nella sostanza, la tecnica rimarrà immutata.

Grazie a codifiche di questo tipo le elaborazioni eseguite sulle immagini (come aumentarne il contrasto, passare da un'immagine al suo negativo, passare da un'immagine a colori ad una con toni di grigio, eccetera) possono essere viste come procedure che alla codifica dell'immagine iniziale presa come input associano come output la codifica dell'immagine elaborata. Ad esempio, sostituendo nella codifica di fig. 1-14 ciascuno 0 con 1 e ciascun 1 con 0 si ottiene il negativo dell'immagine di partenza (fig. 1-17).

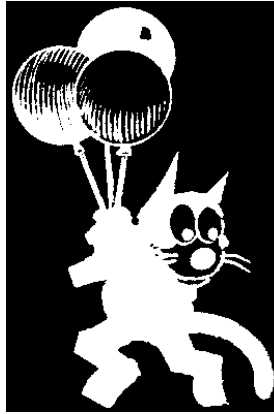


Figura 1-17

Tecniche analoghe si possono impiegare per rappresentare altri tipi di dati. Consideriamo ad esempio i *suoni*. Supponiamo di voler rappresentare un'onda sonora, come quella mostrata nella fig. 1-18. L'asse delle ascisse rappresenta la dimensione del tempo. In estrema sintesi, si può impiegare una tecnica di questo tipo. Si suddivide l'asse delle ascisse x in intervalli che possono essere scelti anche molto piccoli. Dopo di che, per ciascuno di questi intervalli i , si calcola il valore medio delle ordinate dei punti che hanno ascissa in i , e lo si codifica con un numero naturale. Tale operazione viene detta *campionamento*. L'onda sonora di partenza è ora rappresentata sotto forma di un insieme ordinato finito di numeri naturali. La rappresentazione sarà tanto più fedele quanto più piccoli sono gli intervalli della suddivisione.

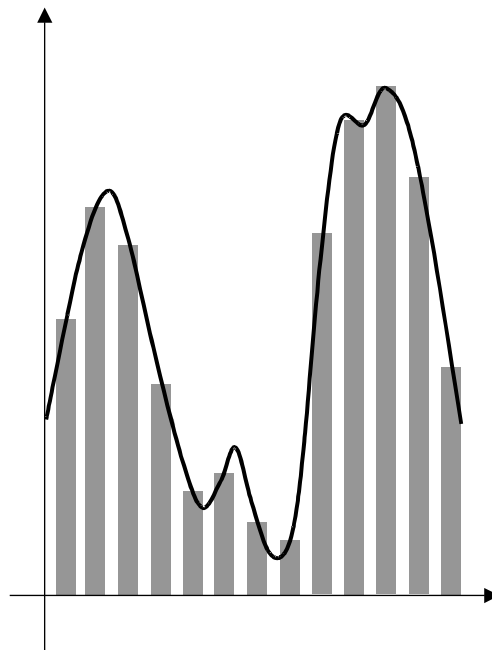


Figura 1-18

I numeri naturali possono dunque essere visti come un mezzo per rappresentare dati generici di tipo discreto, e in questo modo saranno intesi nel seguito di questo testo. Si noti tra l'altro che nell'ultimo esempio abbiamo codificato con numeri naturali i numeri decimali che sono le ordinate della curva di fig. 1-18. Come noto i numeri reali sono rappresentati da numeri decimali finiti o periodici (se sono razionali), o infiniti e non

periodici (se sono irrazionali). In un calcolatore digitale in ogni caso essi vengono approssimati con numeri decimali finiti, i quali possono essere codificati mediante numeri naturali.

Analogico e digitale: il regolo calcolatore

In un *calcolo di tipo analogico* i dati vengono rappresentati per mezzo di grandezze fisiche che variano in modo continuo (ad esempio grandezze elettriche come la corrente o il voltaggio, oppure grandezze geometrico-meccaniche, come la rotazione o lo spostamento reciproco di determinate componenti). I calcoli vengono effettuati agendo fisicamente su tali grandezze. I calcoli analogici si contrappongono ai *calcoli di tipo digitale*, in cui i dati vengono codificati mediante un insieme discreto di simboli, e le computazioni consistono di manipolazioni definite su tali codifiche simboliche. Le codifiche descritte nel testo sono tutte di tipo digitale.

La distinzione analogico/digitale riguarda il modo in cui vengono rappresentati ed elaborati i dati, e non è una distinzione tra tipi diversi di *hardware*. Così vi sono calcolatori elettronici analogici (in cui ad esempio i dati sono rappresentati in termini di voltaggio), come pure calcolatori meccanici sia analogici, sia digitali.

In questi appunti ci occuperemo esclusivamente di calcoli digitali. Tuttavia, per meglio chiarire la distinzione, esaminiamo qui un semplice dispositivo di calcolo analogico. Si tratta del *regolo calcolatore*, inventato nel XVII secolo dal matematico inglese Edmund Gunter. Probabilmente si tratta del calcolatore analogico che storicamente ha avuto la maggiore diffusione. Vediamo in sintesi come funziona.

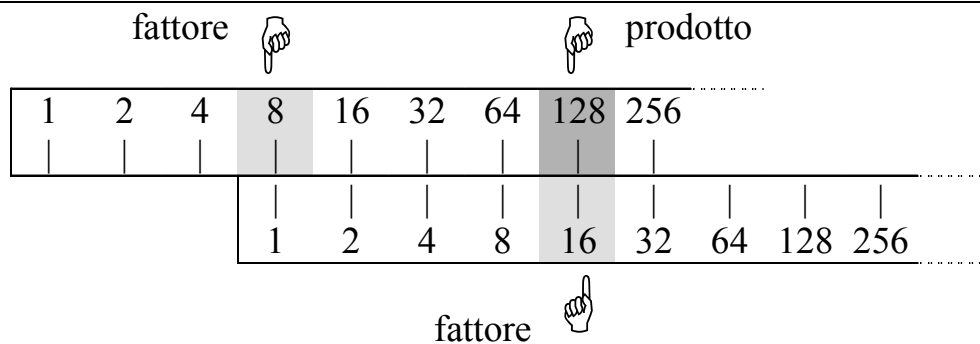
Affianchiamo due righelli, in maniera che siano liberi di scorrere l'uno rispetto all'altro verso destra e verso sinistra. Segniamo su ciascuno di essi delle tacche a distanze uguali, e associamo a ogni tacca una potenza di 2:

20	21	22	23	24	25	26	27	28
20	21	22	23	24	25	26	27	28

In maniera equivalente, possiamo indicare sui righelli i valori corrispondenti:

1	2	4	8	16	32	64	128	256
1	2	4	8	16	32	64	128	256

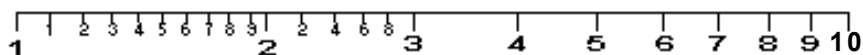
A questo punto i righelli possono essere usati per moltiplicare tra loro le potenze di 2. Supponiamo di voler moltiplicare 8 per 16. Si fa scorrere verso destra il righello inferiore in modo che la tacca del numero 1 si venga a trovare in corrispondenza della tacca del numero 8 sul righello superiore:



Ora, per ottenere il prodotto di 8 e di 16, basta andare a leggere sul righello superiore il numero che corrisponde a 16 sul righello in basso; si ottiene così che $8 \cdot 16 = 128$. Infatti, la distanza $d = 3$ tra la tacca 1 e la tacca 8 sommata alla distanza $d' = 4$ tra la tacca 1 e la tacca 16 è uguale alla distanza $d'' = 7$ tra la tacca 1 e la tacca 128. Questo accade perché, per come abbiamo segnato le tacche, d è tale che $2^d = 8$, ossia d è il *logaritmo in base 2* di 8 (in simboli, $d = \log_2 8$), d' è il *logaritmo in base 2* di 16, e, di conseguenza, $d'' = d + d' = 7 = \log_2 128$ in quanto $128 = 8 \cdot 16 = 2^3 \cdot 2^4 = 2^{3+4}$.

In generale, il *logaritmo in base 2* di un numero n ($\log_2 n$) è quel numero d tale che $2^d = n$. Dati due numeri n e n' , se $d = \log_2 n$ e $d' = \log_2 n'$, allora $d + d' = \log_2(n \cdot n')$. Infatti $n \cdot n' = 2^d \cdot 2^{d'} = 2^{d+d'}$. Su questo si basa il funzionamento del regolo.

Si può infatti generalizzare il procedimento dei righelli scorrevoli in modo da moltiplicare numeri qualsiasi. A tal fine bisogna completare i righelli aggiungendo le tacche dei numeri che non sono potenze di due. Ad esempio si dovrà aggiungere la tacca del 3 tra il 2 e il 4, le tacche del 5, del 6 e del 7 tra il 4 e l'8, e così via. Tutto questo facendo in modo che la distanza della tacca di ciascun numero n da 1 sia uguale al *logaritmo in base 2* di n . Il *logaritmo* di un numero che non sia una potenza di due è un numero reale non razionale. Ad esempio, $\log_2 3 = 1,58496250072\dots$, per cui la tacca del 3 dovrà essere segnata a una distanza $d = 1,58496250072\dots$ dalla tacca dell'1. O ancora, $\log_2 7 = 2,80735492205\dots$, per cui la tacca del 7 dovrà essere segnata a una distanza $d' = 2,80735492205\dots$ dalla tacca dell'1. E così via. In questo modo, sommando le distanze d e d' , si ottiene $d'' = 4,39231742277\dots$, che è il *logaritmo in base 2* di 21 (infatti 21 è il prodotto di 3 e di 7). La scala delle tacche sui righelli del regolo avrà l'aspetto seguente (in questo caso tra le tacche 1 e 2 e tra le tacche 2 e 3 sono indicati anche alcuni valori decimali):



Riportiamo qui di seguito il particolare di un regolo reale:



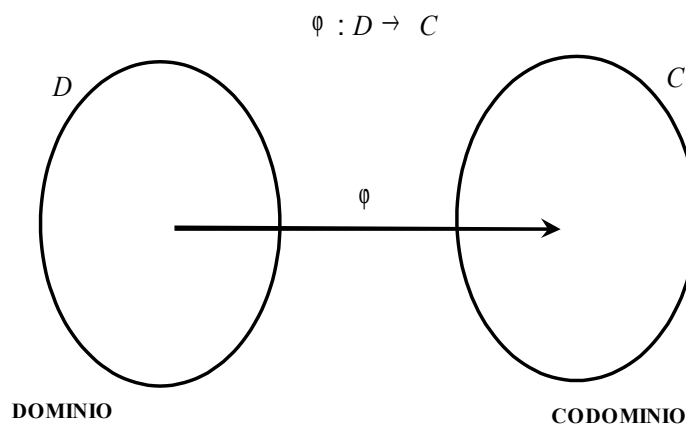
Il regolo è un calcolatore analogico in cui la grandezza fisica che viene utilizzata per il calcolo è lo spostamento reciproco dei due righelli. Essa viene usata per rappresentare i numeri da moltiplicare. Si tratta di una grandezza continua poiché il procedimento che abbiamo descritto non funziona soltanto per le tacche che sono segnate esplicitamente sui righelli (che saranno comunque un insieme discreto). Facendo scorrere in modo continuo i righelli l'uno rispetto all'altro, ogni posizione è "significativa", nel senso che, scelte tre distanze qualunque d , d' e d'' , se $d'' = d + d'$, allora $2^d \cdot 2^{d'} = 2^{d''}$, a prescindere dal fatto che nei punti corrispondenti sia segnata sui righelli una tacca o meno. In altri termini, in linea di principio le tacche sui righelli potrebbero essere fitte quanto si vuole. Ovviamente dal punto di vista pratico tale possibilità di principio è limitata dalla precisione con cui è possibile effettuare le misurazioni. Questo è un problema generale di tutti i calcolatori di tipo analogico, che non trova corrispettivo nel calcolo digitale. Le limitazioni nella precisione delle misurazioni comportano che, in generale, i calcoli di tipo digitale consentano di ottenere una precisione maggiore di quelli analogici.

1.5 Funzioni e algoritmi

1.5.1 Il concetto di funzione

Come vedremo nel prossimo sottoparagrafo, il concetto di algoritmo si collega in modo naturale al concetto matematico di funzione. Introduciamo dunque gli aspetti essenziali della nozione di funzione. In generale, una **funzione** φ è una corrispondenza tra due insiemi (li indichiamo con D e C) che, ad ogni elemento di D preso come **argomento**, associa come **valore uno ed un solo** elemento di C . In altri termini, per ogni argomento $x \in D$, esiste uno ed un solo valore $y \in C$ che φ fa corrispondere a x , e si scrive $\varphi(x) = y$ (oppure $y = \varphi(x)$)⁴.

L'insieme D in cui una funzione φ assume i suoi argomenti si dice il **dominio** di φ ; l'insieme C in cui una funzione φ assume i suoi valori si dice il **codominio** di φ . Per indicare che una funzione φ ha dominio D e codominio C , si scrive $\varphi: D \rightarrow C$ (e si dice che φ è di **tipo** $D \rightarrow C$). La situazione è schematizzata in fig. 1-19.



⁴ \in è il simbolo di appartenenza insiemistica: la scrittura $a \in I$ significa che l'elemento a appartiene all'insieme I .

Figura 1-19

Il concetto di funzione è molto generale in quanto non solo dominio e codominio possono essere due insiemi qualunque, ma anche la corrispondenza può essere di natura qualsiasi. Ad esempio, dato un insieme D di persone, si ottiene una funzione di dominio D facendo corrispondere ad ogni persona in D la sua altezza; in tal caso il codominio C è costituito da numeri decimali finiti che esprimono le misure delle altezze rispetto ad un'unità di misura (solitamente il metro). Si ottengono altre funzioni considerando come varia la temperatura corporea di una persona al trascorrere del tempo, o la quantità di un bene prodotta al variare dell'anno di produzione, eccetera.

Se D è l'insieme delle regioni italiane e C l'insieme delle città italiane, si ottiene una funzione φ di dominio D e codominio C se si fa corrispondere a ciascuna regione il suo capoluogo ($\varphi(\text{Piemonte}) = \text{Torino}$, $\varphi(\text{Liguria}) = \text{Genova}$, e così via); **non** si ottiene una funzione di dominio D se come corrispondenza si assume quella determinata da "essere capoluogo di provincia", poiché, in tal caso, non si associa a ciascuna regione una ed una sola città (le province del Piemonte sono otto, quelle della Liguria quattro, e così via). Così, se D è un insieme di persone e si associa a ciascuna di esse sua madre, la corrispondenza individua una funzione (in quanto ciascun individuo ha una ed una sola madre)⁵, mentre, se D è un insieme di donne e si associano a ciascuna di esse i suoi figli, in generale **non** si ha una funzione (in quanto una donna può non avere figli o averne più di uno).

In matematica si è interessati per lo più a funzioni in cui dominio e codominio sono insiemi di numeri. In tal caso la funzione φ viene spesso individuata fornendo un'espressione algebrica $\varphi(x)$ contenente la lettera x : se a x si sostituisce un numero a del dominio e si eseguono i calcoli indicati nell'espressione $\varphi(x)$, si determina il valore $\varphi(a)$ della funzione che corrisponde all'argomento a . Ad esempio, se assumiamo come dominio e codominio l'insieme \mathbf{R} dei numeri reali, allora con l'espressione:

$$\varphi(x) = x^2 + 4 \quad (*)$$

si definisce una funzione φ di tipo $\mathbf{R} \rightarrow \mathbf{R}$. Infatti, per ogni elemento di \mathbf{R} che si sostituisce a x in (*), $\varphi(x)$ assume come valore uno ed un solo elemento di \mathbf{R} . Ad esempio: $\varphi(3) = 3^2 + 4 = 13$, $\varphi(1,75) = 1,75^2 + 4 = 7,0625$, $\varphi(\) = \$ e così via. Se assumiamo che a x si sostituiscano numeri naturali (ossia che il dominio sia \mathbf{N}), allora (*) definisce una funzione di tipo $\mathbf{N} \rightarrow \mathbf{N}$ in quanto anche i valori sono in \mathbf{N} .

Altri semplici esempi di funzioni definite in questo modo sono:

$$\varphi_1(x) = 2x$$

$$\varphi_2(x) = 4$$

$$\varphi_3(x) =$$

⁵ Naturalmente, occorre che al codominio C appartengano tutte le madri degli individui di D .

La funzione φ_1 raddoppia il numero preso come argomento. La funzione φ_2 è una *funzione costante*: essa associa a ciascun elemento del dominio sempre lo stesso valore nel codominio (in questo caso il numero 4). Il valore della funzione φ_3 è 0 per tutti gli argomenti strettamente maggiori di 2, ed è 1 per tutti gli argomenti minori o uguali a 2. In realtà, come si è detto, ciascuna di esse individua funzioni diverse al variare del dominio; ad esempio, può essere vista sia come una funzione di tipo $\mathbf{N} \rightarrow \mathbf{N}$, sia come di tipo $\mathbf{R} \rightarrow \mathbf{R}$.

Finora abbiamo visto esempi di funzioni ad un solo argomento. Si possono però considerare **funzioni a più argomenti**. In particolare, potremo avere funzioni $\varphi(x_1, x_2, \dots, x_n)$ con un numero n qualsiasi di argomenti. Ad esempio, l'*addizione* ($\varphi(x_1, x_2) = x_1 + x_2$) e la *moltiplicazione* ($\varphi(x_1, x_2) = x_1 \cdot x_2$) di due numeri sono funzioni a due argomenti. L'addizione aritmetica (cioè l'addizione definita sui numeri naturali) è una funzione che prende come argomenti coppie di numeri naturali e produce come valore un numero naturale: data come argomento la coppia (4, 3) l'addizione produce come somma 7, data come argomento la coppia (123, 256), l'addizione produce come somma 379, e così via.

In generale, sia $\varphi(x_1, x_2)$ un'espressione con due argomenti che prende il suo primo argomento x_1 nell'insieme D_1 , il suo secondo argomento x_2 nell'insieme D_2 , e il cui valore appartiene all'insieme C . Allora $\varphi(x_1, x_2)$ definisce una funzione φ di codominio C , e il cui dominio è costituito dall'insieme delle coppie ordinate (x_1, x_2) tali che $x_1 \in D_1$ e $x_2 \in D_2$. Tale insieme viene detto il *prodotto cartesiano* di D_1 e D_2 , e viene indicato con $D_1 \times D_2$. In tal caso φ è una funzione di tipo $D_1 \times D_2 \rightarrow C$.

Nel caso di una funzione aritmetica a due argomenti come l'addizione aritmetica, il dominio è costituito dall'insieme di tutte le coppie ordinate (x_1, x_2) tali che $x_1, x_2 \in \mathbf{N}$, vale a dire dal prodotto cartesiano di \mathbf{N} per se stesso $\mathbf{N} \times \mathbf{N}$, che si indica anche con \mathbf{N}^2 . L'addizione aritmetica è quindi una funzione di tipo $\mathbf{N}^2 \rightarrow \mathbf{N}$.

Generalizzando, un'espressione $\varphi(x_1, x_2, \dots, x_n)$ con n argomenti tale che, se $x_1 \in D_1$, $x_2 \in D_2, \dots, x_n \in D_n$ allora $\varphi(x_1, x_2, \dots, x_n) \in C$, individua una funzione φ di tipo $D_1 \times D_2 \times \dots \times D_n \rightarrow C$, che ha come dominio l'insieme delle n -ple ordinate $D_1 \times D_2 \times \dots \times D_n$, vale a dire il prodotto cartesiano dei vari D_i (con $1 \leq i \leq n$), e come codominio C .

I **connettivi** vero-funzionali **della logica proposizionale** (congiunzione, disgiunzione, negazione, e così via) possono essere visti come funzioni di tipo opportuno. Consideriamo ad esempio la congiunzione \wedge ("e"). Essa prende come argomenti due proposizioni qualsiasi A e B , ciascuna delle quali può assumere uno ed uno solo dei due valori di verità \mathbf{V} (vero) oppure \mathbf{F} (falso), e produce come valore la proposizione $A \wedge B$, il cui valore di verità è determinato sulla base dei valori di verità di A e di B come riportato nella seguente tabella ($A \wedge B$ è vera se e solo se A e B sono entrambe vere):

<i>A</i>	<i>B</i>	<i>A</i> ∧ <i>B</i>
V	V	V
V	F	F
F	V	F
F	F	F

Pertanto la congiunzione si comporta come una funzione che ha come dominio l'insieme $\{\mathbf{V}, \mathbf{F}\}^2$ (cioè l'insieme delle coppie ordinate (v_1, v_2) dove v_i , con $i = 1, 2$, sono valori di verità) e come codominio l'insieme $\{\mathbf{V}, \mathbf{F}\}$, ossia una funzione di tipo $\{\mathbf{V}, \mathbf{F}\}^2 \rightarrow \{\mathbf{V}, \mathbf{F}\}$.

Così la negazione \neg (“non”), che è il connettivo a un argomento caratterizzato dalla seguente tabella:

<i>A</i>	$\neg A$
V	F
F	V

può essere visto come la funzione η di tipo $\{\mathbf{V}, \mathbf{F}\} \rightarrow \{\mathbf{V}, \mathbf{F}\}$ tale che $\eta(\mathbf{V}) = \mathbf{F}$ e $\eta(\mathbf{F}) = \mathbf{V}$.

1.5.2 Funzioni calcolate da algoritmi

Ricollegiamoci ora alla nozione di algoritmo. Il problema che viene risolto da un algoritmo può essere visto come una funzione, in cui i dati in ingresso corrispondono agli argomenti, e il risultato in uscita corrisponde al valore. Dato che per ciascun input un algoritmo produce al più un solo output (e questo è garantito dall'assunzione di determinismo alla base della definizione di algoritmo), la corrispondenza tra input e output può essere caratterizzata in termini funzionali: l'algoritmo calcola una funzione dall'insieme degli input a quello degli output. Questa analogia può essere visualizzata come in fig. 1-20.

Figura 1-20

Queste considerazioni conducono alle seguenti definizioni:

Definizione 1. Dato un algoritmo A , si dice che esso *calcola una funzione* $\varphi: D \rightarrow C$ se e solo se, per ogni $x \in D$, $\varphi(x) = y$ se e soltanto se A con input x produce come output y .

Definizione 2. Si dice che una funzione è *calcolabile (computabile) in modo algoritmico*, o *calcolabile in modo effettivo*, o *effettivamente calcolabile (computabile)*, o più semplicemente *calcolabile (computabile)*, se e solo se esiste un algoritmo che la calcola.

Le funzioni aritmetiche che abbiamo menzionato in precedenza (addizione, moltiplicazione), come pure tutte quelle che si incontrano comunemente nella pratica matematica, sono calcolabili. Uno dei risultati più importanti della teoria della computabilità, sul quale torneremo nel seguito, è che *esistono funzioni non calcolabili*, ossia i cui valori non sono calcolabili mediante alcun algoritmo.

Una funzione effettivamente calcolabile non deve però essere confusa con un algoritmo che ne calcola i valori. Nonostante l'analogia evidenziata nella fig. 1-20, funzioni e algoritmi sono entità concettualmente distinte, e le funzioni sono generalmente definite in maniera indipendente dagli eventuali algoritmi che ne calcolano i valori.

Ciò segue dal fatto che, data una funzione effettivamente calcolabile, esistono più algoritmi per calcolarne i valori. Si considerino ad esempio i due algoritmi per la ricerca di un elemento in un elenco ordinato descritti nel par. 1.1. Essi "fanno la stessa cosa". Ossia, usando una terminologia più rigorosa, possiamo dire che calcolano la stessa funzione (si tratta di una funzione che ha come dominio un insieme che include gli oggetti presenti nell'elenco e come codominio un insieme composto da due elementi, che stanno per le risposte "sì" e "no"). O ancora, si consideri la funzione addizione. Essa può essere calcolata con molti algoritmi diversi. Uno è quello che si impara alle scuole elementari, basato sull'impiego della notazione decimale. Supponiamo però di rappresentare i numeri utilizzando una notazione binaria. Un algoritmo che somma due numeri in notazione binaria è necessariamente diverso da uno basato sulla notazione decimale. Si possono sviluppare anche algoritmi che sommano numeri rappresentati utilizzando le cifre romane. Si tratta di algoritmi diversi che calcolano tutti la stessa funzione, l'addizione. In generale, si può mostrare che per ogni funzione calcolabile esistono in linea di principio infiniti algoritmi che la calcolano.

Dal fatto che una stessa funzione può essere calcolata da più algoritmi diversi segue immediatamente che una funzione calcolabile non può essere identificata con un algoritmo che la calcola. Una funzione infatti viene definita in maniera del tutto indipendente rispetto ai metodi per computarla.

ESERCIZI RELATIVI ALLA PRIMA PARTE

Esercizio 1.1. Una sequenza di parentesi si dice *bilanciata* se, per ogni parentesi aperta, esiste una parentesi chiusa corrispondente. Ad esempio, le due sequenze di parentesi seguenti sono bilanciate:

((() (()))) () ((() ()) ())

mentre le seguenti non lo sono:

)) (()) (() ())

Descrivere a parole un algoritmo che, presa in input una sequenza di parentesi, controlli se essa è bilanciata o meno.

Esercizio 1.2. Rappresentare mediante un diagramma di flusso un algoritmo per la ricerca sequenziale di un nome in un elenco ordinato.

Esercizio 1.3. Rappresentare mediante un diagramma di flusso l'algoritmo per la ricerca binaria di un nome in un elenco ordinato, che abbiamo presentato nel paragrafo 1.1.

Esercizio 1.4. Rappresentare mediante un diagramma di flusso l'algoritmo dell'esercizio 1.1 che stabilisce se una sequenza di parentesi è bilanciata.

Esercizio 1.5. L'algoritmo della fig. 1-9 impiega un ciclo come quello della fig. 1-8 a). Formulare un algoritmo che produca gli stessi risultati impiegando una struttura come quella della fig. 1-8 b).

Esercizio 1.6. Formulare un algoritmo che, assunto come noto il prodotto, prenda in input due numeri naturali m ed n e produca in output m^n .

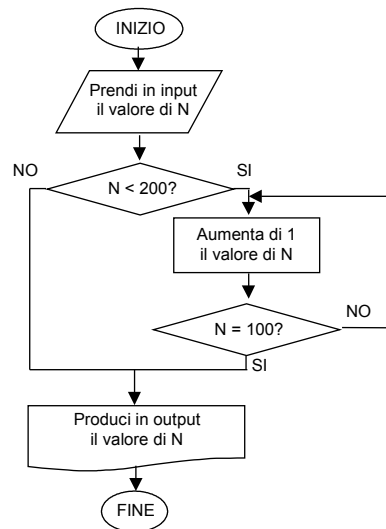
Esercizio 1.7. Si assuma come nota l'operazione *mod*, tale che $X \text{ mod } Y$ sia il resto della divisione di X per Y . Si rappresenti quindi mediante un diagramma di flusso un algoritmo che, preso in input un numero naturale N , produca in output 1 se N è primo, 0 se N non è primo.

Esercizio 1.8. Modificare l'algoritmo della fig. 1-11 in modo che produca sempre un risultato: se $A < B$, produca come output 0.

Esercizio 1.9. Determinare due algoritmi che vadano in *loop* per qualsiasi input, uno basato su un ciclo come quello della fig. 1-8 a), e l'altro basato su un ciclo come quello della fig. 1-8 b).

Esercizio 1.10. È possibile fare in modo che l'algoritmo della fig. 1-13 dia luogo a un calcolo che termina per qualunque input?

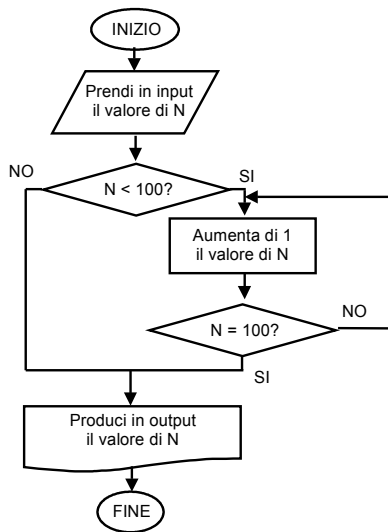
Esercizio 1.11. Stabilire se il seguente algoritmo:



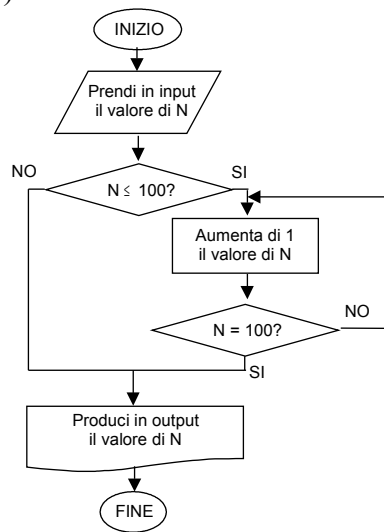
- (a) termina per qualunque valore di N
- (b) non termina per alcun valore di N
- (c) termina se e solo se $N > 200$ oppure $N < 100$
- (d) termina se e solo se $N \geq 200$ oppure $N \leq 100$
- (e) termina se e solo se $N \geq 200$ oppure $N < 100$
- (f) termina se $N > 200$
- (g) termina solo se $N > 200$

Esercizio 1.12. Determinare per quali valori di input i seguenti algoritmi terminano.

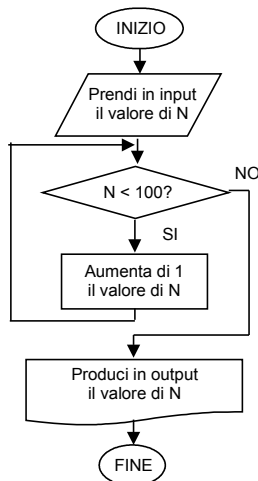
(a)



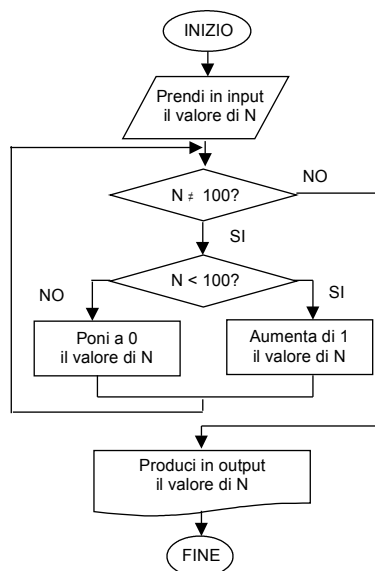
(b)



(c)

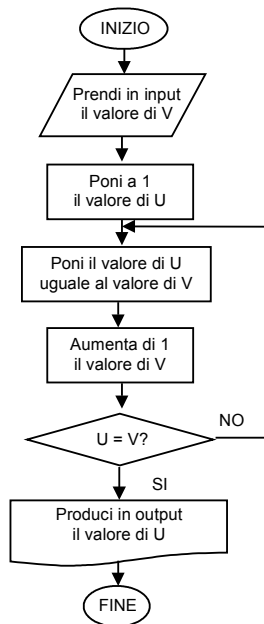


(d)

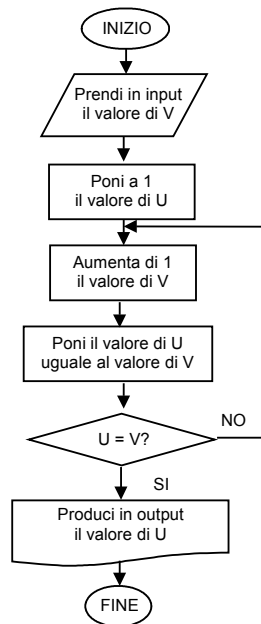


Esercizio 1.13. Stabilire quali dei seguenti algoritmi generano un calcolo che termina sempre.

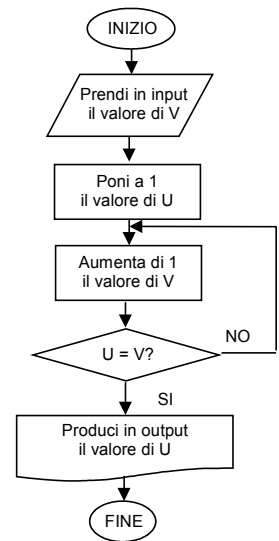
(a)



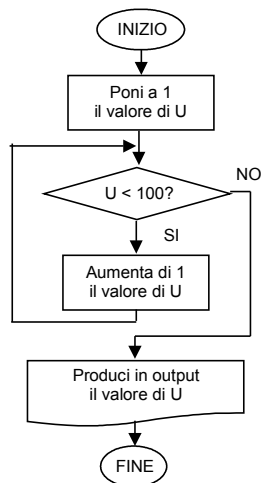
(b)



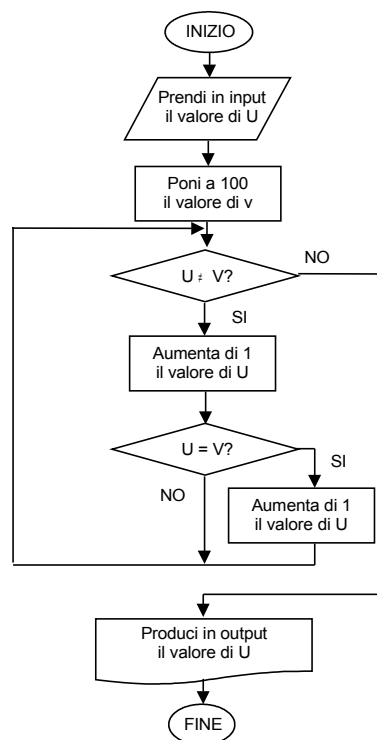
(c)



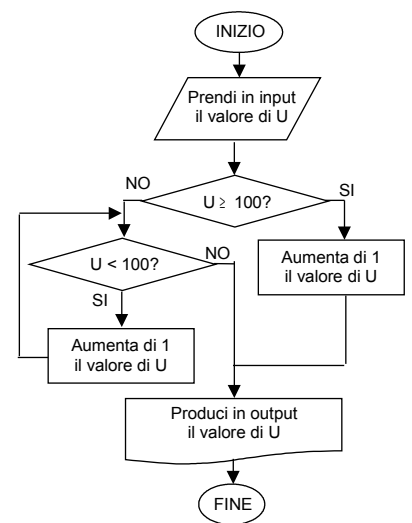
(d)



(e)



(f)



Esercizio 1.14. Sia D un insieme di stati. Stabilire se si ottiene una funzione di dominio D associando a ciascun elemento di D :

- gli stati ad esso confinanti
- la sua capitale
- le sue città con più di 100.000 abitanti

- d) il numero delle sue città con più di 100.000 abitanti
- e) il numero 5

Esercizio 1.15. Stabilire quali delle seguenti espressioni algebriche definiscono una funzione φ di tipo $\mathbf{N} \rightarrow \mathbf{N}$:

- a) $\varphi(x) = x + 27$
- b) $\varphi(x) = x - 10$
- c) $\varphi(x) =$
- d) $\varphi(x) =$
- e) $\varphi(x) =$
- f) $\varphi(x) =$

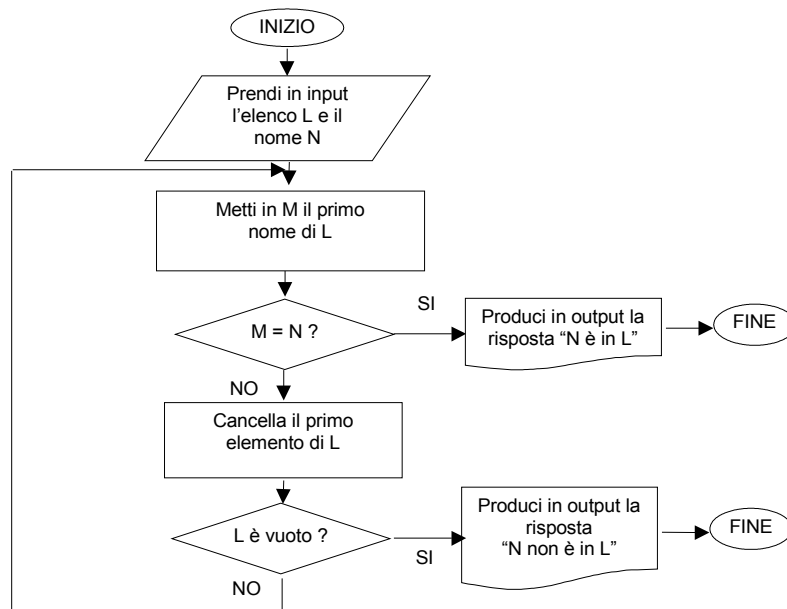
Esercizio 1.16. Come il precedente Esercizio 2.3, ma con $\varphi: \mathbf{R} \rightarrow \mathbf{R}$.

SOLUZIONI DI ALCUNI ESERCIZI RELATIVI ALLA PRIMA PARTE

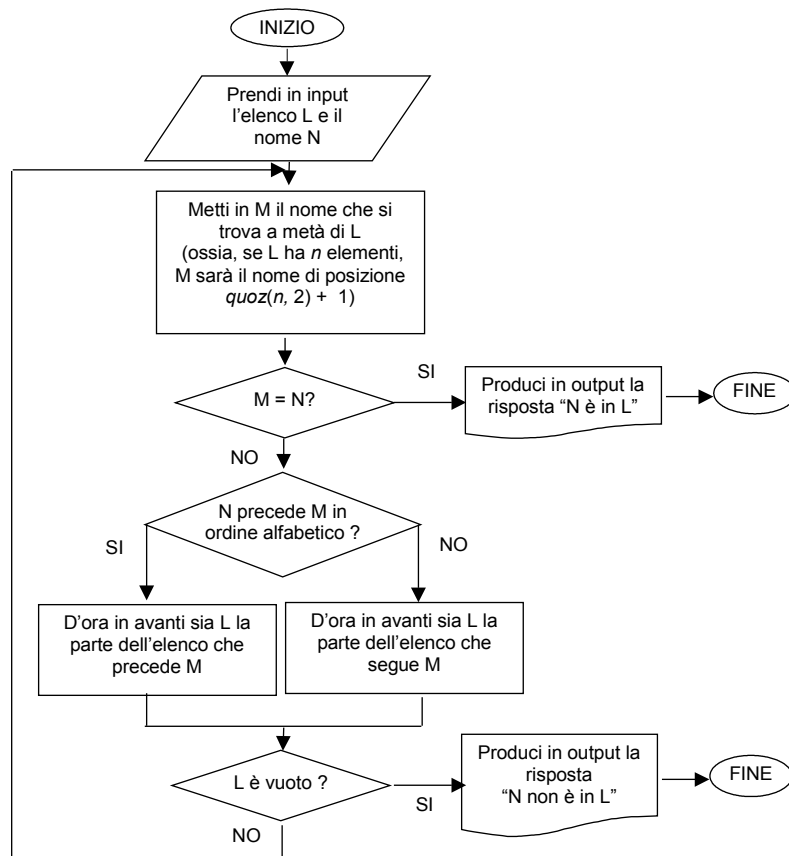
Esercizio 1.1. Un possibile algoritmo per questo compito è il seguente:

- Prendi in input una sequenza S di parentesi.
- (*) - Se S è vuota fermati: S è bilanciata.
- Altrimenti vai avanti fino a che non trovi una “)” oppure fino a che arrivi in fondo a S.
- Se sei arrivato in fondo a S senza trovare “)”, allora fermati: S non è bilanciata.
- Altrimenti cancella “)” e torna indietro di uno.
- Se trovi una “(“ cancellala e torna a (*). Altrimenti fermati: S non è bilanciata.

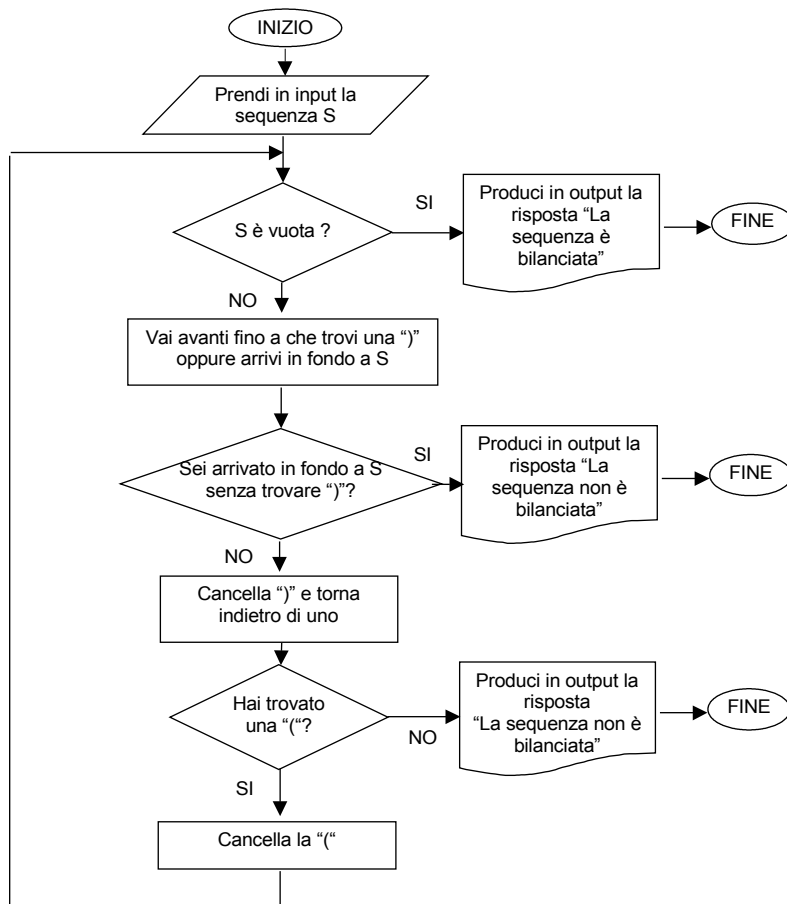
Esercizio 1.2. Un possibile diagramma per il compito specificato è il seguente:



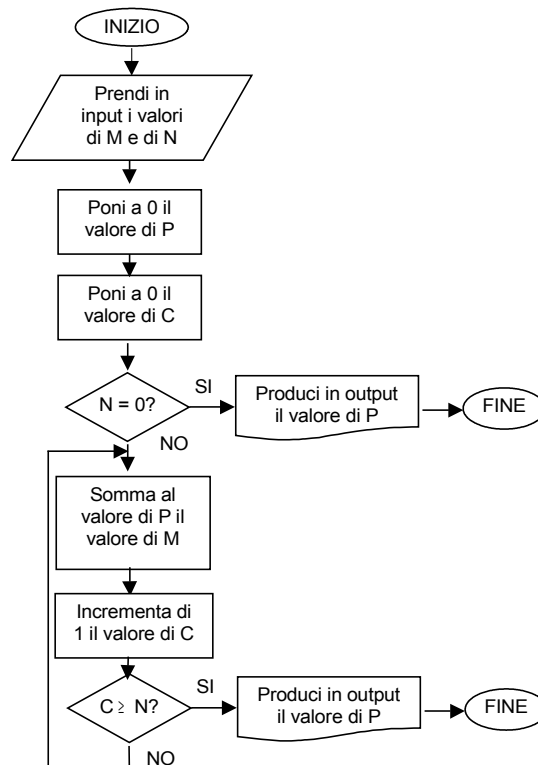
Esercizio 1.3. Un possibile diagramma per il compito specificato è il seguente:



Esercizio 1.4. Un possibile diagramma per il compito specificato è il seguente:



Esercizio 1.5. Un possibile algoritmo con le caratteristiche richieste è il seguente:



Esercizio 1.6. Una maniera per ottenere l'algoritmo voluto consiste nel modificare l'algoritmo di fig. 1-9 del testo, in modo che, all'inizio del calcolo, il valore di P venga posto uguale a 1 anziché a 0, e che l'istruzione *Somma al valore di P il valore di M* all'interno del ciclo diventi *Moltiplica il valore di P per il valore di M*.

Esercizio 1.14. Corrispondono a funzioni **b)**, **d)**, **e)**.

Esercizio 1.15. **a)** e **f)**.

Esercizio 1.16. Tutte tranne **c)**.

2. Macchine di Turing e teoria della computabilità

2.1. Verso una caratterizzazione rigorosa del concetto di algoritmo

Durante tutta la storia delle matematiche sono stati sviluppati algoritmi per risolvere classi sempre più estese di problemi. Tuttavia è soltanto in anni recenti che il concetto stesso di algoritmo è stato fatto oggetto diretto di ricerca matematica. Ciò è avvenuto attorno agli anni '30 del '900, nel contesto delle ricerche sui fondamenti della matematica. Le ricerche precedenti si erano basate su di una nozione di algoritmo del tutto intuitiva, non specificata in modo rigoroso. Tale nozione intuitiva fu del tutto sufficiente fin tanto che lo scopo che ci si proponeva era quello di individuare algoritmi che risolvessero problemi, o classi di problemi determinate. Ma con le ricerche sui fondamenti della matematica avvenne un radicale cambiamento di prospettiva. Furono poste domande di tipo nuovo, che non riguardavano più la possibilità di individuare algoritmi specifici, ma che concernevano l'intera classe dei procedimenti di tipo algoritmico. Soprattutto nel contesto del progetto fondazionalista proposto da David Hilbert, diventava fondamentale rispondere alla domanda se esistessero problemi matematici che non ammettono neppure in linea di principio di essere risolti mediante alcun algoritmo. Tutto ciò era a sua volta strettamente legato allo studio delle proprietà dei sistemi formali della logica matematica. Nel momento in cui la ricerca si indirizzò allo studio delle proprietà della classe di *tutti* i procedimenti algoritmici, tale classe dovette essere caratterizzata in maniera rigorosa, e non fu più sufficiente la tradizionale definizione informale ed intuitiva. Nacque così quel settore della logica matematica che è stato detto in seguito *teoria della computabilità* (o della *calcolabilità*) *effettiva* (oppure anche *teoria della ricorsività*), in cui vengono indagati concetti quali quello di algoritmo e di funzione computabile in modo algoritmico.

Durante gli anni '30 numerosi tra i maggiori logici del periodo, tra i quali Gödel, Church, Post, Kleene e Turing, affrontarono, da punti di vista differenti, il problema di individuare una definizione rigorosa della nozione di algoritmo. In queste pagine verrà presentato uno di questi approcci, quello seguito dal logico inglese Alan Turing, che, rispetto agli studi coevi sulla computabilità, presenta il vantaggio di affrontare il problema in maniera diretta, analizzando il comportamento di un soggetto umano computante, senza presupporre altre nozioni o strumenti formali elaborati nell'ambito della ricerca logico-matematica. Inoltre, Turing ha formulato la sua proposta nei termini di una particolare classe di macchine astratte, che possono essere considerate modelli idealizzati dei calcolatori reali. Infine, parte dell'interesse per il lavoro di Turing risiede nelle implicazioni avute in altri ambiti disciplinari, quali la filosofia della mente, l'intelligenza artificiale e le scienze cognitive.

Come già abbiamo anticipato, per le caratteristiche di determinismo e di finitezza che abbiamo enunciato, ogni algoritmo si presta, almeno in linea di principio, ad essere automatizzato, ad essere eseguito cioè da una macchina opportunamente progettata. Con lo sviluppo dell'informatica, la teoria della computabilità ha dunque assunto, in un certo senso, il ruolo di "teoria dei fondamenti" per questa disciplina.

2.2. Le macchine di Turing

Turing affrontò il problema di fornire un equivalente rigoroso del concetto intuitivo di algoritmo definendo un modello dell'attività di un essere umano che stia eseguendo un calcolo di tipo algoritmico. Egli elaborò tale modello nella forma di una classe di

dispositivi computazionali, di macchine calcolatrici astratte, che in seguito furono dette appunto *macchine di Turing* (d'ora in avanti MT). Le MT sono macchine astratte nel senso che, nel caratterizzarle, non vengono presi in considerazione quei vincoli che sono fondamentali se si intende progettare una macchina calcolatrice reale (ad esempio, le dimensioni della memoria, i tempi del calcolo, e così via), e soprattutto nel senso che esse sono definite a prescindere dalla loro realizzazione fisica (cioè, dal tipo di *hardware* utilizzato). Vale a dire, che cosa sia una MT dipende esclusivamente dalle relazioni funzionali che sussistono tra le sue parti, e non dal fatto di poter essere costruita con particolari dispositivi materiali.

Seguiamo l'analisi del processo di calcolo come viene condotta da Turing stesso nell'articolo "On computable numbers, with an application to the Entscheidungsproblem" (Turing 1936-37), dove il concetto di MT viene formulato per la prima volta. Un calcolo, osserva Turing, consiste nell'operare su di un certo insieme di simboli scritti su di un supporto fisico, ad esempio un foglio di carta. Turing argomenta che il fatto che abitualmente venga usato un supporto bidimensionale è inessenziale, e che si può quindi assumere, senza nulla perdere in generalità, che la nostra macchina calcolatrice utilizzi per la "scrittura" un *nastro* monodimensionale di lunghezza virtualmente illimitata in entrambe le direzioni (tuttavia, come vedremo, in ogni fase del calcolo la macchina potrà disporre soltanto di una porzione finita di esso). Tale nastro sia inoltre suddiviso in celle, in "quadretti", "come un quaderno di aritmetica per bambini", ciascuna delle quali potrà ospitare un solo simbolo alla volta (fig. 2.1).

Quanto ai simboli da utilizzare per il calcolo, ogni macchina potrà disporre soltanto di un insieme finito di essi, che chiameremo l'*alfabeto* di quella macchina. Il fatto che l'alfabeto di cui si può disporre sia finito non costituisce comunque una grave limitazione. È infatti sempre possibile rappresentare un nuovo simbolo mediante una sequenza finita di simboli dell'alfabeto, ed avere così la possibilità di esprimere un numero virtualmente infinito di simboli (come avviene usualmente nella numerazione decimale mediante cifre arabe). Sia dunque $\Sigma \equiv \{s_1, s_2, \dots, s_n\}$ l'alfabeto di una generica MT. Ogni cella del nastro potrà contenere uno di tali simboli, oppure, in alternativa, restare vuota (indicheremo con s_0 la cella vuota).

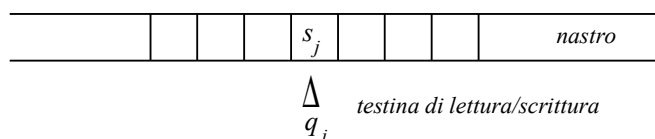


Fig. 2.1

Vi è senza dubbio un limite al numero di simboli che un essere umano può osservare senza spostare lo sguardo sul foglio su cui sta lavorando. Secondo Turing si può quindi assumere senza perdita di generalità che la macchina possa esaminare soltanto una cella alla volta, ed "osservare" ad ogni passo al più un singolo simbolo. A tal fine la macchina sarà dotata di una *testina di lettura*, che sarà collocata, in ogni fase del calcolo, su di una singola cella (fig. 2.1). Essa, per poter accedere alle altre celle del nastro, dovrà spostarsi verso destra o verso sinistra. Chi sta eseguendo un calcolo ha poi la possibilità di scrivere nuovi simboli, di cancellare quelli già scritti o di sostituirli con altri. La testina eseguirà anche tali compiti di cancellazione e di scrittura. Anche in questo caso però essa potrà agire soltanto sulla cella "osservata", e, per accedere ad altre

celle, dovrà prima spostarsi lungo il nastro. Poiché ogni cella può contenere un solo simbolo, scrivendo un nuovo simbolo in una cella il simbolo eventualmente presente in essa si deve ritenere cancellato.

Nell'eseguire un calcolo, un essere umano tiene conto delle operazioni già eseguite e dei simboli osservati in precedenza mediante la propria memoria, cambiando cioè il proprio "stato mentale". Al fine di simulare ciò, supporremo che una macchina possa assumere, in dipendenza dagli eventi precedenti del processo di calcolo, un certo numero di *stati interni* (uno e non più di uno alla volta), che corrispondano agli "stati mentali" dell'essere umano. Tali stati saranno in numero finito, poiché (usando la parole dello stesso Turing) "se ammettessimo un'infinità di stati mentali, alcuni di essi sarebbero 'arbitrariamente prossimi', e sarebbero quindi confusi". Il limitarsi ad un numero finito di stati non costituisce tuttavia un vincolo, in quanto "l'uso di stati mentali più complicati può essere evitato scrivendo più simboli sul nastro". Siano allora q_0, q_1, \dots, q_m gli stati che una generica MT può assumere. Nella rappresentazione grafica indicheremo sotto la testina di lettura/scrittura lo stato della macchina nella fase di calcolo rappresentata. Definiamo *configurazione* di una MT in una data fase di calcolo la coppia costituita dallo stato interno che essa presenta in quel momento e dal simbolo osservato dalla testina (la configurazione della macchina rappresentata nella fig. 2.1 è dunque (q_i, s_j)).

Una MT può dunque eseguire operazioni consistenti in spostamenti della testina lungo il nastro, scrittura e cancellazione di simboli, mutamenti dello stato interno. Scomponiamo tali operazioni in un numero di *operazioni atomiche*, tali da non poter essere ulteriormente scomposte in operazioni più semplici. Nel tipo di macchina descritto ogni operazione può essere scomposta in un numero finito delle operazioni seguenti: (1) sostituzione del simbolo osservato con un altro simbolo (eventualmente con s_0 ; in tal caso si ha la cancellazione del simbolo osservato), e/o (2) spostamento della testina su di una delle celle immediatamente attigue del nastro. Ognuno di tali atti può inoltre comportare (3) un cambiamento dello stato interno della macchina. Nella sua forma più generale, ogni operazione atomica dovrà quindi consistere di un operazione di scrittura e/o di uno spostamento atomico, ed eventualmente di un mutamento di stato. Indicheremo d'ora in avanti rispettivamente con le lettere S, D e C il fatto che una macchina debba eseguire uno spostamento di una cella verso sinistra, di una cella verso destra, oppure non debba eseguire alcuno spostamento (dove C sta per "centro"). Grazie a ciò potremo rappresentare ogni operazione atomica mediante una terna, il primo elemento della quale starà ad indicare il simbolo che deve essere scritto sulla cella osservata, il secondo quale spostamento deve essere eseguito (S, D o C), il terzo infine lo stato che la macchina deve assumere alla fine dell'operazione. Ad esempio, la terna:

$$s_i \ S \ q_j$$

significa che la macchina deve scrivere il simbolo s_i sulla cella osservata, spostarsi di una cella a sinistra, ed assumere infine lo stato q_j . Invece la terna:

$$s_0 \ C \ q_p$$

significa che la macchina deve cancellare il simbolo osservato, non eseguire alcun movimento ed assumere lo stato q_p .

Ogni singola MT è "attrezzata" per eseguire un tipo di calcolo specifico, dispone cioè di una serie di regole, di istruzioni, che le permettano di eseguire il compito per il quale è stata progettata. In un calcolo algoritmico ogni passo deve essere completamente determinato dalla situazione precedente. Nel caso di un calcolatore umano, ogni sua mossa deve dipendere esclusivamente dal ricordo delle operazioni già eseguite e dai simboli che egli può osservare. Analogamente, in una MT, poiché in ogni fase del calcolo la macchina "sa" soltanto in quale stato si trova e quale è il simbolo sulla cella osservata del nastro (cioè sa quale è la sua configurazione corrente), e poiché ogni operazione può essere scomposta in operazioni atomiche, allora ogni generica istruzione avrà la forma seguente:

$$\langle \text{configurazione} \rangle \rightarrow \langle \text{azione atomica} \rangle.$$

In altri termini, ogni istruzione deve specificare quale operazione atomica deve essere eseguita a partire da una determinata configurazione. Un esempio di istruzione è:

$$q_i s_j \rightarrow s_{j'} D q_{i'},$$

che deve essere interpretata come segue: qualora la macchina si trovi nello stato q_i ed il simbolo osservato sia s_j , allora il simbolo $s_{j'}$ dovrà essere scritto sul nastro al posto di s_j , la testina dovrà spostarsi di una cella verso destra e la macchina dovrà assumere lo stato $q_{i'}$. In generale, poiché ogni configurazione è rappresentabile mediante una coppia, ed ogni operazione atomica mediante una terna, un'istruzione (in cui di solito il simbolo "→" viene omesso e considerato sottinteso) avrà la forma di una *quintupla*, i primi due elementi della quale (uno stato interno ed un simbolo dell'alfabeto) indicano la configurazione di partenza, mentre gli ultimi tre elementi specificano l'operazione che deve essere eseguita. Le istruzioni di cui dispone ogni singola MT per eseguire il calcolo per il quale è stata progettata avranno quindi la forma di un opportuno insieme finito di quintuple (che verrà detto la *tavola* di quella MT). Una volta fissato l'alfabeto, ciò che caratterizza ogni singola MT rispetto a tutte le altre è appunto la tavola delle sue quintuple. Affinché un insieme di quintuple costituisca la tavola di una MT è indispensabile che venga rispettata la seguente condizione: poiché il calcolo deve essere deterministico, a partire da una singola configurazione non devono essere applicabili istruzioni diverse. Ciò corrisponde alla condizione che, nella tavola di una macchina, non possano comparire più quintuple con i primi due elementi uguali.

Affinché il calcolo possa terminare, è necessario che ad alcune delle configurazioni possibili non corrisponda alcuna quintupla, altrimenti, qualunque fosse il risultato di una mossa, esisterebbe sempre un'altra mossa che ad essa dovrebbe far seguito. Chiameremo tali configurazioni *configurazioni finali*. Data una MT, è sempre possibile costruirne un'altra che esegua lo stesso calcolo, per la quale esista uno specifico stato interno che compare in tutte e sole le configurazioni finali. Chiameremo tale stato *stato finale*, e stabiliremo convenzionalmente di riservare ad esso il simbolo q_0 . Data una MT generica, per trasformarla in una che abbia q_0 come stato finale si proceda nel modo seguente. Sia (q_n, s_m) una generica configurazione finale della macchina di partenza. La tavola della nuova macchina si ottiene aggiungendo tutte le quintuple del tipo:

$$q_n s_m s_m C q_0.$$

In tutti i casi in cui la macchina di partenza giungeva in uno stato finale, la nuova macchina farà un'ulteriore mossa, assumendo lo stato q_0 (e lasciando inalterato tutto il resto).

I dati vengono forniti a una MT sotto forma di una sequenza finita di simboli dell'alfabeto scritti sul nastro prima dell'inizio del calcolo. Chiameremo *input* tale sequenza di simboli. Il risultato è costituito da ciò che è scritto sul nastro al momento della fermata, e ciò costituisce l'*output* del calcolo. Stabiliamo convenzionalmente che all'inizio del calcolo la testina debba essere collocata in *posizione standard*, vale a dire in corrispondenza del primo simbolo a sinistra dell'input; inoltre lo stato interno della macchina debba essere q_1 .

Vediamo un semplice esempio di MT. Si consideri l'alfabeto $\Sigma \equiv \{| \}$, composto come unico simbolo da una barra verticale. Definiamo una macchina che, presa come input una successione di barre consecutive, restituisca come output tale successione aumentata di un elemento. A tal fine è sufficiente disporre del solo stato interno q_1 (oltre allo stato finale q_0); la tavola della macchina sarà la seguente:

q_1			D	q_1
q_1	s_0		C	q_0

Secondo le convenzioni stabilite, alla partenza la testina deve essere collocata sul primo simbolo a sinistra dell'input, e lo stato di partenza deve essere q_1 (fig. 2.2).

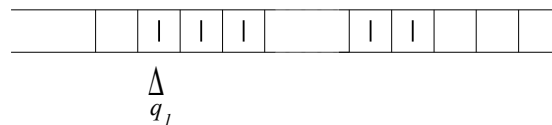


Fig. 2.2

Fintanto che la testina trova celle segnate con | allora, in virtù della prima quintupla, viene riscritto | sulla cella osservata (cioè, vengono lasciate le cose come stanno), e la testina si sposta a destra di una cella mantenendo lo stato q_1 (fig. 2.3).

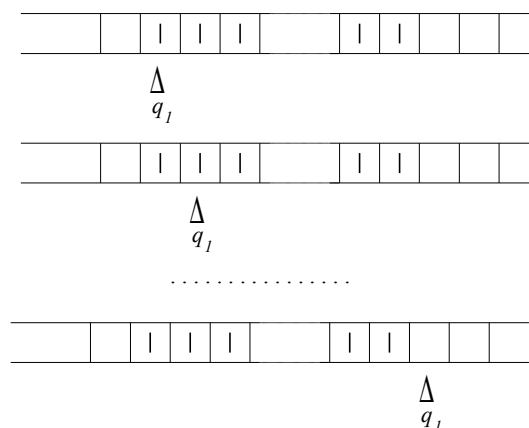


Fig. 2.3

Quando la testina incontra una cella vuota viene attivata la seconda quintupla, in virtù della quale la macchina deve segnare con una barra la cella osservata, non eseguire alcuno spostamento, ed assumere lo stato finale q_0 (fig. 2.4).

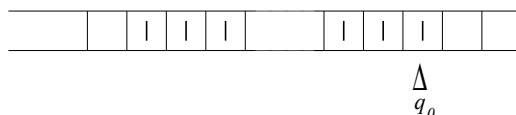


Fig. 2.4

Sin qui abbiamo considerato MT che eseguono calcoli su dati generici. Vediamo ora come si possano codificare i numeri naturali in modo da definire MT che calcolino funzioni aritmetiche. Utilizziamo come alfabeto $\Sigma \equiv \{| \}$. I numeri naturali vengono codificati come segue. Al numero 0 viene fatta corrispondere la sequenza composta da una sola barra. In generale, ogni numero n viene codificato da una sequenza di $n + 1$ barre. Una n -pla di numeri naturali (k_1, \dots, k_n) viene codificata sul nastro scrivendo la sequenza di barre corrispondente ad ogni k_i (con $1 \leq i \leq n$), e lasciando una cella vuota come separatore tra ognuna di tali sequenze. Ad esempio, la terna $(4, 1, 0)$ viene codificata come in fig. 2.5.

Diremo che una macchina M_φ computa una funzione aritmetica φ ad n argomenti (con $n \geq 1$) sse quanto segue vale per ogni n -pla (x_1, \dots, x_n) di numeri naturali. Sia (x_1, \dots, x_n) codificata nel modo sopra descritto e collocata in posizione standard rispetto alla testina (essendo vuota ogni altra cella del nastro). Allora $\varphi(x_1, \dots, x_n) = y$ sse, al termine del calcolo, l'output di M_φ è costituito dalla codifica di y .



Fig. 2.5

Diremo che una funzione φ è T -computabile sse esiste una MT M_φ che la computa.

2.3. Esempi di macchine di Turing

In questo paragrafo presentiamo alcuni esempi di MT che eseguono semplici calcoli. Eccetto i casi in cui sia indicato esplicitamente, ognuna delle macchine ha $\Sigma \equiv \{| \}$, e, al momento dell'avvio, la testina deve essere collocata sulla prima cella a sinistra dell'input. Lo stato iniziale è q_1 .

1. MT che esegue l'addizione di due numeri. Input: codifica di una coppia di numeri naturali.



La macchina riempie con una barra la cella vuota che separa i due numeri dell'input, dopo di che cancella le due barre finali del secondo numero.

2. MT che raddoppia il numero di | consecutive che le viene dato in input. Input: sequenza di |.

q_1		s_0	D	q_2	q_2			D	q_2
q_2	s_0	s_0	D	q_3	q_3			D	q_3
q_3	s_0		D	q_4	q_4	s_0		S	q_5
q_5			S	q_5	q_5	s_0	s_0	S	q_6
q_6			S	q_7	q_6	s_0	s_0	C	q_0 (*)
q_7			S	q_7	q_7	s_0	s_0	D	q_1

La macchina cancella la prima barra a destra dell'input, dopo di che si colloca a destra dell'input (lasciando una cella vuota come separatore), e stampa due barre. Torna quindi indietro, e ripete l'operazione sino a che tutte le barre dell'input sono state cancellate.

3. MT che calcola la funzione $\varphi(x)=2x$. Input: codifica di un numero naturale. La tavola è la stessa della macchina precedente, in cui la quintupla (*) è stata sostituita dalle tre quintuple seguenti:

q_6	s_0	s_0	D	q_8	q_8	s_0	s_0	D	q_8
q_8		s_0	C	q_0					

Poiché la codifica di un numero n è costituita da $n+1$ barre, la codifica di $2n$ è costituita da $2n+1 = 2(n+1)-1$ barre. Quindi questa macchina, dopo avere raddoppiato il numero delle barre in input, cancella una barra e si ferma.

4. MT che calcola la funzione il cui valore è 1 se l'argomento è un numero pari, 0 se l'argomento è dispari. Input: codifica di un numero naturale.

q_1		s_0	D	q_2	q_2		s_0	D	q_1
q_1	s_0		C	q_0	q_2	s_0		D	q_3
q_3	s_0		C	q_0					

La macchina cancella successivamente tutte le barre dell'input, assumendo alternativamente gli stati q_1 e q_2 . Se, quando l'intero input è stato cancellato, lo stato è q_1 (il che accade se l'input era la codifica di un numero dispari), la macchina stampa una barra (la codifica di 0). Altrimenti, se lo stato è q_2 (se cioè l'input era pari), stampa due barre (la codifica di 1). Dopo di che si ferma.

5. MT che calcola la differenza tra due numeri naturali. Input: codifica di una coppia di numeri naturali, di cui il primo maggiore o uguale del secondo. All'inizio del calcolo la testina deve essere collocata sulla prima cella *a destra* dell'input.

q_1		s_0	S	q_2	q_2	s_0	s_0	C	q_0
q_2			S	q_3	q_3			S	q_3
q_3	s_0	s_0	S	q_4	q_4	s_0	s_0	S	q_4
q_4		s_0	D	q_5	q_5	s_0	s_0	D	q_5

q_5			D	q_6		q_6			D	q_6
q_6	s_0	s_0	S	q_1						

La macchina cancella una barra dalla codifica del secondo numero in input. Dopo di che cancella alternativamente una barra dalla codifica del secondo e del primo numero in input, sino a che la codifica del secondo numero non è stata cancellata completamente.

6. MT che controlla se una sequenza di parentesi è bilanciata, se cioè, per ogni parentesi aperta, esiste una parentesi chiusa corrispondente. $\Sigma \equiv \{ (,), X \}$. Input: una sequenza di "(" e ")" (senza celle vuote in mezzo). Output: una sequenza di sole X se le parentesi dell'input erano bene accoppiate; altrimenti, una sequenza di simboli di Σ comprendente "(" oppure ")".

q_1	((D	q_1		q_1	X	X	D	q_1
q_1))	S	q_2		q_2	X	X	S	q_2
q_2	(X	D	q_3		q_3	X	X	D	q_3
q_3)	X	D	q_1		q_1	s_0	s_0	C	q_0
q_2	s_0	s_0	C	q_0						

La macchina percorre l'input da sinistra verso destra mantenendosi nello stato q_1 finché non trova una ")"; allora passa in q_2 e torna in dietro fino alla prima "(" che incontra, che sostituisce con una X ; assume quindi lo stato q_3 e torna a destra, a sostituire con X anche la ")" precedentemente individuata; dopo di che, torna in q_1 e ripete da capo l'operazione; si ferma non appena la testina incontra una cella vuota.

2.4. La tesi di Church

Nel 1936 il logico americano Alonzo Church, in seguito alle sue ricerche sulla computabilità effettiva, propose di identificare la classe delle funzioni calcolabili mediante un algoritmo (o funzioni effettivamente calcolabili) con una particolare classe di funzioni aritmetiche, detta in seguito classe delle *funzioni ricorsive generali* (Church 1936). Tale identificazione divenne nota col nome di *Tesi di Church*. È possibile dimostrare l'equivalenza tra la classe delle funzioni ricorsive generali e la classe delle funzioni T-computabili, in quanto ogni funzione T-computabile è ricorsiva generale, e viceversa (Turing 1937). La Tesi di Church può quindi essere formulata come segue:

una funzione è effettivamente calcolabile sse è T-computabile.

Che ogni funzione ricorsiva generale (o T-computabile) sia effettivamente computabile segue direttamente e in modo ovvio dalla definizione di T-computabilità e di MT. Ciò che invece è interessante nella Tesi di Church è l'implicazione inversa, secondo la quale ogni procedimento algoritmico è riconducibile alla ricorsività generale. Algoritmo e funzione computabile in modo effettivo sono concetti intuitivi, non specificati in modo rigoroso, per cui non è possibile una dimostrazione formale di equivalenza con il concetto di funzione ricorsiva generale. La Tesi di Church non è dunque una congettura che, in linea di principio, potrebbe un giorno diventare un

teorema. Tuttavia, la nozione intuitiva di funzione computabile in modo effettivo è contraddistinta da un insieme di caratteristiche (quali determinismo, finitezza di calcolo, eccetera) che possiamo considerare in larga misura "oggettive". Questo fa sì che sia praticamente sempre possibile una valutazione concorde nel decidere se un dato procedimento di calcolo debba essere considerato algoritmico o meno. Quindi, almeno in linea di principio, è ammissibile che venga "scoperto" un controesempio alla Tesi di Church; è ammissibile cioè che venga individuata una funzione effettivamente calcolabile secondo questi parametri intuitivi, la quale non sia allo stesso tempo ricorsiva generale. In questo paragrafo esporremo le ragioni per cui si ritiene improbabile che un evento del genere si verifichi.

Prima di procedere, è opportuno un chiarimento. Le funzioni ricorsive generali (o T-computabili) sono esclusivamente funzioni aritmetiche. Ciò potrebbe sembrare troppo restrittivo, in quanto esistono algoritmi definiti su oggetti diversi dai numeri naturali. Vi sono algoritmi che stabiliscono se un certo oggetto matematico appartiene a un dato insieme o meno. Ve ne sono altri che eseguono operazioni simboliche sulle espressioni di un sistema formale, stabilendo ad esempio se una data formula gode o meno di una certa proprietà. In logica matematica tuttavia sono state sviluppate tecniche mediante le quali algoritmi di tipo diverso, quali quelli sopra citati, possono essere ricondotti a funzioni aritmetiche. I numeri naturali infatti possono essere utilizzati per rappresentare, mediante opportune codifiche, dati o informazioni di varia natura, purché di tipo discreto (si ricordi a questo proposito il par. 1.4). Nel caso delle MT, si può dimostrare che ogni macchina con un alfabeto Σ di simboli finito può essere trasformata in una macchina equivalente che calcola una funzione aritmetica T-computabile come definita nel par. 2.

Seguendo in parte l'analisi del logico S.C. Kleene (1952, §62), raccoglieremo gli argomenti a favore della Tesi di Church in due gruppi, (a) e (b).

(a) Il primo gruppo di argomenti poggia su quella che si può chiamare *evidenza euristica*. Rientra in questo gruppo la constatazione che, per ogni singola funzione calcolabile che sia stata esaminata, è sempre stato possibile dimostrare la sua appartenenza alla classe delle funzioni ricorsive generali. Analogamente, si è dimostrato che le operazioni note per definire funzioni effettivamente calcolabili a partire da altre funzioni effettivamente calcolabili conservano la ricorsività generale. Tale indagine è stata condotta per un grande numero di funzioni, di classi di funzioni e di operazioni. Infine, i vari metodi tentati per costruire funzioni effettivamente calcolabili che non fossero ricorsive generali hanno condotto tutti al fallimento, nel senso che le funzioni ottenute erano tutte a loro volta ricorsive generali, oppure non erano calcolabili in modo effettivo.

(b) Nel secondo gruppo di argomenti viene considerata *l'equivalenza delle diverse formulazioni* proposte. Abbiamo accennato al fatto che numerosi studiosi hanno lavorato ad una definizione rigorosa del concetto di algoritmo. Ebbene, tutti i tentativi che furono elaborati per caratterizzare in modo rigoroso la classe di tutte le funzioni effettivamente computabili si rivelarono equivalenti, nel senso che la classe di funzioni ottenuta era sempre la classe delle funzioni ricorsive generali. Ciò che è particolarmente rilevante ai fini di una "corroborazione" della Tesi di Church è la diversità degli strumenti e dei concetti impiegati nelle diverse formulazioni. In molti casi tali formulazioni traggono la loro origine da concetti matematici preesistenti. Nel caso della *ricorsività generale di Herbrand-Gödel* si prendono le mosse dal concetto di sistema di equazioni, nella λ -*ricorsività* di Church (1936) si parte dall'idea di un calcolo di sole funzioni, il λ -*calcolo*. Schönfinkel (1924) e Curry (1929, 1930, 1932) elaborarono il

cosiddetto *calcolo dei combinatori*. Ad E. Post (1943, 1946) è dovuto l'approccio basato sui *sistemi normali* o *canonici*. Negli anni cinquanta, il logico sovietico A. A. Markov (1951, 1954) propose un'ulteriore formulazione tramite quelli che vennero poi detti appunto *algoritmi di Markov*. Tale indipendenza dalla formulazione utilizzata è ovviamente un forte elemento a favore della Tesi di Church.

Un posto a sé merita l'apporto all'evidenza della Tesi di Church fornito dall'analisi del concetto di calcolo algoritmico compiuta da Turing. Il concetto di *macchina di Turing* si distingue dalla maggior parte degli approcci sopra elencati in quanto non si tratta di un concetto matematico elaborato per ragioni diverse e proposto in un secondo tempo come formulazione rigorosa del concetto di algoritmo, quanto piuttosto di un tentativo diretto di costruire un modello dell'attività di un essere umano che esegue un calcolo di tipo deterministico. Storicamente, fu proprio l'analisi di Turing ad aumentare notevolmente il convincimento della correttezza della Tesi di Church.

Questo tipo di approccio al problema della computabilità effettiva ha condotto alcuni studiosi a considerare la Tesi di Church come una sorta di "legge empirica" piuttosto che come un enunciato a carattere logico-formale". Il logico Emil Post, il quale, nel 1936, propose un concetto di macchina calcolatrice in parte analogo a quello sviluppato da Turing, sottolineava il suo disaccordo da chi tendeva ad identificare la Tesi di Church con un assioma o una mera definizione. Essa dovrebbe piuttosto essere considerata, afferma Post, una ipotesi di lavoro, che, se opportunamente corroborata, dovrebbe assumere il ruolo di una "legge naturale", una "fondamentale scoperta circa le limitazioni del potere matematizzante dell'*Homo sapiens*" (Post 1936, pag. 105).

La Tesi di Church può essere utilizzata come "scorciatoia" nelle dimostrazioni: per dimostrare che esiste una MT che svolge un certo compito, si fa vedere che c'è un algoritmo intuitivo per quel compito e poi, appellandosi appunto alla Tesi di Church, si conclude che esiste una MT equivalente. Vedremo alcuni esempi del genere nella terza parte di questa dispensa. Ovviamente, ai fini di una dimostrazione rigorosa e completa ciò non è sufficiente, ed è necessario dimostrare in maniera diretta l'esistenza della MT desiderata.

2.5. La macchina di Turing universale e il calcolatore di von Neumann

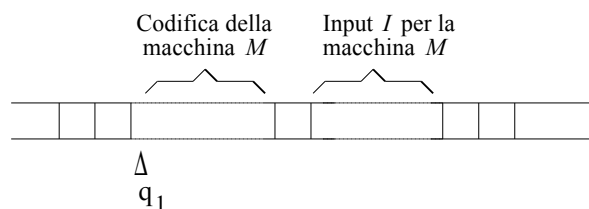
L'interesse delle MT per la teoria delle macchine calcolatrici e per l'informatica risiede innanzi tutto nel fatto che le MT sono un modello del calcolo algoritmico, di un tipo di calcolo quindi che è, in linea di principio, automatizzabile, eseguibile cioè da un dispositivo meccanico. Ogni MT è il modello astratto di un calcolatore - astratto in quanto prescinde da alcuni vincoli di limitatezza cui i calcolatori reali devono sottostare; ad esempio, la memoria di una MT (vale a dire il suo nastro) è potenzialmente estendibile all'infinito (anche se, in ogni fase del calcolo, una MT può sempre utilizzarne solo una porzione finita), mentre un calcolatore reale ha sempre limiti ben definiti di memoria.

Vi sono altre ragioni che giustificano l'analogia tra MT e moderni calcolatori digitali. Sino ad ora abbiamo considerato MT che sono in grado di effettuare un solo tipo di calcolo, sono cioè dotate di un insieme di quintuple che consente loro di calcolare una singola funzione (ad esempio la somma, o il prodotto). Esiste tuttavia la possibilità di definire una MT, detta *Macchina di Turing Universale* (d'ora in poi MTU), che è in grado di simulare il comportamento di ogni altra MT. Ciò è reso possibile dal fatto che le quintuple di ogni MT possono essere rappresentate in maniera tale da poter essere scritte sul nastro di una MT. Abbiamo accennato al fatto (par. 4) che

i numeri naturali possono essere utilizzati per codificare informazioni di tipo discreto di diverso genere. In particolare, è possibile sviluppare un metodo per codificare mediante numeri naturali la tavola di una qualsiasi MT. In questo modo, il codice di una MT può essere scritto sul nastro e dato in input a un'altra MT. Inoltre, tale codifica può essere definita in maniera tale che, dato un codice, si possa ottenere la tavola corrispondente e viceversa mediante un procedimento algoritmico (una codifica che goda di questa proprietà è detta una *codifica effettiva*).

Si può dimostrare che esiste un MT (la MTU appunto) che, preso in input un opportuno codice effettivo delle quintuple di un'altra macchina, ne simula il comportamento. In altre parole, la MTU è una macchina il cui input è composto da due elementi (si veda la parte superiore di fig. 2.6): 1. la codifica della tavola di una MT (chiamiamola M), 2. un input per M (chiamiamolo I). Per ogni M e per ogni I , la MTU "decodifica" le quintuple di M , e le applica ad I , ottenendo lo stesso output che M avrebbe ottenuto a partire da I (come schematizzato nella parte inferiore di fig. 2.6).

Inizio del calcolo:



Fine del calcolo:

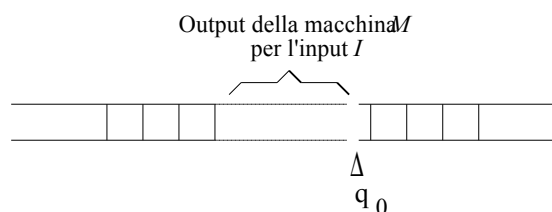


Fig. 2.6

Poiché la MTU è in grado di simulare il comportamento di qualsiasi MT, allora essa, in virtù della Tesi di Church, è in grado di calcolare qualsiasi funzione che sia calcolabile mediante un algoritmo. Ciò che caratterizza la MTU rispetto alle MT usuali è costituito dal fatto di essere una macchina calcolatrice *programmabile*. Mentre infatti le normali macchine di Turing eseguono un solo programma, che è "incorporato" nella tavola delle loro quintuple, la MTU assume in input il programma che deve eseguire (cioè, la codifica delle quintuple della MT che deve simulare), e le quintuple che compongono la sua tavola hanno esclusivamente la funzione di consentirle di interpretare e di eseguire il programma ricevuto in input.

Un'altra caratteristica fondamentale della MTU è dato dal tipo di trattamento riservato ai programmi. La MTU tratta i programmi (cioè la codifica delle quintuple della MT da simulare) e i dati (l'input della MT da simulare) in maniera sostanzialmente analoga: essi vengono memorizzati sullo stesso supporto (il nastro), rappresentati utilizzando lo stesso alfabeto di simboli ed elaborati in modo simile. Queste caratteristiche sono condivise dagli attuali calcolatori, che presentano la struttura nota come *architettura di von Neumann* (dal nome dello scienziato di origine ungherese John von Neumann che la ideò). La struttura di un calcolatore di von Neumann è raffigurata,

molto schematicamente, nella fig. 2.7. Un dispositivo di input e un dispositivo di output permettono di accedere dall'esterno alla memoria del calcolatore, consentendo, rispettivamente, di inserirvi e di estrarne dei dati. Le informazioni contenute in memoria vengono elaborate da una singola unità di calcolo (detta CPU - *Central Processing Unit*), che opera sequenzialmente su di essi. La caratteristica più importante della macchina di von Neumann è costituita dal fatto che sia dati che programmi vengono trattati in modo sostanzialmente omogeneo, ed immagazzinati nella stessa unità di memoria. Così, quando un programma deve essere eseguito, l'unità di calcolo lo reperisce in memoria, e lo applica quindi ai dati, anch'essi conservati in memoria. Questo consente una grande flessibilità al sistema. Ad esempio, poiché dati e programmi sono oggetti di natura omogenea, è possibile costruire programmi che prendano in input altri programmi e li elaborino, e che producano programmi in output. Queste possibilità sono ampiamente sfruttate negli attuali calcolatori digitali, e da esse deriva gran parte della loro potenza e della loro facilità d'uso (ad esempio, un compilatore o un sistema operativo sono essenzialmente programmi che operano su altri programmi). In questo senso limitato, un calcolatore di von Neumann costituisce una realizzazione concreta della MTU (e la memoria dati/programmi può essere considerata l'equivalente del nastro della MTU). Anche la potenza computazionale è la stessa, nel senso che, se lo si suppone dotato di una memoria e di tempi di calcolo virtualmente illimitati, un calcolatore di von Neumann è in grado di calcolare tutte le funzioni computabili secondo la Tesi di Church (per questo si dice che una macchina di von Neumann è un *calcolatore universale*). La MTU costituisce quindi un modello astratto degli attuali calcolatori digitali (elaborato prima della loro realizzazione fisica).

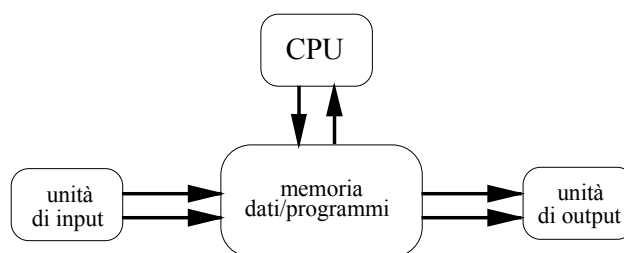


Fig. 2.7

Si noti che anche i vari linguaggi di programmazione sviluppati in informatica consentono di definire tutte e sole le funzioni ricorsive generali (purché, ovviamente, si supponga che tali linguaggi "girino" su calcolatori ideali con memoria e tempi di calcolo illimitati). Questo vale sia per i linguaggi di programmazione di alto livello (come PASCAL, FORTRAN, BASIC, VISUAL BASIC, C, C++, JAVA, LISP, PROLOG, eccetera), sia per i vari tipi di *codice assembler*. In questo senso, tali linguaggi possono essere considerati analoghi ai vari formalismi citati al punto (b) del par. 4.

2.6. Il problema della fermata

La tesi di Church ha molte importanti conseguenze dal punto di vista teorico. Dalla sua validità consegue l'esistenza di problemi che non sono risolvibili mediante un algoritmo (come si ricorderà, stabilire se tutti i problemi matematici possono essere in linea di principio risolti con un algoritmo era stata una delle motivazioni principali per lo studio rigoroso del concetto di algoritmo). In particolare, si può dimostrare che non è

effettivamente decidibile il *problema della fermata* (*halting problem*) per le MT, cioè il problema di stabilire se, per ogni MT M e per ogni input I , M con input I termina il suo calcolo o meno. Va precisato innanzi tutto che il fatto che la tavola di una MT comprenda almeno una configurazione finale è una condizione necessaria ma non sufficiente perché la macchina termini il calcolo. Si consideri ad esempio la MT seguente:

$$\begin{array}{ccccc} q_1 & s_0 & s_0 & D & q_1 \\ q_1 & | & s_0 & C & q_2 \end{array}$$

La coppia (q_2, s_0) costituisce una configurazione finale; se tuttavia questa macchina viene attivata col nastro completamente vuoto, il suo calcolo andrà avanti all'infinito.

L'indcidibilità del problema della fermata comporta che non esista alcun algoritmo che, data una generica MT (o il suo codice secondo una opportuna codifica effettiva) e dato un generico input per essa, consenta di stabilire se il calcolo di quella macchina con quell'input termina o meno. Diamo qui di seguito una breve traccia intuitiva di come, partendo dalla tesi di Church, si possa giungere a questo risultato ragionando per assurdo. Data una generica macchina M , sia C_M il suo codice in base a una codifica effettiva (scritta nell'alfabeto $\Sigma \equiv \{\}$). Supponiamo, per assurdo, che il problema della fermata per le MT sia decidibile. Per la tesi di Church, questo comporta che deve esistere una certa macchina di Turing H si comporti nella maniera seguente. Per ogni macchina di Turing M e per ogni input I di M ,

$$H \text{ con input } C_M \text{ e } I \left\{ \begin{array}{l} \text{dà come output 1 se il calcolo di } M \text{ per l'input } I \text{ termina} \\ \text{dà come output 0 se il calcolo di } M \text{ per l'input } I \text{ non termina.} \end{array} \right.$$

Qualora esistesse la macchina H , allora sarebbe banale costruire un'altra macchina H' che si comporti come segue:

$$H' \text{ con input } C_M \left\{ \begin{array}{l} \text{dà come output 1 se il calcolo di } M \text{ per l'input } C_M \text{ termina} \\ \text{dà come output 0 se il calcolo di } M \text{ per l'input } C_M \text{ non termina.} \end{array} \right.$$

H' infatti calcola una funzione che è un "caso particolare" della funzione calcolata da H (in quanto C_M è un valore particolare di I). Se tuttavia esistesse H' , allora si potrebbe a sua volta costruire una macchina Z così definita:

$$Z \text{ con input } C_M \left\{ \begin{array}{l} \text{genera un calcolo che non termina se } H' \text{ con input } C_M \text{ dà come output 1} \\ \text{(cioè, se il calcolo di } M \text{ per l'input } C_M \text{ termina)} \\ \text{dà come output 0 se } H' \text{ con input } C_M \text{ dà come output 0} \\ \text{(cioè, se il calcolo di } M \text{ per l'input } C_M \text{ non termina).} \end{array} \right.$$

Per ottenere Z a partire da H' sarebbe sufficiente aggiungere alla tavola di H' alcune quintuple che facciano in modo che, se l'output di H' è 1, allora abbia origine un

calcolo che non termina (ad esempio, la testina potrebbe iniziare a spostarsi a destra sul nastro qualunque sia il simbolo osservato).

Ora, si immagini di dare in input a Z il suo stesso codice C_Z . E' facile constatare che, in base alla definizione di Z , Z con input C_Z darebbe origine a un calcolo che termina se e soltanto se il calcolo di Z per l'input C_Z non termina, il che è palesemente assurdo. Ne consegue quindi che una macchina che si comporti come H non può esistere, e che quindi, se è vera la tesi di Church, non può esistere un algoritmo che decida il problema della fermata⁶.

Da questo risultato consegue che esistono problemi i quali, neppure in linea di principio, possono essere risolti da un calcolatore. Ad esempio, non può esistere alcun programma che sia grado di stabilire in generale se un programma qualsiasi con un certo input terminerà il suo calcolo o meno. E' importante ricordare che l'indecidibilità del problema della fermata è strettamente collegata ai risultati di limitazione della logica matematica, in primo luogo i teoremi di Gödel.

2.7. Le macchine di Turing e la mente: il test di Turing

Abbiamo accennato alla tendenza ad interpretare la Tesi di Church come un'ipotesi empirica sulle capacità computazionali degli esseri umani. Su questa linea procedono alcuni sviluppi successivi del pensiero dello stesso Turing. Nel suo saggio "Macchine calcolatrici ed intelligenza" (Turing 1950) assistiamo ad una sorta di "radicalizzazione" di questo modo di intendere la Tesi di Church. Facendo riferimento a calcolatori reali, che tuttavia vengono caratterizzati in maniera analoga a macchine di Turing, Turing si dichiara fiducioso che macchine di questo tipo possano giungere a simulare, nel volgere di pochi decenni, non soltanto il "comportamento computazionale" di un essere umano, ma anche qualsiasi altra attività cognitiva umana. Turing propone di riformulare la domanda "possono pensare le macchine?" nei termini del cosiddetto *gioco dell'imitazione*. Il gioco viene giocato da tre "attori": a) un essere umano, b) una macchina calcolatrice e c) un altro essere umano, l'interrogante. L'interrogante non può vedere a) e b), non sa chi dei due sia l'essere umano, e può comunicare con loro solo in maniera indiretta (ad esempio, attraverso un terminale video e una tastiera). L'interrogante deve sottoporre ad a) e a b) delle domande, in maniera tale da scoprire, nel più breve tempo possibile, quale dei due sia l'uomo e quale la macchina. a) si comporterà in modo da agevolare c), mentre b) dovrà rispondere in modo da ingannare c) il più a lungo possibile. Invece di chiedersi se le macchine possono pensare, dice Turing, è più corretto chiedersi se una macchina possa battere un uomo nel gioco dell'imitazione, o, comunque, quanto a lungo possa resistergli. Questo "esperimento mentale" viene oggi abitualmente indicato col nome di *Test di Turing*.

Turing era decisamente troppo ottimista circa le possibili prestazioni delle macchine calcolatrici: "Credo che entro circa 50 anni sarà possibile programmare calcolatori ... per far giocare loro il gioco dell'imitazione così bene che un esaminatore medio non avrà più del 70 per cento di probabilità di compiere l'identificazione esatta dopo cinque

⁶ Nel paragrafo 2.4 abbiamo detto che si può usare la Tesi di Church come "scorciatoia" per dimostrare teoremi che possono essere dimostrati anche in maniera diretta. Si noti che *questo non è un caso del genere*: la dimostrazione dell'indecidibilità del problema della fermata dipende in maniera essenziale dall'accettazione della Tesi di Church. In particolare, *senza fare appello alla Tesi di Church*, si può dimostrare che nessuna MT può decidere il problema della fermata (per ottenere una dimostrazione rigorosa di questo fatto basta far vedere nei dettagli come, supposto per assurdo che esista la macchina H , si possano costruire le macchine H' e Z). Da questo risultato, *facendo appello alla Tesi di Church*, si può concludere che il problema della fermata non può essere deciso da nessun algoritmo.

minuti di interrogazione. Credo che la domanda iniziale, 'possono pensare le macchine?', sia troppo priva di senso per meritare una discussione. Ciò nonostante credo che alla fine del secolo l'uso delle parole e l'opinione corrente si saranno talmente mutate che chiunque potrà parlare di macchine pensanti senza aspettarsi di essere contraddetto". Ci troviamo qui di fronte ad una sorta di versione "estremista", o "radicale", della Tesi di Church, che, grosso modo, potrebbe essere formulata come segue: *ogni attività cognitiva è T-computabile* (il che non vuol dire, ovviamente, che la nostra mente funziona *come* una macchina di Turing, ma che ogni attività mentale è simulabile da un dispositivo che abbia la stessa potenza computazionale delle macchine di Turing). Seppure modificata e raffinata rispetto alla formulazione di Turing, una assunzione di questo genere è a fondamento di numerose teorie e ricerche svolte nell'ambito di quel settore di ricerca che va sotto il nome di *scienze cognitive*. Si tratta di un ambito di ricerca interdisciplinare che ha per oggetto lo studio della mente, e che raccoglie i contributi di diverse discipline quali la psicologia cognitiva, la linguistica, la filosofia, l'informatica e le neuroscienze. Ciò che accomuna le ricerche svolte nelle scienze cognitive è appunto l'ipotesi che gli strumenti di tipo computazionale possano essere in qualche misura adeguati come modelli per lo studio delle facoltà mentali. In particolare, tra le scienze cognitive, *l'intelligenza artificiale* è quel settore dell'informatica che si prefigge di elaborare programmi di calcolatore che simulino specifiche attività cognitive umane, sia allo scopo di meglio comprendere queste ultime, sia allo scopo di costruire manufatti tecnologicamente rilevanti.

2.8. Oltre von Neumann: reti neurali e calcolo parallelo

Nel corso della storia dell'informatica, sono stati proposti vari modelli alternativi al calcolatore di von Neumann. Infatti, benché siano estremamente versatili, alle macchine con architettura di von Neumann sono stati imputati dei limiti dal punto di vista informatico. In particolare, è stata criticata la netta separazione tra immagazzinamento ed elaborazione dei dati che questo tipo di architettura comporta. In un calcolatore di von Neumann memoria e unità centrale di calcolo (CPU) sono due componenti rigidamente distinte. L'unità di calcolo attinge di volta in volta ai dati contenuti nella memoria, ma quest'ultima rimane sostanzialmente passiva durante la maggior parte della durata del calcolo. Si tratta del cosiddetto problema del "collo di bottiglia" dell'architettura di von Neumann: le informazioni vengono elaborate solo quando vengono richiamate dalla CPU del sistema. Ciò comporta problemi di efficienza nello sfruttamento delle risorse computazionali.

Queste limitazioni hanno un corrispettivo anche dal punto di vista dello studio computazionale della mente. Una distinzione netta tra memorizzazione delle informazioni e loro elaborazione è difficilmente giustificabile sulla base delle conoscenze disponibili sul sistema nervoso. Nel cervello non esiste alcun dispositivo centralizzato per il controllo dell'elaborazione. Le operazioni computazionali nel sistema nervoso sembrano demandate ad un meccanismo di controllo altamente distribuito. Inoltre, non esiste una separazione netta tra dispositivi per la memorizzazione e per l'elaborazione delle informazioni. Ciò pone problemi ai modelli computazionali dell'intelligenza artificiale e delle scienze cognitive tradizionali, relativi alla mancanza di plausibilità dal punto di vista anatomico e neurofisiologico del paradigma computazionale di Turing e di von Neumann. Il punto centrale è che il cervello è un dispositivo di calcolo *altamente parallelo*. Il numero dei neuroni è di un ordine stimabile tra 10^{10} e 10^{11} , e ciascuno di essi si comporta come una singola unità

di calcolo, che lavora contemporaneamente a tutte le altre. I neuroni sono altamente interconnessi: ogni neurone ha moltissime sinapsi in entrata e in uscita, mediante le quali scambia i propri input e i propri output con gli altri neuroni. Ogni neurone esegue operazioni relativamente semplici. La complessità dei meccanismi cognitivi viene determinata dall'interazione di un grande numero di neuroni.

Su considerazioni di questo genere si è basato lo sviluppo delle cosiddette *reti neurali*, una classe di dispositivi di calcolo in parte motivati dall'intento di superare i limiti del modello di von Neumann. Si tratta di sistemi distribuiti ad alto parallelismo, ispirati, in senso lato, alle proprietà del sistema nervoso. Una rete neurale è costituita da un insieme di *unità* (che sono il corrispettivo dei neuroni), collegate tra loro da *connessioni*, che costituiscono l'analogo delle sinapsi. Ogni unità ha un certo numero di connessioni in ingresso e/o un certo numero di connessioni in uscita. Ciascuna unità costituisce un semplice processore, un singolo dispositivo di calcolo che, ad ogni fase del calcolo, riceve i propri input attraverso le connessioni in ingresso, li elabora, e invia l'output alle altre unità connesse per mezzo delle sue connessioni in uscita. In una rete tutte le unità operano in parallelo, e non esiste alcun processo di ordine "più alto", nessuna CPU che ne coordini l'attività. Il calcolo che ciascuna unità esegue è di norma molto semplice; la potenza computazionale del sistema deriva dal grande numero delle unità e delle connessioni. Nell'ambito delle scienze cognitive, sulle reti neurali si basano le teorie e i modelli di tipo connessionista. Il *connessionismo* è una tendenza nello studio computazionale della mente che ha avuto un grande sviluppo nel corso degli ultimi quindici anni, e che si è in parte contrapposta agli approcci dell'intelligenza artificiale e delle scienze cognitive tradizionali. Rispetto a queste ultime, il connessionismo è caratterizzato appunto da una maggiore attenzione per i rapporti tra attività cognitive e struttura del sistema nervoso.

E' opportuno notare tuttavia che questi sviluppi non hanno comportato un superamento dei risultati della teoria della computabilità effettiva, o una qualche forma di "falsificazione" della tesi di Church. Di fatto, tutti i modelli computazionali basati sulle reti neurali che siano stati effettivamente realizzati sono risultati riconducibili entro i limiti della ricorsività generale, nel senso che le funzioni computate da tali modelli risultano essere funzioni ricorsive generali.

Più in generale, benché le MT e i calcolatori con architettura di von Neumann siano dispositivi di calcolo di tipo strettamente sequenziale, è possibile estendere la validità della tesi di Church anche a calcoli di tipo parallelo. Rilevanti in questa direzione sono state ad esempio le ricerche di Robin Gandy (1980), che è partito dalla constatazione che il concetto di MT corrisponde ad una nozione di calcolo troppo specifica e particolare perché le possa essere ricondotto ogni tipo di dispositivo di calcolo concepibile. Ad esempio, nelle MT si assume che il calcolo proceda secondo una sequenza di passi elementari, elaborando un solo simbolo alla volta, mentre un calcolatore artificiale può procedere in parallelo, elaborando contemporaneamente un numero arbitrario di simboli. Per superare tali limitazioni, Gandy ha formulato, utilizzando strumenti di tipo insiemistico, una caratterizzazione estremamente generale del concetto di macchina calcolatrice, in cui le MT rientrano come caso particolare. Egli ha dimostrato quindi che ogni funzione calcolabile da tali macchine è ricorsiva generale, a patto che vengano rispettate alcune condizioni molto generali di finitezza (determinismo, possibilità di descrivere il calcolo in termini discreti, e così via).

Va ricordato tuttavia che, dal punto di vista applicativo, l'architettura di von Neumann (o sue varianti che non ne differiscono in maniera sostanziale) resta il modello di calcolatore di gran lunga più diffuso. Attualmente calcolatori con

architettura non di von Neumann vengono progettati e utilizzati per tipi di applicazioni specifiche.

3. Computabilità, automi e grammatiche formali

3.1 Premessa

Una delle caratteristiche più peculiari dei linguaggi umani è il loro carattere *generativo*: dati un insieme finito di espressioni di partenza (le parole o i morfemi di una lingua) e le capacità cognitive dei parlanti di generare e comprendere enunciati (capacità che a loro volta sono presumibilmente finite) il linguaggio consente di produrre e comprendere un numero praticamente illimitato di enunciati.

Si consideri ad esempio l'enunciato seguente:

(*) *Le cinquantasei lepri che ieri aggredirono il vescovo di Salerno sono fuggite in autobus verso Savona*

È molto plausibile che una frase come (*) non sia mai stata formulata da nessuno prima d'ora. Tuttavia, nonostante la sua bizzarria, non abbiamo alcuna difficoltà nel riconoscerla come una frase del tutto corretta della lingua italiana, a differenza ad esempio di:

(**) *Uno lepre cinquantasei ieri aggredendo Salerno verso fuggire autobus autobus*

che, sebbene composta interamente da parole italiane, non è una frase dell'italiano.

Il carattere generativo del linguaggio riguarda sia il piano sintattico, sia quello semantico. Sul piano sintattico noi riconosciamo (*) come una frase corretta dell'italiano anche se la incontriamo per la prima volta. Sul piano semantico, se conosciamo il significato delle parole che vi compaiono, siamo in grado di comprendere il significato di (*), per quanto stravagante e inatteso esso possa apparirci. In questa sede ci occuperemo esclusivamente degli aspetti sintattici⁷.

Argomento di questo capitolo è quel settore di ricerca in cui si sviluppano modelli formali in grado di rendere conto del carattere generativo dei sistemi linguistici, siano essi lingue naturali o linguaggi artificiali. La *teoria delle grammatiche formali* studia le proprietà di sistemi che consentono di caratterizzare linguaggi eventualmente infiniti utilizzando strumenti finiti. Si tratta di un settore di ricerca nato nell'ambito della logica matematica e della teoria della computabilità, che ha avuto importanti sviluppi in linguistica ad opera soprattutto di Noam Chomsky e ha trovato in seguito applicazione anche in informatica⁸.

3.2 Che cos'è una grammatica formale?

Una grammatica formale consente di specificare (con mezzi finiti) quali sono le espressioni sintatticamente corrette (ossia grammaticali) di un linguaggio.

⁷ Gli aspetti semantici del carattere generativo dei linguaggi vengono indagati soprattutto nell'ambito della semantica formale. Utilizzando strumenti di tipo logico e insiemistico, la semantica formale studia come il significato di espressioni sintatticamente complesse di un linguaggio dipenda dal significato delle espressioni che le compongono. Queste tecniche vengono applicate sia alle lingue naturali, sia a linguaggi artificiali come i linguaggi di programmazione. Sulla semantica formale delle lingue naturali si vedano ad esempio Casalegno (1997) e Chierchia e McConnell-Ginet (1990).

⁸ De Palma (1974) è una raccolta di articoli classici che ripercorrono la nascita di questo settore di ricerca.

Dato un *alfabeto* A , ossia un insieme finito di simboli, una **stringa** (o *parola*) su A è una n -pla ordinata di elementi A .

Se l'alfabeto è $Al = \{a, b, c, d, e, +, -, *, /, (,)\}$, un esempio di stringa su Al è la 7-pla seguente:

$$(a, d, +, (, (, a, *)$$

Nello scrivere le stringhe ometteremo le virgole e le parentesi all'inizio e alla fine, per cui scriveremo la stringa precedente più semplicemente come segue:

$$a d + ((a *$$

Altre stringhe su Al sono le seguenti:

- (i) $a + b$
- (ii) $(((+ - (*))))$
- (iii) $a b c (d) /$
- (iv) $a * (c - e)$
- (v) $(e + c) / (a + (a - b))$
- (vi) $+ a b$

Alcune di esse, cioè la (i), la (iv) e la (v), sono espressioni algebriche “corrette”, mentre le altre sono sequenze arbitrarie di simboli. Al può quindi essere usato per scrivere espressioni algebriche ben formate, le quali costituiscono un sottoinsieme proprio dell'insieme di tutte le stringhe su Al .

In generale, chiamiamo **linguaggio** L con *alfabeto* A un sottoinsieme di tutte le stringhe su A e *stringhe grammaticali* di L le stringhe che appartengono a L .

Potremmo ad esempio definire un linguaggio con alfabeto Al che includa come grammaticali espressioni algebriche come (i), (iv) e (v) ed escluda stringhe “casuali” come (ii), (iii) e (vi).

Una **grammatica** per un linguaggio L è uno strumento per specificare in modo rigoroso quali sono le stringhe grammaticali di L rispetto a tutte le stringhe che si possono scrivere con lo stesso alfabeto.

Vediamo un esempio. Consideriamo il seguente alfabeto:

$$S = \{il, un, topo, gatto, rincorre, mangia, dorme, canta\}$$

Vogliamo individuare una grammatica che consenta di caratterizzare quelle stringhe sull'alfabeto S che corrispondono a frasi sintatticamente corrette dell'italiano (come ad esempio *il topo canta* oppure *un gatto rincorre il topo*), distinguendole dalle stringhe che non corrispondono a espressioni sintatticamente corrette dell'italiano (come ad esempio *un un topo canta gatto*) o non costituiscono frasi complete (ad esempio *il gatto*).

Una grammatica del genere può essere formulata come segue:

- (1) $E \rightarrow SN SV$
- (2) $SN \rightarrow \text{Articolo Nome}$
- (3) $SV \rightarrow \text{VerbTrans SN}$
- (4) $SV \rightarrow \text{VerbIntr}$
- (5) $\text{VerbTrans} \rightarrow \text{rincorre}$
- (6) $\text{VerbTrans} \rightarrow \text{mangia}$

- (7) VerbIntr \rightarrow *dorme*
- (8) VerbIntr \rightarrow *canta*
- (9) Articolo \rightarrow *un*
- (10) Articolo \rightarrow *il*
- (11) Nome \rightarrow *topo*
- (12) Nome \rightarrow *gatto*

Le espressioni del tipo $e_1 \rightarrow e_2$ vengono chiamate **produzioni**, o **regole di produzione**, o **regole di riscrittura**. Una grammatica comprende sempre un insieme finito di produzioni. Nelle espressioni e_1 ed e_2 di ogni regola di produzione compaiono simboli dell'alfabeto del linguaggio, assieme ad altri simboli che non ne fanno parte. Questi ultimi vengono detti **simboli non terminali** della grammatica. La grammatica del nostro esempio (che d'ora in avanti chiameremo G) utilizza i simboli non terminali "E", "S", "N", "Articolo", "Nome", "VerbTrans" e "VerbIntr". Seguiremo la convenzione di scrivere i simboli non terminali in tondo e con iniziale maiuscola, per distinguerli dai simboli dell'alfabeto che sono scritti in corsivo con iniziale minuscola (d'ora in poi chiameremo i simboli dell'alfabeto anche **simboli terminali** della grammatica). I simboli non terminali possono essere interpretati come nomi metateorici di categorie sintattiche. Ad esempio, in G "E" sta per "enunciato", "SN" per "sintagma nominale", "SV" per "sintagma verbale"; il significato intuitivo degli altri simboli non terminali di G è ovvio. Data una regola di produzione $e_1 \rightarrow e_2$, in e_1 deve sempre comparire almeno un simbolo non terminale.

Data una grammatica per un certo linguaggio, vediamo come si ottengono le stringhe che fanno parte del linguaggio. Si parte da uno specifico simbolo non terminale detto **assioma** (o *simbolo iniziale*) della grammatica. Ogni grammatica deve avere uno ed un solo assioma. In G l'assioma è il simbolo E. Una regola $e_1 \rightarrow e_2$ indica che ogni volta che si trova l'espressione e_1 la si può riscrivere (cioè sostituire) con l'espressione e_2 . A partire dall'assioma si procede applicando le regole, in modo da riscrivere parti dell'espressione via via ottenuta, fino a ottenere una stringa composta esclusivamente di simboli terminali. Si dice allora che la grammatica consente di **derivare** tale stringa, e che la stringa fa parte del **linguaggio generato** dalla grammatica. Il linguaggio generato da una grammatica comprende tutte e sole le stringhe che la grammatica consente di derivare. Ad esempio G consente di derivare le stringhe *un gatto canta* e *il gatto rincorre un topo*, le quali quindi fanno parte del linguaggio generato da G , mentre non consente di derivare la stringa *un gatto canta il topo*, che quindi non fa parte del linguaggio generato da G .

Verifichiamo che la grammatica G consente di derivare *il gatto rincorre un topo*. Si parte dall'assioma:

E

In base alla regola (1) (che è l'unica che in questo momento è possibile applicare) possiamo riscrivere "E" sostituendolo con "SN SV":

SN SV

In base alla regola (2) "SN" si può riscrivere come "Articolo Nome", per cui otteniamo:

Articolo Nome SV

In base alla regola (10) “Articolo” si può riscrivere con il simbolo terminale “*il*”, per cui otteniamo:

il Nome SV

E così via. Riportiamo tutti i passaggi che consentono di ottenere *il gatto rincorre un topo* a partire dall’assioma E.

1) E	assioma
2) SN SV	da 1) per mezzo della regola (1)
3) Articolo Nome SV	da 2) per mezzo della regola (2)
4) <i>il</i> Nome SV	da 3) per mezzo della regola (10)
5) <i>il gatto</i> SV	da 4) per mezzo della regola (12)
6) <i>il gatto</i> Verb Trans SN	da 5) per mezzo della regola (3)
7) <i>il gatto rincorre</i> SN	da 6) per mezzo della regola (5)
8) <i>il gatto rincorre</i> Articolo Nome	da 7) per mezzo della regola (2)
9) <i>il gatto rincorre un</i> Nome	da 8) per mezzo della regola (9)
10) <i>il gatto rincorre un topo</i>	da 9) per mezzo della regola (11)

I passi 1)-10) costituiscono una **derivazione** in G della stringa *il gatto rincorre un topo*. Il processo di derivazione non è deterministico: ad ogni passo si può scegliere tra le varie produzioni disponibili quale applicare.

Per formulare le grammatiche in maniera più sintetica si adotta di solito la seguente convenzione. Quando si hanno più regole del tipo $A \rightarrow B_1, A \rightarrow B_2, \dots, A \rightarrow B_n$ (cioè regole nelle quali la parte a sinistra della freccia è la stessa), si scrivono sulla stessa riga nella maniera seguente: $A \rightarrow B_1 \mid \dots \mid B_n$. Adottando questa convenzione, le regole di G si possono formulare più concisamente:

$E \rightarrow \text{SN SV}$
 $\text{SN} \rightarrow \text{Articolo Nome}$
 $\text{SV} \rightarrow \text{VerbTrans SN} \mid \text{VerbIntr}$
 $\text{VerbTrans} \rightarrow \text{rincorre} \mid \text{mangia}$
 $\text{VerbIntr} \rightarrow \text{dorme} \mid \text{canta}$
 $\text{Articolo} \rightarrow \text{il} \mid \text{un}$
 $\text{Nome} \rightarrow \text{topo} \mid \text{gatto}$

È facile constatare che la grammatica G consente di generare solo un insieme finito di stringhe. Ossia, il linguaggio generato da G è un linguaggio finito. Non è difficile estenderla in maniera da generare un linguaggio infinito. Passiamo ad esempio dall’alfabeto S all’alfabeto $S' = S \cup \{e, \text{oppure}, \text{ma}\}$, aggiungiamo il simbolo non terminale “Connettivo” e le produzioni seguenti:

$E \rightarrow E \text{ Connettivo } E$
 $\text{Connettivo} \rightarrow e \mid \text{oppure} \mid \text{ma}$

Questa nuova grammatica G' consente di generare tutte le (infinite) stringhe del tipo: *un gatto dorme ma il topo canta, il gatto rincorre il topo e il topo dorme, un gatto dorme e un topo canta oppure un gatto mangia il topo*, e così via.

Un altro modo per estendere G in modo da generare un linguaggio infinito è il seguente. Passiamo dall'alfabeto S all'alfabeto $S'' = S \cup \{che\}$ e sostituiamo la seconda produzione con:

$$SN \rightarrow \text{Articolo Nome} \mid SN \text{ che } SV$$

Con questa nuova grammatica, che chiameremo G'' , diventa possibile generare tutte le (infinite) stringhe del tipo: *un gatto che dorme rincorre il topo, un gatto rincorre il topo che dorme, il topo che rincorre il gatto canta, il topo che rincorre il gatto che dorme canta*, e così via.

La possibilità di generare infinite stringhe dipende dalla presenza di produzioni in cui il simbolo non terminale a sinistra della freccia compare anche nell'espressione che si trova sulla destra (in G' si tratta del simbolo "E" nella prima delle nuove produzioni; in G'' è il simbolo "SN").

In generale, lo stesso linguaggio può essere generato da grammatiche diverse. Consideriamo ad esempio due grammatiche $G1$ e $G2$, che utilizzano entrambe l'alfabeto $\{a, b, c\}$ e impiegano i simboli non terminali S, A, B e C (dei quali S è l'assioma). Le produzioni di $G1$ e di $G2$ sono rispettivamente:

$$\begin{array}{ll} S \rightarrow S1 C & S \rightarrow A S1 \\ S1 \rightarrow A B & S1 \rightarrow B C \\ A \rightarrow aA \mid a & A \rightarrow aA \mid a \\ B \rightarrow bB \mid b & B \rightarrow bB \mid b \\ C \rightarrow cC \mid c & C \rightarrow cC \mid c \end{array}$$

Le due grammatiche sono diverse (in particolare, sono diverse le prime due produzioni), ma il linguaggio generato è lo stesso: coincide con l'insieme di tutte e sole le stringhe in cui m occorrenze di a sono seguite da n occorrenze di b , seguite a loro volta da k occorrenze di c (con $m, n, k \geq 1$). Si possono cioè derivare tutte le stringhe del tipo *aaabbbbbbcc, abbbbbcc, aabcccc, abcc*.

Si dicono **equivalenti** due grammatiche che generano lo stesso linguaggio.

3.3 La gerarchia di Chomsky

Alla fine degli anni '50 il linguista Noam Chomsky (1959) propose una classificazione delle grammatiche formali che oggi va sotto il nome di *gerarchia di Chomsky*. Egli individuò quattro classi di grammatiche, ciascuna delle quali include la successiva.

La classe più generale della gerarchia di Chomsky è costituita dalle **grammatiche di tipo 0**, nelle quali non si impone alcun vincolo sulla forma delle produzioni: può essere usata qualunque regola $e_1 \rightarrow e_2$.

La classe successiva è costituita dalle grammatiche di **tipo 1**, dette anche **grammatiche non decrescenti**. In una grammatica non decrescente non deve esserci alcuna produzione $e_1 \rightarrow e_2$ in cui e_2 sia più corta di e_1 (la lunghezza di e_1 e di e_2 è data dal numero di simboli terminali e/o non terminali che le compongono). Ad esempio, una produzione come la seguente può far parte di una grammatica di tipo 0, ma non di una grammatica di tipo 1:

$$S a \rightarrow b$$

in quanto l'espressione a sinistra della freccia è formata da due simboli mentre quella a destra è formata da un solo simbolo.

Da ciò il nome di questa classe: applicando una produzione a un'espressione quest'ultima non può mai decrescere, cioè non può mai accadere che, dopo aver applicato una regola, si ottenga un'espressione più corta di quella da cui si era partiti.

Si può facilmente constatare che le grammatiche G , G' e G'' del paragrafo precedente sono grammatiche non decrescenti⁹. Un altro esempio di grammatica non decrescente è la grammatica GnD , che ha come alfabeto l'insieme $\{a, b, c\}$, come simboli non terminali S e B , dei quali S è l'assioma, e le cui produzioni sono:

$$\begin{aligned} S &\rightarrow aSBc \mid abc \\ cB &\rightarrow Bc \\ bB &\rightarrow bb \end{aligned}$$

Il linguaggio generato da questa grammatica comprende tutte e sole le stringhe del tipo abc , $aabbcc$, $aaabbbccc$, ..., in cui n occorrenze di a sono seguite da n occorrenze di b , seguite a loro volta da n occorrenze di c (con $n \geq 1$).

Si può dimostrare che, per ogni grammatica non decrescente, esiste una grammatica ad essa equivalente in cui tutte le produzioni hanno la forma:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

dove A è un simbolo non terminale e α , β e γ sono stringhe di simboli terminali e/o non terminali. Le stringhe α e β (ma non la stringa γ) possono avere lunghezza nulla¹⁰. In base a una produzione di questo tipo, se il simbolo non terminale A è collocato tra le stringhe α e β , allora A può essere sostituito dalla stringa γ . Ossia, in questo genere di produzioni la possibilità di riscrivere un simbolo non terminale dipende dal *contesto* in cui il simbolo compare. Per questa ragione le grammatiche di tipo 1 vengono dette anche **grammatiche dipendenti dal contesto** (o grammatiche *context sensitive*).

Le grammatiche di **tipo 2** sono dette anche **grammatiche libere dal contesto** (*context free*). Tutte le produzioni di una grammatica libera dal contesto hanno la forma:

$$A \rightarrow \gamma$$

dove A è un simbolo non terminale e γ è una stringa di simboli terminali e/o non terminali. Questo schema corrisponde al caso particolare dello schema $\alpha A \beta \rightarrow \alpha \gamma \beta$ in cui α e β hanno entrambe lunghezza nulla. In questo caso il simbolo non terminale A può essere sostituito con γ a prescindere dal contesto in cui A compare. Da qui il nome di grammatiche libere dal contesto.

Un esempio di grammatica di tipo 2 è la grammatica LC , che ha come alfabeto l'insieme $\{a, b\}$, come unico simbolo non terminale l'assioma S , e le cui produzioni sono:

⁹ Si noti che una grammatica che includa tra le sue regole $Aa \rightarrow b \mid C \mid dD$ non è una grammatica di tipo 1 (cioè non decrescente). Infatti questa scrittura equivale alle tre produzioni seguenti: $Aa \rightarrow b$, $Aa \rightarrow C$, $Aa \rightarrow dD$, le prime due delle quali violano il vincolo imposto sulle produzioni di questa classe di grammatiche.

¹⁰ Se γ avesse lunghezza nulla la grammatica non sarebbe non decrescente.

$$S \rightarrow aSb$$

$$S \rightarrow ab$$

Il linguaggio generato da LC comprende tutte e sole le stringhe del tipo $ab, aabb, aaabbb, \dots$, in cui n occorrenze di a sono seguite da n occorrenze di b (con $n \geq 1$).

Anche le grammatiche G, G' e G'' del paragrafo precedente sono libere dal contesto.

L'ultima classe di grammatiche della gerarchia di Chomsky è la classe delle grammatiche di **tipo 3**, o **grammatiche lineari**. Una grammatica si dice *lineare destra* (o *regolare*) se tutte le produzioni hanno la forma:

$$A \rightarrow tB$$

oppure:

$$A \rightarrow t$$

dove A e B sono simboli non terminali e t è un simbolo terminale. Una grammatica si dice *lineare sinistra* se tutte le produzioni hanno la forma

$$A \rightarrow Bt$$

oppure:

$$A \rightarrow t$$

dove A e B sono simboli non terminali e t è un simbolo terminale. Si può dimostrare che, per ogni grammatica lineare destra, ne esiste una lineare sinistra ad essa equivalente, e viceversa.

Un esempio di grammatica lineare (destra) è la grammatica LD che ha come alfabeto l'insieme $\{a, b\}$, come simboli non terminali S e B , dei quali S è l'assioma, e le cui produzioni sono:

$$S \rightarrow aS$$

$$S \rightarrow B$$

$$B \rightarrow bB$$

$$B \rightarrow b$$

Il linguaggio generato da LD comprende tutte e sole le stringhe del tipo $ab, abb, aab, aabb, abbb, aaab, aabbb, \dots$, in cui n occorrenze di a sono seguite da m occorrenze di b (con $m, n \geq 1$).

Si può constatare agevolmente dalle definizioni che tutte le grammatiche di tipo 3 sono anche grammatiche di tipo 2, ma non viceversa (esistono cioè grammatiche di tipo 2 che *non* sono grammatiche di tipo 3), che tutte le grammatiche di tipo 2 sono anche grammatiche di tipo 1, ma non viceversa, e così via. Perciò, se indichiamo con G_i l'insieme delle grammatiche di tipo i (con i che va da 0 a 3), valgono le seguenti relazioni di inclusione stretta:

$$G_3 \subset G_2 \subset G_1 \subset G_0$$

Diremo che un **linguaggio è di tipo i** (con i che va da 0 a 3) quando esiste una grammatica di tipo i che lo genera. Si può dimostrare che tra gli insiemi dei linguaggi

valgono relazioni di inclusione analoghe a quelle tra le grammatiche: se L_i è l'insieme dei linguaggi di tipo i (con i che va da 0 a 3), valgono le seguenti relazioni di inclusione stretta:

$$L_3 \subset L_2 \subset L_1 \subset L_0$$

Pertanto, ci sono linguaggi generati da una grammatica di tipo G_i che non possono essere generati da alcuna grammatica di tipo G_{i+1} . Ad esempio, si può dimostrare che il linguaggio generato dalla grammatica dipendente dal contesto GnD che abbiamo descritto sopra non può essere generato da alcuna grammatica libera dal contesto, e che il linguaggio generato dalla grammatica libera dal contesto LC non può essere generato da alcuna grammatica lineare.

I linguaggi finiti sono un sottoinsieme proprio dei linguaggi di tipo 3: ogni linguaggio finito può essere generato da una grammatica lineare. Ad esempio, abbiamo visto che la grammatica G del paragrafo precedente è una grammatica libera dal contesto (quindi di tipo 2), ed è facile constatare che non è una grammatica lineare. Poiché, come abbiamo osservato, il linguaggio che essa genera è finito, esiste tuttavia una grammatica lineare ad essa equivalente¹¹.

3.4 Alberi di derivazione e grammatiche ambigue

In una grammatica di tipo 2 la derivazione di una stringa può anche essere rappresentata graficamente per mezzo di un albero, detto *albero sintattico*, o *albero di derivazione* (sugli alberi e la relativa terminologia si veda la finestra "Alberi"). Vediamo ad esempio l'albero di derivazione nella grammatica G della stringa *il gatto rincorre un topo* (fig. 3-1).

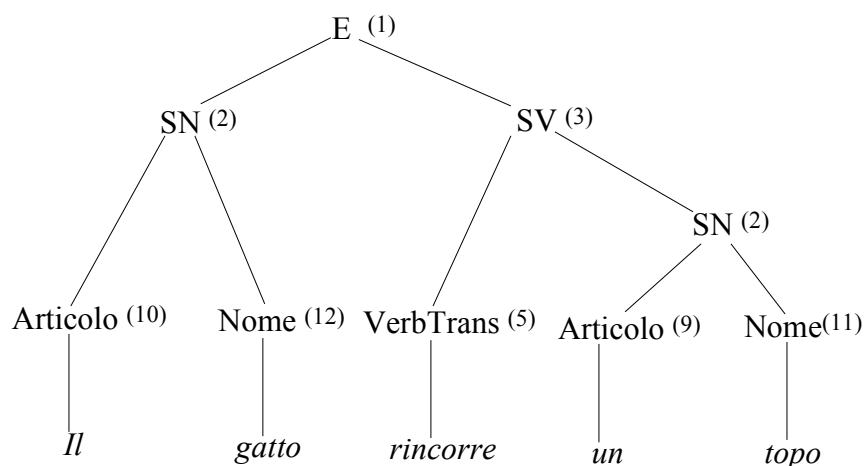


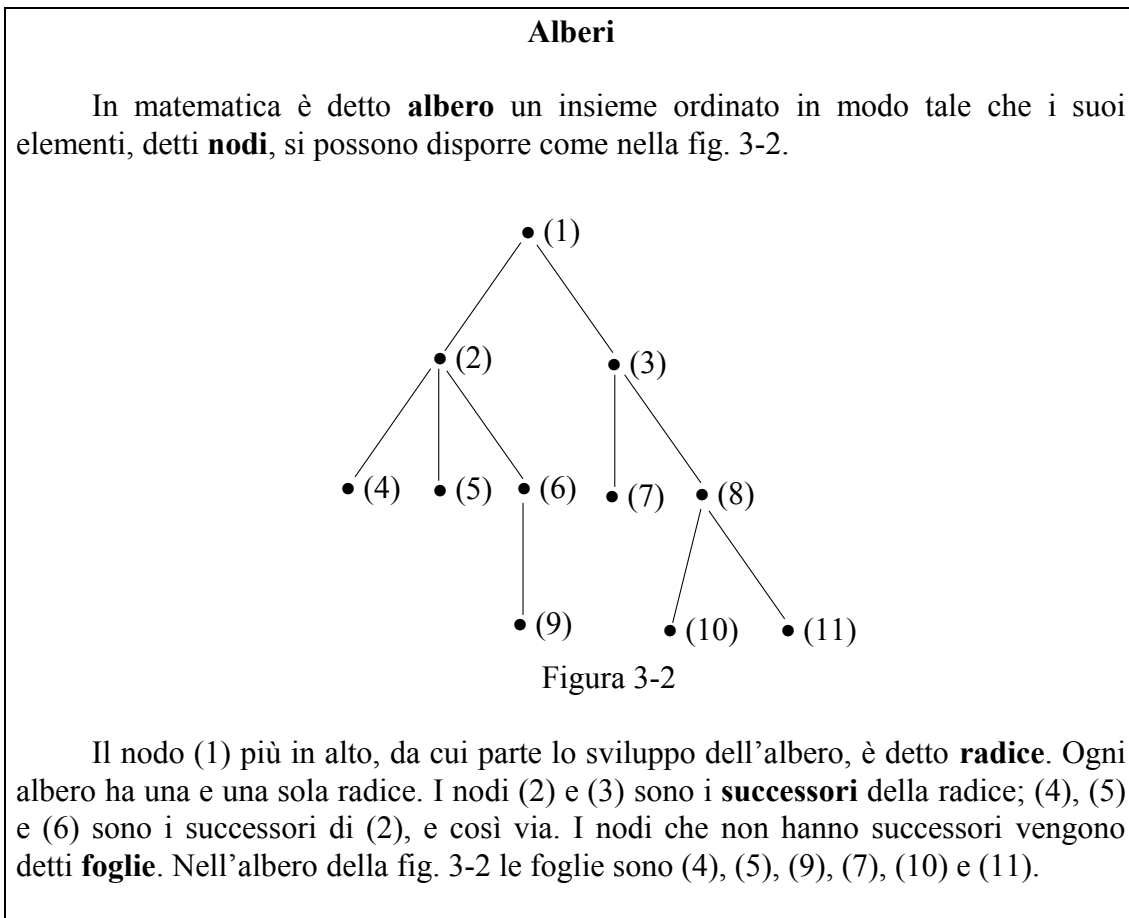
Figura 3-1

In un albero di derivazione le foglie sono simboli terminali e i nodi che non sono foglie sono simboli non terminali. In particolare, la radice deve essere l'assioma della grammatica. Leggendo le foglie da sinistra verso destra si ottiene la stringa derivata. Se

¹¹ Si veda a questo proposito l'esercizio 3.11.

un nodo S ha come successori i nodi S_1, \dots, S_n , allora nella grammatica vi è una produzione $S \rightarrow S_1 \dots S_n$ che è stata impiegata per riscrivere il simbolo S .

Nella figura 3-1, a fianco a ogni nodo abbiamo riportato il numero della produzione corrispondente.



Vi sono grammatiche che assegnano a una stessa stringa strutture sintattiche diverse: la stessa stringa può essere generata in modi diversi, attraverso derivazioni che non differiscono soltanto per l'ordine con cui sono state applicate le regole. Le **grammatiche** con questa caratteristica vengono dette **ambigue**. Nel caso di grammatiche libere dal contesto, una grammatica ambigua associa a certe stringhe alberi di derivazione differenti.

Un esempio di grammatica ambigua è la grammatica G' del paragrafo 2. Si consideri la stringa:

(***) *un gatto corre e un topo canta oppure un gatto canta*

Essa può essere analizzata nei termini dei due alberi della fig. 3-3 (questi due alberi non corrispondono a un'analisi sintattica completa: per non complicare troppo le figure, le tre stringhe *un gatto corre*, *un topo canta* e *un gatto canta* non sono state analizzate).

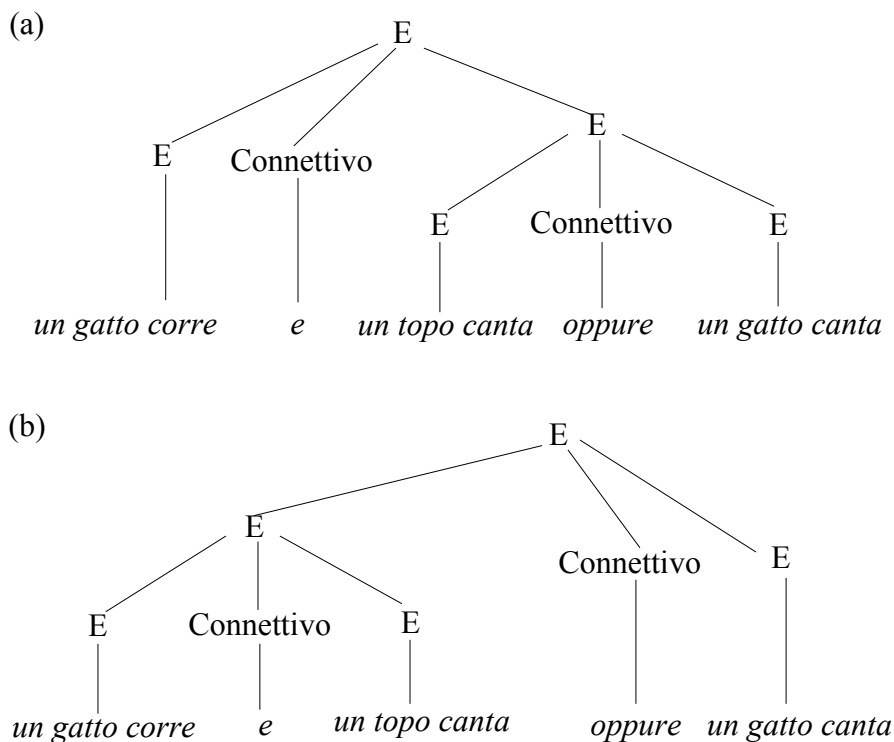


Figura 3-3

Gli alberi (a) e (b) corrispondono a derivazioni diverse in G' . In (a) il connettivo *e* collega la stringa *un gatto corre* alla stringa *un topo canta oppure un gatto canta*; in (b) il connettivo *oppure* collega la stringa *un gatto corre e un topo canta* alla stringa *un gatto canta*.

Se interpretiamo le stringhe generate da G' come enunciati, e i connettivi come connettivi proposizionali cui è associata l'usuale interpretazione verofunzionale, allora la differenza di struttura sintattica tra (a) e (b) ha anche una controparte semantica: nel caso in cui *un gatto corre* sia falso e *un gatto canta* sia vero, se (***) viene analizzata come in (a) risulta falsa, se invece viene analizzata come in (b) risulta vera.

Nel caso di sistemi artificiali come i linguaggi di programmazione l'ambiguità è un fenomeno da evitare ad ogni costo, per cui le grammatiche sviluppate per tali sistemi devono essere definite in maniera tale da non essere ambigue. Nel caso delle lingue naturali invece l'ambiguità è un dato di fatto di cui si deve inevitabilmente tenere conto (ad esempio la stringa (***) corrisponde a un enunciato perfettamente grammaticale ma ambiguo dell'italiano). Nel caso di costrutti linguistici ambigui le grammatiche formali sviluppate per lo studio delle lingue naturali devono essere in grado di generare strutture sintattiche diverse che rendano conto delle diverse analisi possibili.

3.5 Grammatiche e macchine

Alle diverse classi della gerarchia di Chomsky sono associate varie proprietà computazionali delle grammatiche e delle classi di linguaggi corrispondenti.

Distinguiamo tra due tipi di problemi rilevanti rispetto alle proprietà delle grammatiche. **Generare** un linguaggio consiste nel produrre una dopo l'altra mediante un algoritmo (o una macchina) tutte e sole le stringhe che costituiscono il linguaggio. **Riconoscere** (o **accettare**) un linguaggio è invece un problema di decisione: presa in

input una stringa di simboli dell'alfabeto, si tratta di produrre una risposta positiva se la stringa appartiene al linguaggio, e una risposta negativa in caso contrario.

Computazionalmente riconoscere un linguaggio è più difficile che generarlo: come vedremo, in generale il riconoscimento di un linguaggio richiede strumenti più potenti della sua generazione.

Insiemi ricorsivamente numerabili e insiemi ricorsivi

Intuitivamente, diciamo che un insieme è *ricorsivamente enumerabile* se e solo se esiste una MT che ne genera uno dopo l'altro tutti gli elementi; diciamo che un insieme è *ricorsivo* se esiste una MT che, preso in input (la codifica di) un oggetto, è in grado di decidere se esso appartiene all'insieme o meno.

Si può dimostrare che tutti gli insiemi ricorsivi sono anche ricorsivamente enumerabili, ma che, in generale, non vale il viceversa (esistono cioè insiemi ricorsivamente enumerabili che non sono ricorsivi).

Da queste definizioni, e dalle definizioni di generazione e di accettazione di un linguaggio date nel testo, segue che un linguaggio è generato da una MT se e solo se costituisce un insieme ricorsivamente enumerabile; e che un linguaggio è accettato da una MT se e solo se costituisce un insieme ricorsivo.

Si può dimostrare che, data qualunque grammatica a struttura di frase (cioè, qualunque **grammatica di tipo 0**), il linguaggio da essa generato costituisce un insieme ricorsivamente enumerabile, ossia può essere generato da una MT (vedi la finestra *Insiemi ricorsivamente numerabili e insiemi ricorsivi*)¹².

Diamo la traccia di una dimostrazione di questo fatto che fa appello alla Tesi di Church (dello stesso risultato si può dare una dimostrazione rigorosa diretta, che non faccia appello alla Tesi di Church).

Data una qualsiasi grammatica a struttura di frase, si può definire come segue un algoritmo intuitivo che generi il linguaggio corrispondente, ossia produca una dopo l'altra tutte le stringhe del linguaggio generato:

- si parte dall'assioma;
- si selezionano tutte le produzioni che possono essere applicate all'assioma, e si applicano una dopo l'altra ottenendo un insieme (finito) E_1 di espressioni;
- per ciascun elemento di E_1 si selezionano tutte le produzioni che possono essergli applicate; poi si applicano una dopo l'altra agli elementi di E_1 le produzioni corrispondenti ottenendo un nuovo insieme E_2 (sempre finito) di espressioni,

e così via.

Man mano che nel corso del processo si ottiene una stringa di soli simboli terminali la si produce in output: tale stringa fa parte del linguaggio generato dalla grammatica. Se una stringa appartiene al linguaggio generato, questo procedimento prima o poi consente produrla in output.

¹² Se definiamo un linguaggio estensionalmente, come un insieme qualunque di stringhe su un alfabeto, allora non tutti i linguaggi possono essere generati da una grammatica. Tutti i possibili insiemi di stringhe su un dato alfabeto costituiscono un insieme più che numerabile. Esiste dunque un insieme più che numerabile di linguaggi. L'insieme di tutte le grammatiche, invece, è un insieme numerabile. Di conseguenza esistono infiniti linguaggi che non sono generati da alcuna grammatica.

In base alla Tesi di Church, tutto ciò che può essere fatto da un algoritmo intuitivo può essere fatto anche da una MT, per cui ogni linguaggio generato da una grammatica di tipo 0 può essere generato anche da una MT, e quindi, in base alla definizione, costituisce un insieme ricorsivamente enumerabile.

Analogamente, si può dimostrare che, data una **grammatica di tipo 1 (non decrescente)**, il linguaggio da essa generato costituisce un insieme ricorsivo, ossia può essere accettato da una MT (vedi ancora la finestra *Insiemi ricorsivamente numerabili e insiemi ricorsivi*).

Anche qui, diamo la traccia di una dimostrazione di questo fatto che fa appello alla Tesi di Church (tenendo conto che lo stesso risultato si può dimostrare in maniera diretta, senza fare appello alla Tesi di Church).

Data una **grammatica di tipo 1**, si verifica facilmente che esiste un algoritmo intuitivo per decidere se una stringa appartiene o meno al linguaggio generato dalla grammatica. Presa in input una stringa s , sia n la sua lunghezza. Si applica un procedimento simile a quello dell'algoritmo precedente con la differenza che, ogni volta che si ottiene un'espressione la cui lunghezza è maggiore di n , la si scarta e non la si prende più in considerazione. Infatti, per come sono definite le grammatiche di tipo 1, l'applicazione delle regole non può mai ridurre la lunghezza delle espressioni, per cui possiamo essere certi che s non potrà essere generata a partire da espressioni più lunghe di n . In questo modo, se s è una stringa del linguaggio, come nel caso precedente verrà generata in un numero finito di passi. Se s invece non fa parte del linguaggio ce ne renderemo conto perché, a un certo punto, non verranno più prodotte stringhe nuove poiché tutte le espressioni ottenute superano la lunghezza n . Il procedimento potrà così terminare producendo una risposta negativa.

In virtù della Tesi di Church, se, per ogni linguaggio di tipo 1, esiste un algoritmo intuitivo che lo accetta, allora esiste anche da una MT che lo accetta. Quindi, possiamo concludere che ogni linguaggio di tipo 1 può essere riconosciuto da una MT, e quindi costituisce un insieme ricorsivo. Si può dimostrare che questo risultato non può essere esteso alle grammatiche di tipo 0; esistono cioè linguaggi di tipo 0 che non sono ricorsivi.

Si può dimostrare che, per riconoscere (o accettare) i linguaggi di tipo 2, sono sufficienti dispositivi di calcolo meno potenti delle MT, detti **automi a pila**, i quali, invece del nastro delle MT, impiegano un supporto di memoria meno flessibile, detto appunto *pila* (in inglese *stack*). A differenza del nastro delle MT, lungo il quale la testina può scorrere liberamente in entrambe le direzioni, una pila si comporta come una catasta di libri appoggiati su un piano: si può accedere soltanto all'elemento in cima alla pila, eventualmente rimuoverlo oppure aggiungerne uno nuovo sopra gli altri (si veda la finestra "Automi a pila").

Per riconoscere i linguaggi generati dalle **grammatiche di tipo 3** sono sufficienti dispositivi ancora meno potenti, detti **automi a stati finiti**. Sostanzialmente, un automa a stati finiti è un dispositivo di calcolo che ha solo un numero finito di stati di memoria interni (analoghi agli stati di una MT), ma non ha alcun supporto di memoria esterno come il nastro delle MT o la pila degli automi a pila (si veda la finestra "Automi a stati finiti").

Possiamo ricapitolare quanto detto in questo paragrafo per mezzo di una tabella:

Tipi di linguaggi	
Tipo 3 (Lineari)	Accettati da un automa a stati finiti
Tipo 2 (Liberi dal contesto)	Accettati da un automa a pila
Tipo 1 (Dipendenti dal contesto)	Accettati da una MT (tutti i linguaggi di tipo 1 sono ricorsivi)
Tipo 0	Generati da una MT (i linguaggi di tipo 0 sono ricorsivamente enumerabili ma, in generale, non ricorsivi)

Automi a stati finiti

Un **automa a stati finiti** (o **automa finito**, in simboli AF) è una macchina calcolatrice astratta con associato un insieme finito $Q = \{q_0, \dots, q_n\}$ di stati interni. In ciascuna fase del calcolo un automa finito si trova in uno e uno solo degli stati di Q . L'automa riceve via via dall'esterno una successione di dati in input. La transizione allo stato successivo viene determinata esclusivamente sulla base dello stato corrente e dall'input ricevuto. In un certo senso, gli automi a stati finiti sono delle MT "senza nastro": gli stati interni sono analoghi a quelli delle MT, ma, mentre per le MT il nastro costituisce una memoria aggiuntiva potenzialmente illimitata sulla quale leggere, scrivere e spostarsi a piacimento, un automa a stati finiti può tenere traccia delle operazioni eseguite solo attraverso i cambiamenti di stato. Pertanto gli automi a stati finiti risultano molto meno potenti delle MT.

Qui prenderemo in considerazione automi a stati finiti che *riconoscono*, o *accettano*, le stringhe di un linguaggio. Si tratta di automi che prendono in input uno dopo l'altro (da sinistra verso destra) i simboli che compongono una stringa. Si dice che un automa AF_L *riconosce*, o *accetta*, un linguaggio L se e solo se, per ogni stringa S , AF_L è in grado di decidere se S appartiene a L o meno.

Tra gli stati q_0, \dots, q_n di un automa si deve specificare uno *stato iniziale* e un insieme non vuoto di *stati finali* (che può comprendere eventualmente anche lo stato iniziale). Come stato iniziale utilizzeremo q_0 . All'inizio del calcolo un automa si trova sempre nello stato q_0 ; a questo punto legge in input il primo simbolo della stringa S e il calcolo comincia. Ogni automa è attrezzato con un "programma", un insieme di istruzioni che, sulla base dello stato corrente e del simbolo letto, specifica lo stato successivo. Una volta avvenuto il cambio di stato, viene preso in input il simbolo successivo della stringa, e così via. Le istruzioni di un automa finito hanno la forma:

$$q_i \quad s_j \quad q_k$$

dove q_i è lo stato corrente, s_j è il simbolo preso in input e q_k è lo stato successivo: se lo stato è q_i e il simbolo in input è s_j , allora l'automa assume lo stato q_k (e passa a leggere in input il simbolo successivo).

Quando un automa AF_L si ferma, se la stringa S degli input è stata esaminata interamente, e se AF_L si trova in uno stato finale, allora S fa parte del linguaggio L riconosciuto (o accettato) da AF_L ; altrimenti (se cioè ci sono simboli di S che non

sono ancora stati presi in input oppure se AF_L non si trova in uno stato finale), S non fa parte di L .

Vediamo ad esempio un automa finito AF_{LD} che accetta il linguaggio generato dalla grammatica LD del par. 3.3. LD genera tutte e sole le stringhe in cui n occorrenze di a sono seguite da m occorrenze di b (con $m, n \geq 1$). AF_{LD} ha tre stati, q_0 , q_1 e q_2 , dei quali q_2 è l'unico stato finale. Le istruzioni di AF_{LD} sono le seguenti:

- 1) $q_0 \quad a \quad q_1$
- 2) $q_1 \quad a \quad q_1$
- 3) $q_1 \quad b \quad q_2$
- 4) $q_2 \quad b \quad q_2$

Supponiamo di voler stabilire se la stringa $S = aabbb$ fa parte del linguaggio accettato da AF_{LD} . All'inizio del calcolo AF_{LD} si trova nello stato iniziale q_0 , e prende in input il primo simbolo della stringa:

$$\begin{array}{c} a \quad a \quad b \quad b \quad b \\ \uparrow \\ q_0 \end{array}$$

In base all'istruzione 1), AF_{LD} passa nello stato q_1 , e va a leggere il secondo simbolo di S :

$$\begin{array}{c} a \quad a \quad b \quad b \quad b \\ \uparrow \\ q_1 \end{array}$$

Ora, per l'istruzione 2), AF_{LD} resta in q_1 , e va a leggere il terzo simbolo:

$$\begin{array}{c} a \quad a \quad b \quad b \quad b \\ \uparrow \\ q_1 \end{array}$$

Ora si applica l'istruzione 3), per cui AF_{LD} passa in q_2 , poi va a leggere il quarto simbolo:

$$\begin{array}{c} a \quad a \quad b \quad b \quad b \\ \uparrow \\ q_2 \end{array}$$

A questo punto si applica l'istruzione 4): AF_{LD} resta in q_2 , e va a leggere il quinto simbolo. Dopo di che viene applicata ancora una volta l'istruzione 4): lo stato resta q_2 , ma, non essendoci più simboli da leggere, l'automa si ferma e il calcolo termina. Poiché q_2 è uno stato finale e la stringa S è stata esaminata interamente, S fa parte del linguaggio accettato da AF_{LD} .

È facile constatare che stringhe come *abba*, *aaaa* e *bbbaa* non vengono accettate da AF_{LD} : nel primo caso, quando AF_{LD} si ferma ha raggiunto uno stato finale, ma la stringa non è stata esaminata interamente; nel secondo caso la stringa viene esaminata interamente, ma senza che venga raggiunto uno stato finale; nel terzo caso AF_{LD} si ferma senza aver raggiunto uno stato finale e senza avere esaminato l'intera stringa.

Un automa a stati finiti può essere rappresentato per mezzo di un **diagramma a stati** nella maniera seguente. Gli stati non terminali vengono rappresentati con cerchi semplici, gli stati terminali con cerchi doppi. Ogni istruzione $q_i \xrightarrow{s_j} q_k$ viene rappresentata con un arco orientato (ossia una freccia) contrassegnato con il simbolo s_j che va dal nodo q_i al nodo q_k , come in fig. 3-4.

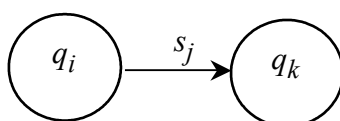


Figura 3-4

In questa modo l'automa AF_{LD} può essere rappresentato come in fig. 3-5.

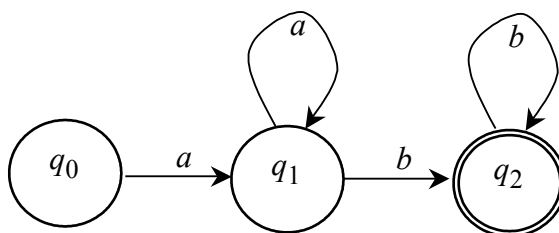


Figura 3-5

Si verifica facilmente che una stringa s_1, \dots, s_k è accettata da un automa finito se e solo se nel corrispondente diagramma a stati esiste un percorso che parte dallo stato iniziale e termina in uno finale, in cui gli archi attraversati sono contrassegnati, nell'ordine, con i simboli s_1, \dots, s_k .

Automi a pila

Gli **automi a pila**, in simboli AP (in inglese *pushdown automata*) possono essere visti come automi a stati finiti dotati di una memoria ausiliaria sulla quale durante il calcolo possono scrivere, leggere o cancellare simboli. Tale memoria è organizzata come una **pila**: i simboli sono memorizzati uno sopra l'altro, e in ogni momento si può leggere o rimuovere solo il simbolo collocato in cima alla pila, oppure si possono aggiungere sopra gli altri nuovi simboli. Per accedere ai simboli sottostanti si devono prima rimuovere i simboli collocati sopra. La presenza della pila rende questi automi più potenti degli automi a stati finiti (per cui possono accettare linguaggi che gli automi a stati finiti non accettano). Essi però sono meno potenti delle MT in quanto, rispetto al nastro di queste ultime, la pila è comunque meno flessibile.

Come gli automi a stati finiti, ogni automa a pila ha un insieme finito Q di stati interni che deve comprendere uno stato iniziale e almeno uno stato finale. Per accettare una stringa anche gli automi a pila ne leggono i simboli scorrendola da sinistra verso destra.

I simboli che si possono memorizzare nella pila appartengono a un alfabeto diverso rispetto ai simboli dell'input. Indicheremo con $Al_P = \{A_1, \dots, A_m\}$ l'insieme (finito) di simboli che un automa può scrivere nella propria pila.

Nelle istruzioni di un automa a pila la transizione allo stato interno successivo può dipendere anche dal simbolo che si trova in cima alla pila. Inoltre un'istruzione può causare anche la cancellazione del simbolo in cima alla pila o la sua sostituzione con altri simboli. In generale, le istruzioni hanno la forma seguente:

$$(q_i, s_j, A_k) \rightarrow (q_l, A_h \dots A_t)$$

dove q_i e q_l sono stati interni, s_j è un simbolo dell'input e $A_k, A_h \dots A_t$ sono simboli di Al_P . Il significato di tale istruzione è il seguente: se lo stato interno è q_i , il simbolo preso in input è s_j e il simbolo in cima alla pila è A_k , passa nello stato q_l , cancella A_k e aggiungi in cima alla pila, nell'ordine, $A_h \dots A_t$. Poi passa a leggere in input il simbolo successivo.

È possibile avere istruzioni del tipo:

$$(q_i, s_j, A_k) \rightarrow (q_l, -)$$

e

$$(q_i, s_j, -) \rightarrow (q_l, A_h \dots A_t)$$

Nel primo caso, se lo stato è q_i , il simbolo in input è s_j e il simbolo in cima alla pila è A_k , si passa nello stato q_l e si cancella A_k dalla pila (senza aggiungere nulla). Nel secondo caso, se lo stato è q_i e il simbolo in input è s_j , allora, a prescindere da quale sia il simbolo in cima alla pila, si passa nello stato q_l e si aggiungono alla pila $A_h \dots A_t$.

All'inizio del calcolo la pila deve essere vuota. Una stringa fa parte del linguaggio accettato da un automa a pila AP se e solo se, quando AP si ferma, sono vere contemporaneamente queste tre condizioni: a) tutta la stringa di input è stata letta; b) l'automata è in uno stato finale; c) la pila è vuota.

Vediamo un automa AP_{LC} che accetta il linguaggio LC del paragrafo 3, formato da tutte le stringhe con n occorrenze di a seguite da n occorrenze di b (con $n \geq 1$). AP_{LC} ha due stati interni q_0 e q_1 , di cui q_1 è l'unico stato finale. A è l'unico simbolo che AP_{LC} può scrivere nella sua pila. Le istruzioni di AP_{LC} sono le seguenti:

- 1) $(q_0, a, -) \rightarrow (q_0, A)$
- 2) $(q_0, b, A) \rightarrow (q_1, -)$
- 3) $(q_1, b, A) \rightarrow (q_1, -)$

La fig. 3-6 mostra il calcolo svolto da AP_{LC} a partire dalla stringa di input $aabb$.

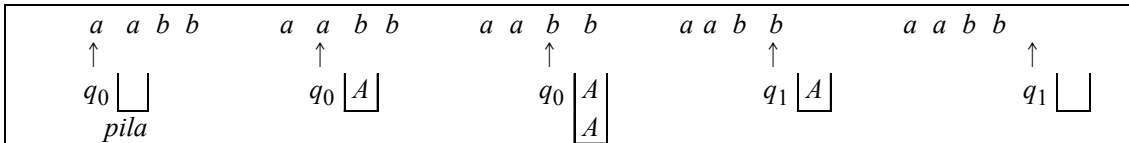


Figura 3-6

Per ciascuna fase del calcolo a fianco dello stato interno abbiamo rappresentato la situazione della pila. All'inizio AP_{LC} prende in input il primo simbolo a , lo stato è q_0 e la pila è vuota. Viene applicata l'istruzione 1): lo stato rimane q_0 e viene aggiunto A in cima alla pila. Dopo di che AP_{LC} prende in input il secondo simbolo a e viene applicata di nuovo l'istruzione 1). Poi AP_{LC} legge il terzo simbolo, che è b e va perciò applicata l'istruzione 2): si ha un cambiamento di stato e viene cancellato il simbolo in cima alla pila. Al passo successivo si applica l'istruzione 3), che mantiene l'automa nello stato q_1 e comporta la cancellazione dell'ultimo simbolo dalla pila. A questo punto l'automa si ferma: la pila è vuota, q_1 è uno stato finale e tutti i simboli della stringa sono stati esaminati. Pertanto la stringa $aabb$ fa parte del linguaggio accettato da AP_{LC} .

L'esempio chiarisce come funziona AP_{LC} : l'automa usa la pila per contare le occorrenze di a nella prima parte della stringa. Quando incontra la prima b , AP_{LC} passa dallo stato q_0 allo stato q_1 , e comincia a togliere dalla pila una A per ogni b che trova. Pertanto, quando AP_{LC} si ferma, affinché tutta la stringa sia stata letta e la pila sia vuota, il numero delle a deve essere lo stesso del numero delle b .

L'uso della pila è essenziale per accettare il linguaggio LC , che infatti non può essere accettato da alcun automa finito.

3.6 Conclusioni

La teoria delle grammatiche formali e la proposta, dovuta essenzialmente a Noam Chomsky, di impiegarle per descrivere la struttura sintattica delle lingue naturali è alla base della linguistica teorica contemporanea. La proposta chomskyana di utilizzare le grammatiche formali per spiegare psicologicamente la capacità degli esseri umani di produrre e comprendere il linguaggio è stata determinante anche per lo sviluppo delle scienze cognitive e della concezione dei processi mentali come processi computazionali.

Le grammatiche formali hanno inoltre una notevole rilevanza applicativa in informatica, in quanto vengono utilizzate per descrivere la sintassi dei linguaggi di programmazione. Il processo di traduzione automatica dei programmi da un linguaggio di programmazione di alto livello al codice assembler e al linguaggio macchina presuppone una fase di analisi sintattica automatica (*parsing*) resa possibile dal fatto che la sintassi dei linguaggi di alto livello è espressa mediante una grammatica formale. Affinché tale analisi possa stabilire se un programma è sintatticamente corretto o meno, i linguaggi di programmazione devono essere decidibili (non possono cioè essere linguaggi di tipo 0). Di solito i linguaggi di programmazione sono di tipo 1, ossia linguaggi sensibili al contesto. Tuttavia si preferisce caratterizzarne la sintassi per mezzo di grammatiche libere dal contesto (di tipo 2) cui vengono affiancate delle

restrizioni di tipo contestuale¹³. Questo consente di rendere più efficiente il processo di *parsing*.

Nell'ambito della linguistica, stabilire a quale classe appartengano le lingue naturali è una questione empirica ben più complessa, e a tutt'oggi ancora aperta. Vi sono tuttavia buone ragioni per ritenere che, in generale, le lingue naturali non siano libere dal contesto, cioè che non sia sufficiente una grammatica di tipo 2 per generarle, e richiedano quindi una grammatica di tipo 1, sensibile al contesto¹⁴.

¹³ La cosiddetta forma BNF (*Bacchus Normal Form*, o *Bacchus-Naur Form*) che viene usata abitualmente per esprimere la sintassi dei linguaggi di programmazione altro non è che una notazione equivalente per le grammatiche libere dal contesto.

¹⁴ Su grammatiche formali e linguistica si vedano i capp. 16-22 di Partee *et al.* [1990] e in particolare, per questo tipo di problemi, i paragrafi 17.3.2 e 18.6.

ESERCIZI RELATIVI ALLA TERZA PARTE

Esercizio 3.1. Verificare che le seguenti stringhe appartengono al linguaggio generato da G : a) *un gatto dorme*; b) *un topo mangia il gatto*.

Esercizio 3.2. Come formulata nel paragrafo 2, la grammatica G non genera frasi quali *un topo mangia* o *il gatto rincorre*, dove un verbo transitivo non è seguito da un complemento oggetto. Modificare G in modo che generi anche questo tipo di frasi.

Esercizio 3.3. Verificare che le seguenti stringhe appartengono al linguaggio generato da G' : a) *il topo canta ma il gatto dorme*; b) *un gatto rincorre il topo e il topo dorme*; c) *un gatto rincorre un topo oppure il topo mangia il gatto*; d) *un topo dorme ma il gatto canta e il topo mangia il gatto*.

Esercizio 3.4. Verificare che le seguenti stringhe appartengono al linguaggio generato da G'' : a) *il topo che mangia il gatto rincorre un topo*; b) *un topo rincorre il gatto che canta*; c) *il gatto che rincorre il topo mangia un topo che canta*; d) *un gatto mangia il topo che rincorre il gatto che dorme*.

Esercizio 3.5. Sviluppare gli alberi sintattici che corrispondono alle derivazioni degli esercizi 3.1 e 3.3-4.

Esercizio 3.6. Completare gli alberi sintattici corrispondenti alle due possibili analisi nella grammatica G' della stringa *un gatto corre e un topo canta oppure un gatto canta*.

Esercizio 3.7. Sviluppare due derivazioni in G' che corrispondano alle due possibili analisi sintattiche della stringa *un gatto corre e un topo canta oppure un gatto canta*.

Esercizio 3.8. Determinare una grammatica che generi tutte e sole le espressioni algebriche ben formate che si possono scrivere con il linguaggio Al .

Esercizio 3.9. Individuare una derivazione della stringa *aaaabbbbcccc* nella grammatica GnD .

Esercizio 3.10. Determinare le produzioni di una grammatica lineare sinistra LS che generi lo stesso linguaggio di LD .

Esercizio 3.11. Determinare le produzioni di una grammatica lineare destra che generi lo stesso linguaggio di G .

Esercizio 3.12. Individuare una derivazione della stringa *un topo rincorre il gatto* nella grammatica dell'esercizio precedente.

Esercizio 3.13. Sia AF un automa finito con $Q = \{q_0, q_1\}$ e q_1 come unico stato finale, caratterizzato dalle istruzioni:

q_0	a	q_0	q_1	b	q_0
q_0	b	q_1	q_1	a	q_1

Rappresentare AF mediante un diagramma a stati e stabilire qual è il linguaggio accettato da AF.

Esercizio 3.14. Specificare gli stati e scrivere le istruzioni di un automa a stati finiti AF_G che accetti il linguaggio generato dalla grammatica G del par. 3.2. Rappresentare AF_G mediante un diagramma a stati.

Esercizio 3.15. Specificare gli stati, scrivere le istruzioni e disegnare il diagramma a stati di un AF che accetti tutte le stringhe sul linguaggio $\{a, b\}$ con esattamente due occorrenze di a .

Esercizio 3.16. Specificare gli stati, scrivere le istruzioni e disegnare il diagramma a stati di un AF che accetti tutte le stringhe sul linguaggio $\{a, b\}$ con almeno due occorrenze di a .

Esercizio 3.17. Specificare gli stati, scrivere le istruzioni e disegnare il diagramma a stati di un AF che accetti tutte le stringhe sul linguaggio $\{a, b\}$ in cui compare almeno una sequenza di tre a consecutive.

Esercizio 3.18. Verificare che le stringhe 1) $bbaa$, 2) $aaabb$, 3) $aabbb$ e 4) $aabba$ non fanno parte del linguaggio accettato dall'automa a pila AF_{LC} .

Esercizio 3.19. Stabilire qual è il linguaggio accettato da un AP con due stati interni q_0 e q_1 , di cui q_1 è l'unico stato finale, caratterizzato dalle seguenti istruzioni:

$$\begin{aligned}(q_0, a, -) &\rightarrow (q_0, AA) \\(q_0, b, A) &\rightarrow (q_1, -) \\(q_1, b, A) &\rightarrow (q_1, -)\end{aligned}$$

Esercizio 3.20. Determinare gli stati e le istruzioni di un automa a pila che accetti il linguaggio formato da tutte le stringhe in cui n occorrenze di a sono seguite da m occorrenze di b , le quali sono seguite a loro volta da altre n occorrenze di a (con $m, n \geq 1$). Ad esempio devono appartenere al linguaggio accettato le stringhe $aabbbbaa$, $aaaabbaaaa$, eccetera.

Esercizio 3.21. Stabilire qual è il linguaggio accettato da un AP con quattro stati interni q_0, q_1, q_2 e q_3 , di cui q_3 è l'unico stato finale, caratterizzato dalle seguenti istruzioni:

$$\begin{aligned}(q_0, a, -) &\rightarrow (q_0, A) \\(q_0, b, A) &\rightarrow (q_1, -) \\(q_1, b, A) &\rightarrow (q_1, -) \\(q_1, a, -) &\rightarrow (q_2, A) \\(q_2, a, -) &\rightarrow (q_2, A) \\(q_2, b, A) &\rightarrow (q_3, -) \\(q_3, b, A) &\rightarrow (q_3, -)\end{aligned}$$

SOLUZIONI DEGLI ESERCIZI RELATIVI ALLA TERZA PARTE

Esercizio 3.1. Riportiamo a titolo di esempio una derivazione di a) in G :

- | | |
|----------------------|-----------------------------|
| 1) E | 5) <i>un gatto</i> SV |
| 2) SN SV | 6) <i>un gatto</i> VerbIntr |
| 3) Articolo Nome SV | 7) <i>un gatto dorme</i> |
| 4) <i>un</i> Nome SV | |

Esercizio 3.2. Sostituire la regola $SV \rightarrow \text{VerbTrans SN} \mid \text{VerbIntr}$ di G con la regola $SV \rightarrow \text{VerbTrans SN} \mid \text{VerbTrans} \mid \text{VerbIntr}$.

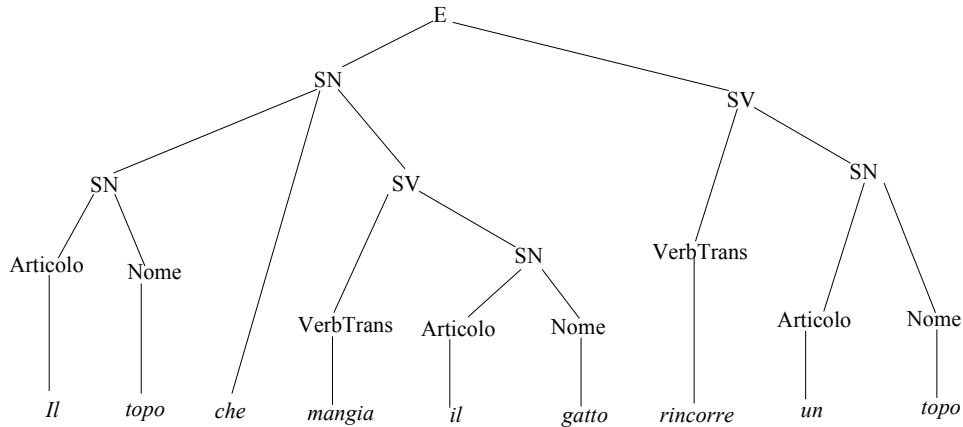
Esercizio 3.3. Riportiamo a titolo di esempio una derivazione di a) in G' :

- | | |
|---|---|
| 1) E | 9) <i>il topo canta ma</i> E |
| 2) E Connettivo E | 10) <i>il topo canta ma</i> SN SV |
| 3) SN SV Connettivo E | 11) <i>il topo canta ma</i> Articolo Nome SV |
| 4) Articolo Nome SV Connettivo E | 12) <i>il topo canta ma il</i> Nome SV |
| 5) <i>il</i> Nome SV Connettivo E | 13) <i>il topo canta ma il gatto</i> SV |
| 6) <i>il topo</i> SV Connettivo E | 14) <i>il topo canta ma il gatto</i> VerbIntr |
| 7) <i>il topo</i> VerbIntr Connettivo E | 15) <i>il topo canta ma il gatto dorme</i> |
| 8) <i>il topo canta</i> Connettivo E | |

Esercizio 3.4. Riportiamo a titolo di esempio una derivazione di a) in G'' :

- | | |
|---|--|
| 1) E | 10) <i>il topo che mangia il</i> Nome SV |
| 2) SN SV | 11) <i>il topo che mangia il gatto</i> SV |
| 3) SN <i>che</i> SV SV | 12) <i>il topo che mangia il gatto</i> VerbTrans SN |
| 4) Articolo Nome <i>che</i> SV SV | 13) <i>il topo che mangia il gatto rincorre</i> SN |
| 5) <i>il</i> Nome <i>che</i> SV SV | 14) <i>il topo che mangia il gatto rincorre</i> Articolo
Nome |
| 6) <i>il topo che</i> SV SV | 15) <i>il topo che mangia il gatto rincorre un</i>
Nome |
| 7) <i>il topo che</i> VerbTrans SN SV | 16) <i>il topo che mangia il gatto rincorre un topo</i> |
| 8) <i>il topo che mangia</i> SN SV | |
| 9) <i>il topo che mangia</i> Articolo Nome SV | |

Esercizio 3.5. Riportiamo a titolo di esempio l'albero che corrisponde alla derivazione della stringa a) dell'esercizio 3.4:



Esercizio 3.7. (*Traccia*). In entrambi i casi la derivazione inizia con questi passi: 1) E; 2) E Connettivo E. Dopo di che, nel caso dell'analisi corrispondente all'albero (a) di fig. 9-3 si applica la produzione $E \rightarrow SN\ SV$ all'occorrenza di sinistra di E in 2), e la produzione $E \rightarrow E\ Connettivo\ E$ all'occorrenza di destra; nel caso dell'albero (b) si fa il viceversa.

Esercizio 3.8. $S \rightarrow Exp\ Op\ Exp$
 $Exp \rightarrow (Exp\ Op\ Exp) \mid Lettera$
 $Op \rightarrow + \mid - \mid * \mid /$
 $Lettera \rightarrow a \mid b \mid c \mid d \mid e$

Esercizio 3.9.

- | | |
|------------------------|--------------------------|
| 1) S | 8) <i>aaaabBBcccBc</i> |
| 2) <i>aSBc</i> | 9) <i>aaaabBBccBcc</i> |
| 3) <i>aaSBcBc</i> | 10) <i>aaaabBBcBccc</i> |
| 4) <i>aaaSBcBcBc</i> | 11) <i>aaaabBBBcccc</i> |
| 5) <i>aaaabcBcBcBc</i> | 12) <i>aaaabbbBcccc</i> |
| 6) <i>aaaabBccBcBc</i> | 13) <i>aaaabbbbBcccc</i> |
| 7) <i>aaaabBcBccBc</i> | 14) <i>aaaabbbbcccc</i> |

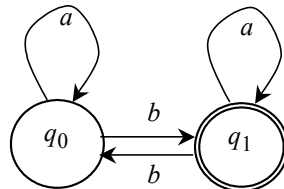
Esercizio 3.10. $S \rightarrow Sb \mid A$
 $A \rightarrow Aa \mid a$

Esercizio 3.11. $S \rightarrow il\ S1 \mid un\ S1$
 $S1 \rightarrow topo\ S2 \mid gatto\ S2$
 $S2 \rightarrow dorme \mid canta \mid rincorre\ S3 \mid mangia\ S3$
 $S3 \rightarrow il\ S4 \mid un\ S4$
 $S4 \rightarrow topo \mid gatto$

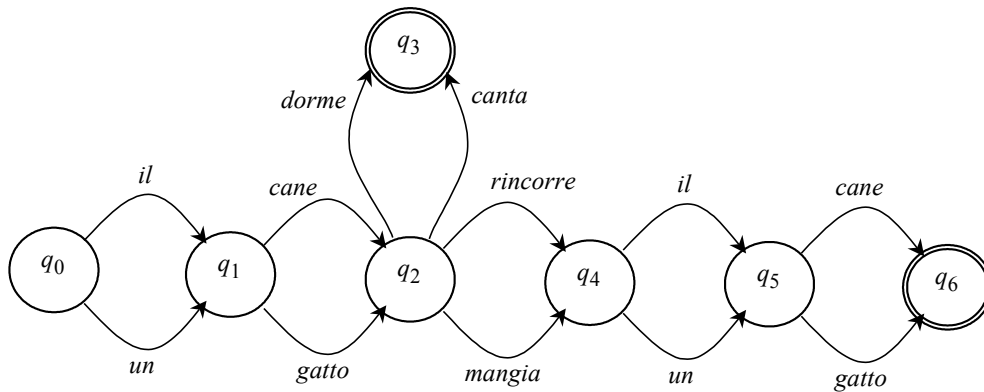
Esercizio 3.12.

- 1) S
- 2) un S1
- 3) un topo S2
- 4) un topo rincorre S3
- 5) un topo rincorre il S4
- 6) un topo rincorre il gatto

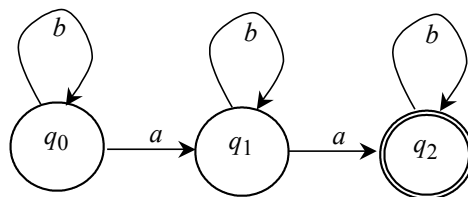
Esercizio 3.13. L'automa AF accetta le stringhe sull'alfabeto $\{a, b\}$ con un numero dispari di occorrenze di b . Ciò si constata facilmente se si esamina il diagramma a stati di AF:



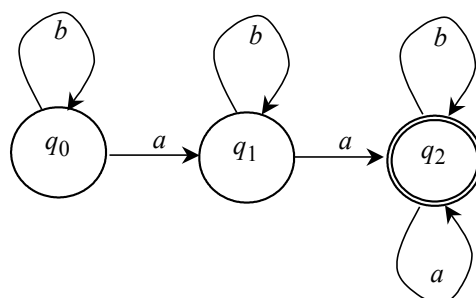
Esercizio 3.14. Il diagramma a stati di un automa AF_G è il seguente:



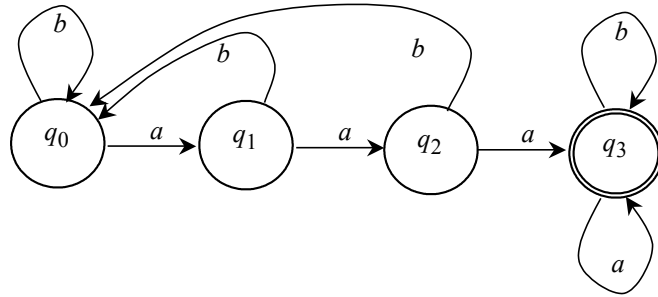
Esercizio 3.15. Il diagramma a stati di un AF con le caratteristiche richieste dall'esercizio è il seguente:



Esercizio 3.16. Il diagramma a stati di un AF con le caratteristiche richieste dall'esercizio è il seguente:



Esercizio 3.17. Il diagramma a stati di un AF con le caratteristiche richieste dall'esercizio è il seguente:



Esercizio 3.18. 1) L'automa si ferma subito perché non c'è nessuna istruzione che preveda il caso che lo stato sia q_0 e il simbolo in input b ; 2) quando l'automa si ferma la pila non è vuota; 3) l'automa si blocca sull'ultima b perché non ci sono più A nella pila e non può essere applicata la terza istruzione; 4) l'automa si blocca sull'ultima a perché non c'è nessuna istruzione che preveda che lo stato sia q_1 e il simbolo in input a .

Esercizio 3.19. L'automa accetta il linguaggio di tutte le stringhe formate da n occorrenze di a seguite da m occorrenze di b , dove $m = 2n$ e $n \geq 1$. Ad esempio appartengono al linguaggio accettato le stringhe $aabbbb$, $aaabbbbb$, eccetera.

Esercizio 3.20. Un possibile automa per il compito richiesto ha tre stati interni q_0 , q_1 e q_2 , di cui q_2 è l'unico stato finale, ed è caratterizzato dalle seguenti istruzioni:

$$\begin{aligned} (q_0, a, -) &\rightarrow (q_0, A) \\ (q_0, b, A) &\rightarrow (q_1, A) \\ (q_1, b, A) &\rightarrow (q_1, A) \\ (q_1, a, A) &\rightarrow (q_2, -) \\ (q_2, a, A) &\rightarrow (q_2, -) \end{aligned}$$

Esercizio 3.21. L'automa accetta il linguaggio di tutte le stringhe formate nell'ordine da n occorrenze di a , n occorrenze di b , m occorrenza di a e m occorrenza di b (con $m, n \geq 1$). Ad esempio appartengono al linguaggio accettato le stringhe $aabbab$, $aabbaabbb$, eccetera.

Ulteriori letture

Per chi volesse affrontare gli aspetti tecnici della teoria della computabilità, alcune trattazioni approfondite sono (Kleene 1952 [parte III], Hermes 1961; Rogers 1967; Minsky 1967; Lewis e Papadimitriou 1981; Davis e Weyuker 1983; Odifreddi 1989). (Frixione e Palladino 2004) è un'introduzione alla teoria che non presuppone conoscenze logico-matematiche. (Davis 1965) è una raccolta di articoli storici sulla teoria della computabilità. Una raccolta di articoli classici di logica e filosofia della matematica, con molti punti di contatto con i temi qui trattati è (van Heijenoort 1967). Sui problemi di filosofia della matematica che hanno portato alla formulazione della teoria della computabilità si veda (Borga e Palladino 1997). (Turing 1994) è una raccolta di scritti di Alan Turing. Una biografia di Turing è stata scritta da Andrew Hodges (1983). L'articolo dove viene proposto il test di Turing è compreso in (Somenzi e Cordeschi 1994). Partee *et al.* (1990) è un'introduzione agli strumenti matematici per la linguistica, che include un'ampia trattazione delle grammatiche formali. Infine, un libro che tratta in modo affascinante e molto particolare i temi della computabilità e dei teoremi di limitazione della logica in relazione alla teoria della mente è (Hofstadter 1979).

Bibliografia

- Borga, M. e Palladino, D. (1997). *Oltre il mito della crisi. Fondamenti e filosofia della matematica nel ventesimo secolo*. La Scuola, Brescia.
- Casalegno, P. (1997). *Filosofia del linguaggio. Un'introduzione*, La Nuova Italia Scientifica (poi Carocci), Roma.
- Chierchia, G. e McConnell-Ginet, S. (1990). *Meaning and Grammar: An Introduction to Semantics*, MIT Press, Cambridge, Mass.; tr. it.: *Significato e grammatica: semantica del linguaggio naturale*, F. Muzzio, Padova, 1993.
- Chomsky, N. (1959). On certain formal properties of grammars, *Information and Control* 2, pp. 137-167; tr. it. in De Palma (1974).
- Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345-363.
- Curry, H.B. (1929). An analysis of logical substitution. *American Journal of Mathematics*, 51: 363-384.
- Curry, H.B. (1930). Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics*, 52:509-536, 789-834.
- Curry, H.B. (1932). Some additions to the theory of combinators. *American Journal of Mathematics*, 54:551-558.
- Davis, M. (a cura di) (1965). *The Undecidable*. Raven Press, Hewlett, New York.
- Davis, M. e Weyuker, E. (1983). *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Academic Press, New York.
- De Palma, A. (a cura di) (1974). *Linguaggio e sistemi formali*, Einaudi, Torino.
- Frixione, M. e Palladino, D. (2004). *Funzioni, macchine, algoritmi. Introduzione alla teoria della computabilità*. Carocci, Roma.
- Gandy, R. (1980). Church's thesis and principles for mechanisms. In *The Kleene Symposium*, 123-148, North Holland, Amsterdam.

- van Heijenoort, J. (a cura di) (1967). *From Frege to Gödel*. Harvard University Press, Harvard.
- Hermes, H. (1961). *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit*. Springer, Berlin, Heidelberg. Tr. it. *Numerabilità, decidibilità, computabilità*, Boringhieri, Torino, 1973.
- Hodges, A. (1983). *Alan Turing: The Enigma*. New York, Simon and Shuster. Tr. it. *Storia di un enigma. Vita di Alan Turing (1912-1954)*. Bollati, Torino, 1991.
- Hofstadter, D. (1979). *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books. Tr. it. *Gödel, Escher, Bach: un'eterna ghirlanda brillante*, Adelphi, Milano, 1984.
- Kleene, S.C. (1952). *Introduction to metamathematics*. North Holland, Amsterdam.
- Lewis, H.R. e Papadimitriou, C.H. (1981). *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, NJ.
- Markov, A.A. (1951). Theory of algorithms. *American Mathematical Society Translations*, seconda serie, 15(1960):1-14 (trad. ingl. dell'originale russo).
- Markov, A.A. (1954). *Theory of algorithms*. National Science Foundation and Israel Program for Scientific Translation (1961) (trad. ingl. dell'originale russo).
- Minsky, M. (1967). *Computation. Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs.
- Odifreddi, P. (1989). *Classical Recursion Theory : The Theory of Functions and Sets of Natural Numbers*. Elsevier, Amsterdam.
- Partee, B.H., Meulen, A. e Wall, R. (1990). *Mathematical Methods in Linguistics*, Kluwer, Dordrecht.
- Post, E. (1936). Finite combinatory processes - formulation 1. *Journal of Symbolic Logic*, 1:103-105.
- Post, E. (1943). Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65:197-215.
- Post, E. (1946). A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:284-316.
- Rogers H. (1967). *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York. Tr. it. *Teoria delle funzioni ricorsive e della computabilità effettiva*, Tecniche Nuove, Milano, 1992.
- Schönfinkel, M. (1924). Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305-316.
- Somenzi, V. e Cordeschi, R. (a cura di) (1994). *La filosofia degli automi. Origini dell'intelligenza artificiale*. Bollati, Torino.
- Turing, A. (1936-7). On computable number, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, serie 2, 42:230-365; A correction, *ibid.*, 43:544-546 (ristampato in Davis 1965).
- Turing, A. (1937). Computability and λ -definability. *Journal of Symbolic Logic*, 2:153-163.
- Turing, A. (1950). Computing machinery and intelligence. *Mind*, 59:433-460.
- Turing, A. (1994). *Intelligenza meccanica*. A cura di G. Lolli, Bollati, Torino.